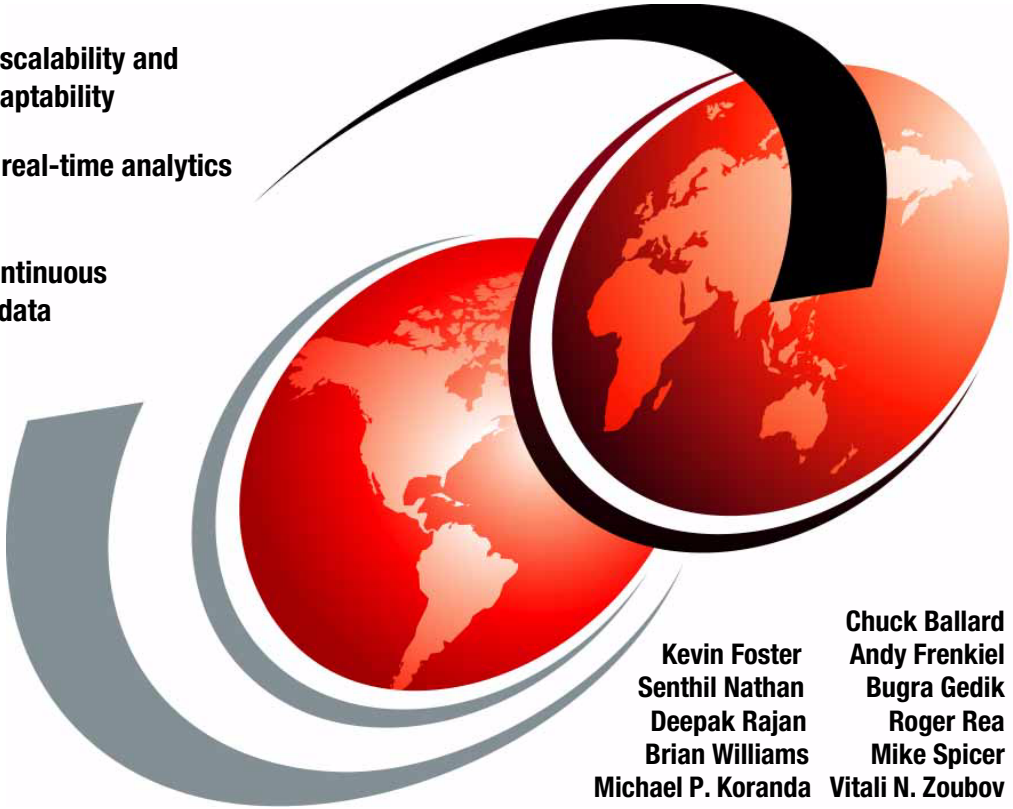
IBM® Information Management Software

IBM InfoSphere Streams

Assembling Continuous Insight in the Information Revolution

- Supporting scalability and dynamic adaptability
- Performing real-time analytics on Big Data
- Enabling continuous analysis of data



Chuck Ballard
Kevin Foster
Andy Frenkiel
Senthil Nathan
Bugra Gedik
Deepak Rajan
Roger Rea
Brian Williams
Mike Spicer
Michael P. Koranda
Vitali N. Zoubov

Redbooks

ibm.com/redbooks

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000001
IPR2024-01507 (Amazon.com, Inc., Amazon.com Services LLC v. Nokia Technologies Oy)



International Technical Support Organization

IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution

October 2011

SG24-7970-00

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000003

Note: Before using this information and the product it supports, read the information in “Notices” on page ix.

First Edition (October 2011)

This edition applies to Version 2.0.0 of InfoSphere Streams (Product Number 5724-Y95).

© Copyright International Business Machines Corporation 2011. All rights reserved.
Note to U.S. Government Users Restricted Rights -- Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000004

Contents

Notices ix
Trademarks x

Preface xi
The team who wrote this book xii
Now you can become a published author, too! xvii
Comments welcome xvii
Stay connected to IBM Redbooks xviii

Chapter 1. Introduction 1
1.1 Stream computing 2
1.1.1 Business landscape 6
1.1.2 Information environment 9
1.1.3 The evolution of analytics 14
1.1.4 Relationship to Big Data 17
1.2 IBM InfoSphere Streams 17
1.2.1 Overview of Streams 19
1.2.2 Why use Streams 24
1.2.3 Examples of Streams implementations 27

Chapter 2. Streams concepts and terms 33
2.1 IBM InfoSphere Streams: Solving new problems 34
2.2 Concepts and terms 39
2.2.1 Streams instances, hosts, host types, and admin services 40
2.2.2 Projects, applications, streams, and operators 44
2.2.3 Applications, Jobs, Processing Elements, and Containers 47
2.3 End-to-end example: Streams and the lost child 48
2.3.1 The Lost Child application 50
2.3.2 Example Streams Processing Language code review 53
2.4 IBM InfoSphere Streams tools 58
2.4.1 Creating an example application inside Streams Studio 59

Chapter 3. Streams applications 67
3.1 Streams application design 69
3.1.1 Design aspects 70
3.1.2 Data sources 72
3.1.3 Output 75
3.1.4 Existing analytics 78
3.1.5 Performance requirements 79

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000005

3.2 SPL design patterns 80
3.2.1 Reducing data using a simple filter 82
3.2.2 Reducing data using a simple schema mapper 86
3.2.3 Data Parallel 90
3.2.4 Data Pipeline 97
3.2.5 Outlier detection 101
3.2.6 Enriching streams from a database 107
3.2.7 Tuple Pacer 111
3.2.8 Processing multiplexed streams 116
3.2.9 Updating import stream subscriptions 120
3.2.10 Updating export stream properties 127
3.2.11 Adapting to observations of runtime metrics 134

Chapter 4. InfoSphere Streams deployment 139
4.1 Architecture, instances, and topologies 140
4.1.1 Runtime architecture 140
4.1.2 Streams instances 143
4.1.3 Deployment topologies 147
4.2 Streams runtime deployment planning 151
4.2.1 Streams environment 151
4.2.2 Sizing the environment 151
4.2.3 Deployment and installation checklists 152
4.3 Pre- and post-installation of the Streams environment 160
4.3.1 Installation and configuration of the Linux environment 160
4.3.2 InfoSphere Streams installation 163
4.3.3 Post-installation configuration 164
4.3.4 Streams user configuration 168
4.4 Streams instance creation and configuration 170
4.4.1 Streams shared instance configuration 170
4.4.2 Streams private developer instance configuration 175
4.5 Application deployment capabilities 179
4.5.1 Dynamic application composition 181
4.5.2 Operator host placement 184
4.5.3 Operator partitioning 189
4.5.4 Parallelizing operators 190
4.6 Failover, availability, and recovery 194
4.6.1 Restarting and relocating processing elements 194
4.6.2 Recovering application hosts 197
4.6.3 Recovering management hosts 199

Chapter 5. Streams Processing Language 203
5.1 Language elements 204
5.1.1 Structure of an SPL program file 206

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000006

5.1.2 Streams data types 209
5.1.3 Stream schemas 210
5.1.4 Streams punctuation markers 212
5.1.5 Streams windows 212
5.1.6 Stream bundles 220
5.2 Streams Processing Language operators 220
5.2.1 Operator usage language syntax 225
5.2.2 The Source operator 227
5.2.3 The Sink operator 231
5.2.4 The Functor operator 232
5.2.5 The Aggregate operator 234
5.2.6 The Split operator 236
5.2.7 The Punctor operator 240
5.2.8 The Delay operator 241
5.2.9 The Barrier operator 241
5.2.10 The Join operator 242
5.2.11 The Sort operator 245
5.2.12 Additional SPL operators 247
5.3 The preprocessor 248
5.4 Export and Import operators 255
5.5 Example Streams program 259
5.6 Debugging a Streams application 260
5.6.1 Using Streams Studio to debug 262
5.6.2 Using streamtool to debug 266
5.6.3 Other debugging interfaces and tools 269
5.6.4 The Streams Debugger 269

Chapter 6. Advanced Streams programming 277
6.1 Developing native functions 278
6.1.1 Signature and usage in a sample application 278
6.1.2 Writing a function model 279
6.1.3 Implementing the native function in C++ 281
6.1.4 Compiling and running the application 282
6.1.5 Additional capabilities 282
6.2 Overview of Primitive operators 284
6.2.1 Generic Primitive operators 285
6.2.2 Non-generic Primitive operators 285
6.3 Developing non-generic C++ Primitive operators 286
6.3.1 Writing a Source operator 286
6.3.2 Writing a non-Source operator 294
6.3.3 Additional capabilities 301
6.4 Generic C++ Primitive operators 304
6.4.1 Writing a generic operator 305

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000007

6.4.2 Additional capabilities 312

Chapter 7. Streams integration approaches 317
7.1 Streams integration with IBM BigInsights 318
7.1.1 Application scenarios 318
7.1.2 Large scale data ingest 319
7.1.3 Bootstrap and enrichment 320
7.1.4 Adaptive analytics model 321
7.1.5 Application development 322
7.1.6 Application interactions 323
7.1.7 Enabling components 324
7.1.8 BigInsights summary 325
7.2 Streams integration with data mining 325
7.2.1 Value of integration 325
7.2.2 Sample scenario 327
7.2.3 Building the models 328
7.2.4 The contract: Data Analyst and Streams Component Developer . . 333
7.2.5 Preparing to write the operator 335
7.2.6 The Streams Component Developer 336
7.2.7 The Streams Application Developer 346
7.3 Streams integration with data stores 355
7.3.1 Concepts 355
7.3.2 Configuration files and connection documents 356
7.3.3 Database specifics 359
7.3.4 Examples 365

Appendix A. InfoSphere Streams installation and configuration 369
Installation considerations and requirements for Streams 370
Installing Streams on a single host 371
Installing Streams on multiple hosts 392

Appendix B. Toolkits and samples 397
Overview 398
What is a toolkit or sample 398
Why do we provide toolkits or samples 398
Where the toolkits and samples can be found 399
Other publicly available assets 401
Database Toolkit 401
Streams and databases 401
Database Toolkit V2.0 402
What is included in the Database Toolkit 403
Mining Toolkit 404
Streams and data mining 404
Mining Toolkit V2.0 405

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000008

How the Mining Toolkit works 405
Financial Toolkit 406
Why use Streams for financial markets 406
Financial Markets Toolkit V2.0 407
What is included in the Financial Toolkit 408
Internet Toolkit 410
Streams and Internet sources 410
Internet Toolkit V2.0 410
What is included in the Internet Toolkit 411

Appendix C. Additional material 413
Locating the web material 413
Using the web material 414
Downloading and extracting the web material 414

Glossary 415

Abbreviations and acronyms 419

Related publications 421
IBM Redbooks 421
Other publications 421
Online resources 421
IBM education support 422
IBM training 422
Information Management Software Services 423
IBM Software Accelerated Value Program 424
How to get Redbooks 425
Help from IBM 426

Index 427

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000009

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000010

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:
IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785 U.S.A.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000011

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both. These and other IBM trademarked terms are marked on their first occurrence in this information with the appropriate symbol (® or ™), indicating US registered or common law trademarks owned by IBM at the time this information was published. Such trademarks may also be registered or common law trademarks in other countries. A current list of IBM trademarks is available on the Web at http://www.ibm.com/legal/copytrade.shtml

The following terms are trademarks of the International Business Machines Corporation in the United States, other countries, or both:

Ascential®	Informix®	Smarter Cities™
DataStage®	InfoSphere™	Smarter Planet™
DB2®	Passport Advantage®	solidDB®
developerWorks®	RAA®	Tivoli®
GPFS™	Redbooks®	WebSphere®
IBM®	Redbooks (logo) ®	

The following terms are trademarks of other companies:

Netezza, and N logo are trademarks or registered trademarks of Netezza Corporation, an IBM Company.

Microsoft, Windows, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Snapshot, NOW, and the NetApp logo are trademarks or registered trademarks of NetApp, Inc. in the U.S. and other countries.

Java, and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000012

Preface

An information revolution has been underway now for more than 40 years. A key event in that revolution was the development of the relational database, and it changed the world. Now, another technology innovation has started a similar change. IBM® InfoSphere™ Streams is radically transforming the information industry by assembling continuous insights from data in motion, that is, from data that might not have yet been stored. This new technology has been developed to deal with the issues of high volume and velocity data, which we call *Big Data*.

This new approach, coupled with relational databases, enables higher expectations regarding the kinds of applications that can be deployed in the enterprise. In much the same way that technology, such as the assembly line, was the culmination of innovation in the Industrial Revolution, InfoSphere Streams is becoming the “assembler of continuous insight” of the Information Revolution. InfoSphere Streams captures and uses data in motion, or streaming data, by processing it as it passes through enterprise applications. With traditional data processing, you typically run queries against relatively static sources of data, which then provides you with a query result set for your analysis. With stream computing, you execute a process that can be thought of as a continuous query. The results are continuously updated as the sources of data are refreshed, or where extended continuous streams of data flow to the application and are continuously analyzed by static queries that *assemble* the results.

Based on the requirements, the assembled results may be stored for further processing or they may be acted on immediately and not stored. As a result, operations can be scaled out to more servers or cores because each core can focus on its own operation in the overall sequence. Like an assembly line, steps that take longer can be further segmented into sub-steps and processed in parallel. Because the data is processed in an *assembly line* fashion, high throughputs can be achieved, and because the data is processed as it arrives, initial results are available much sooner and the amount of inventory, or enrichment data, that needs to be kept on hand is decreased.

IBM has developed IBM InfoSphere Streams to address the emerging requirement for more timely results coupled with greater volumes and varieties of data. The IBM InfoSphere Streams product is the assembly line for insights of the Information Revolution, complementing the relational databases that began the revolution. What is new in Streams is the higher level of abstraction and the ability to make assembly lines available at a distributed cluster level.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000013

InfoSphere Streams also includes many elemental building blocks, such as text analytics, data mining, and relational analytics, to make it easier for enterprises to build these assembly lines for business insight. We believe that this approach will become widely used, enable us to meet the coming business challenges, and, as with the manufacturing assembly line, will transform the information industry.

InfoSphere Streams provides an execution platform and services for user-developed queries and applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous streams of data. These queries can be changed over time, without interrupting processing, based on intermediate results or application requirements. Developing applications in this type of environment is significantly different than traditional approaches because of the continuous flow nature of the data.

In this IBM Redbooks® publication, we describe the environment for this type of processing, provide an overview of the architecture of the InfoSphere Streams offering, and discuss how best to develop programming applications to operate in this type of environment. This book is intended for professionals that require an understanding of how to process high volumes of streaming data or need information about how to implement systems to satisfy those requirements.

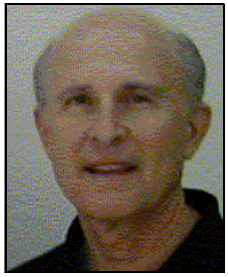
Imagine the business impact of being able to analyze more data than before and faster than before. What if it suddenly became cost effective to analyze data that was too time sensitive or expensive to analyze before? You could more consistently identify new market opportunities before competitors, and be agile enough more immediately respond to changes in the market.

There is tremendous potential in this type of processing. Begin the evolution in your environment today and start to gain the business advantage.

The team who wrote this book

This book was produced by a team of specialists from around the world working with the International Technical Support Organization, in San Jose, California.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000014



Chuck Ballard is a Project Manager at the International Technical Support organization, in San Jose, California. He has over 35 years experience, holding positions in the areas of Product Engineering, Sales, Marketing, Technical Support, and Management. His expertise is in the areas of database, data management, data warehousing, business intelligence, and process re-engineering. He has written extensively on these subjects, taught classes, and presented at conferences and seminars worldwide. Chuck has both a Bachelor's degree and a Master's degree in Industrial Engineering from Purdue University.



Kevin Foster is a Big Data Solutions Architect at IBM Silicon Valley Lab. He has worked in the database software area for nearly 30 years. In the first half of his career, Kevin worked as a system integrator building custom applications for the United States Government, Boeing, Xerox, Sprint, American Electric Power, and many others. In 1996, he began the next phase of his career working in a variety of roles in software product development, engineering management, technical marketing, and consulting. His current focus is on partner and customer technical enablement for the IBM InfoSphere Streams product. Kevin has a Bachelor's degree in Mathematics from California State University Stanislaus and a Master's degree in Computer Science from Stanford University.



Andy Frenkiel is a researcher and developer at IBM Research in Yorktown Heights, NY. Since starting with IBM in 1989, he has worked on development environments, communication and performance management systems. As a developer on the IBM InfoSphere Streams product, he served as a team lead for the Streams Studio IDE and is currently working on research topics related to design patterns for stream applications. Andy has a Bachelor's degree in Political Science from Colby College, and a Master's degree in Computer Science from Columbia University.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000015



Bugra Gedik is currently a research staff member at the IBM T. J. Watson Research Center. His research interests are in distributed data management systems, with a particular focus on runtime and compiler design for stream processing. Bugra has served as the chief architect for the IBM InfoSphere Streams product, and is the co-inventor of the SPL and the SPADE stream processing languages. He is the recipient of the IEEE ICDCS 2003, IEEE DSN 2011, and ACM DEBS 2011 best paper awards. Bugra has served as the PC chair for the ACM DEBS 2009 and IEEE CollaborateCom 2007 conferences. He has published over 50 peer-reviewed articles in the areas of distributed computing, data management, and programming languages, and is an IBM master inventor. Bugra received his Ph.D. from Georgia Institute of Technology and his Bachelor of Science degree from Bilkent University.



Michael P. Koranda is a Senior Technical Staff Member with IBM Software Group and has been working at IBM for over 30 years. He has been working on the development of the InfoSphere Streams product for the last 6 years and is currently the product Release Manager.



Senthil Nathan is a Senior Technical Staff Member at IBM Watson Research Center, in Hawthorne, New York. He has 26 years of experience in building software for different kinds of enterprise applications. Senthil has contributed to many application areas, including SOA, web services, PHP, Web 2.0, and Stream Computing. He has written useful technical articles targeted towards developers and has also taught classes on those subjects. Senthil has a degree in Electronics Engineering from Madurai University.



Deepak Rajan is a Research Staff Member at IBM T. J. Watson Research Center. His expertise is in the area of optimization, particularly applied to scheduling and graph problems that arise in distributed computing. He is also adept at software development in C++ / Java, and has worked in IBM software teams, implementing his algorithms in product code. Deepak received his Bachelor of Technology degree from the Indian Institute of Technology at Madras and Master of Science and Ph.D. degrees in Operations Research from the University of California at Berkeley.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000016



Roger Rea is the InfoSphere Streams Product Manager in IBM Software Group. Previously, he has held a variety of sales, technical, education, marketing and management jobs at IBM, Skill Dynamics, and Tivoli® Systems. He has received four IBM 100% Clubs, one Systems Engineering Symposium, and numerous Excellence Awards and Teamwork Awards. Roger earned a Bachelor of Science in Mathematics and Computer Science, Cum Laude, from the University of California at Los Angeles (UCLA), and a Master's Certificate in Project Management from George Washington University.



Mike Spicer is a Senior Technical Staff Member with IBM Software Group and the Chief Architect for InfoSphere Streams. He is an expert in high performance stream processing systems with nearly 20 years experience and deep contributions to multiple stream related projects. Mike has been an architect and leader on the InfoSphere Streams development team over the last 3 years focusing on the Streams run time. Before joining the Streams team, he was the architect for IBM DB2® Data Stream Edition, and a lead engineer for the IBM Informix® Real-Time Loader. Mike was also a technical lead for the first IBM WebSphere® Front Office for Financial Markets (WFO) product.



Brian Williams is an Executive IT Architect with IBM Software Services Federal (ISSF), in Bethesda, MD. He has over 27 years experience with expertise in enterprise information management, information security, and relational database design and development. Most recently, he has been designing, implementing, and managing stream processing applications using IBM InfoSphere Streams. In addition, he has developed and taught training courses, workshops, and conference labs for InfoSphere Streams. Brian has a Master's degree in Computer Science from Johns Hopkins University.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000017



Vitali N. Zoubov is a Principal Consultant with the IBM Software Group Lab Services. He joined IBM through the acquisition of Ascential® Software in 2005 and played the key roles in the successful implementations of a number of data warehousing and data integration solutions. His area of expertise include IBM Information Server with extensive knowledge of IBM DataStage® and IBM InfoSphere Streams. Vitali holds a Master's degree in Applied Mathematics from Lviv University and a Ph. D. degree in Physics and Mathematics from Kiev University.

We want to thank others who contributed to this IBM Redbooks publication, in the form of written content and subject expertise:

Dan Farrell
Client Technical Specialist, IBM Software Group, Worldwide Sales Enablement, Denver, CO.

James Giles
Senior Manager, InfoSphere Streams Development, IBM Software Group, Information Management, Somers, NY.

John Santosuosso
Application Consultant, IBM Systems &Technology Group, Systems Agenda Delivery, Rochester, MN.

Sandra L. Tucker
Executive IT Specialist, Information Management Lab Services Center of Excellence, Camden, DE.

Laurie A. Williams
Information Management Software, InfoSphere Streams Processing, Rochester, MN.

Chitra Venkatramani
Research Staff Member, InfoSphere Streams Development Research, IBM T. J. Watson Research Center, Hawthorne, NY.

We also want to thank the following people for providing help in the development of this book as reviewers, advisors, and project support:

Michael B. Accola, Rochester, MN
Michael J. Branson, Rochester, MN
Paul Bye, Rochester, MN
Diego Folatelli, San Jose, CA
Shrinivas S. Kulkarni, Bangalore, India

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000018

Mark Lee, Farnborough, United Kingdom
Michael A. Nobles, Atlanta, GA
Elizabeth Peterson, Seattle, WA
Warren Pettit, Nashville, TN
Paul D Stone, Hursley, United Kingdom
Robert M. Uleman, San Francisco, CA
IBM

Mary Comianos, Publications Management
Ann Lund, Residency Administration
Emma Jacobs, Graphics
Wade Wallace, Editor
International Technical Support Organization

Now you can become a published author, too!

Here's an opportunity to spotlight your skills, grow your career, and become a published author—all at the same time! Join an ITSO residency project and help write a book in your area of expertise, while honing your experience using leading-edge technologies. Your efforts will help to increase product acceptance and customer satisfaction, as you expand your network of technical contacts and relationships. Residencies run from two to six weeks in length, and you can participate either in person or as a remote resident working from your home base.

Find out more about the residency program, browse the residency index, and apply online at:

ibm.com/redbooks/residencies.html

Comments welcome

Your comments are important to us!

We want our books to be as helpful as possible. Send us your comments about this book or other IBM Redbooks publications in one of the following ways:

- Use the online **Contact us** review Redbooks form found at:

  ibm.com/redbooks

- Send your comments in an email to:

  redbooks@us.ibm.com

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000019

- Mail your comments to:

  IBM Corporation, International Technical Support Organization
  Dept. HYTD Mail Station P099
  2455 South Road
  Poughkeepsie, NY 12601-5400

Stay connected to IBM Redbooks

- Find us on Facebook:

  http://www.facebook.com/IBMRedbooks

- Follow us on Twitter:

  http://twitter.com/ibmredbooks

- Look for us on LinkedIn:

  http://www.linkedin.com/groups?home=&gid=2130806

- Explore new Redbooks publications, residencies, and workshops with the IBM Redbooks weekly newsletter:

  https://www.redbooks.ibm.com/Redbooks.nsf/subscribe?OpenForm

- Stay current on recent Redbooks publications with RSS Feeds:

  http://www.redbooks.ibm.com/rss.html

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000020

1

Introduction

In this chapter, we give you some background about an exciting analytic paradigm called stream computing. We introduce you to the concepts of streaming data and stream computing from the business landscape and informational environment perspectives, and the emergence of real-time analytic processing (RTAP). We then build on this fundamental information to show how IBM has been actively involved in this extreme shift in the process of decision making. The result of years of research into how to meet this challenge is IBM InfoSphere Streams (Streams). To help position this shift, we give you some history about the birth of Streams and how it is designed to provide an optimal platform for developing and deploying key applications to use streaming data in the decisions you need to make and the actions you need to take for the health of your business, and even the quality of your life.

Another of the objectives in this chapter is to provide insight into the following key questions and concepts:

- So what exactly is streaming data?
- What sort of things can be learned from it to improve the fabric of our lives and the functions of our businesses?
- How can developing streams-based applications enable you to become more proactive in the decision making process?
- How does IBM InfoSphere Streams enable you to harness the power of the plethora of information available from traditional and non-traditional sources of data that are moving all around us?

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000021

1.1 Stream computing

Bob Dylan once sang, “For the times they are a-changing” during the 1960s. Although those words could be true, no matter when they are spoken, they could also be considered an insightful understatement of every aspect of our present lives. We live in an age where the functions of communication that were once completely the purview of land lines and home computers are now primarily handled by intelligent cellular phones, where digital imaging has replaced bulky X-ray films, and the popularity of MP3 players and eBooks are changing the dynamics of something as familiar as the local library.

For years, self-professed information evangelists have taken every opportunity to expound on the explosive increase in the amount of available data and the challenge to glean key useful information from the deluge in time to take action. Actually, we have heard these proclamations so often we have become somewhat desensitized to the true impact of such a statement. At a time when this rate of increase is surpassing everyone's projections, many of us still feel the challenge of being able to do something with this tsunami of information to make it relevant to our jobs. Unfortunately, we rarely can. The flood of data is pertinent to all aspects of our lives. We cannot treat all that information as so much background noise any longer. It is time to be more than just aware that all that information is there. We need to integrate it into our businesses, our governments, our educational processes, and our lives. We need to use all of this information to make things happen, not just document and review what has happened.

This desire is not new. We have always wanted to use available information to help us make more informed decisions and take appropriate action in real time. The realization of such a lofty goal as truly real-time analysis has been challenged by many things over the years. There have been limitations in the capabilities of the IT landscape to support the delivery of large volumes of data for analysis in real time, such as processing power, available computing memory, network communication bandwidth, and storage performance. Though some of these constraints still exist, there have been significant strides in minimizing or alleviating some or all of them. One of the biggest challenges to presenting data in real time is not directly related to the capabilities of the data center. One of the main challenges has been the ability to acquire the actual information in a way that makes it available to the analysis process and tools in time to take appropriate action.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000022

Even today, typical analytical processes and tools are limited to using stored, and usually structured, data. To accommodate this requirement, data acquisition has historically required several time-consuming steps, such as collection through data entry or optical scanning, cleansing, transformation, enrichment and finally loading this data into an appropriate data store. The time it takes for these steps to complete results in a delay before the data is available to be analyzed and used to drive any actionable decisions. Often this delay is enough that any action taken from the analysis is more reactive than proactive in nature.

If there have been challenges to acquiring information in real time, we can only expect the situation to get worse. Our world is becoming more and more instrumented. Because of this phenomenon, traditionally unintelligent devices are now a source of intelligent information. Tiny processors, many with more processing power than the desktop computers of years ago, are being infused in the everyday objects of our lives. Everything from the packages of products you buy, to the appliances in our homes, to the cars we drive now have the capability to provide us with information we could use to make more informed decisions. An example of this situation is shown in Figure 1-1.



Figure 1-1 Information is available from formerly inanimate sources

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000023

Along with the proliferation of instrumentation in everyday products and processes, we are experiencing a surge in interconnectivity as well. Not only is the actual data available right from the sources, but those sources are also interconnected in such a way that we can acquire that data as it is being generated. The acquisition of data is no longer limited to the realm of passive observation and manual recording. The information produced and communicated by this massive wave of instrumentation and interconnectivity makes it possible to capture what is actually happening at the time it happens. Data can be acquired at the source and made available for analysis in the time it takes a machine to *talk* to another machine. Where we once assumed, estimated, and predicted, we now have the ability to know.

These advances in availability and functionality are the driving force in the unprecedented, veritable explosion in the amount of data available for analysis. Are we really ready to use this information? Even with the recent improvements in Information technology, the predicted steady increase in the amount of traditional stored data available was a considerable challenge to realizing true real-time analytics. The world is now able to generate inconceivable amounts of data. This unexpected increase in volume is likely to still outstrip the potential gains we have made in technology, unless we begin to change the way we approach analysis. Just to get that amount of data stored for analysis could prove to be an insurmountable task.

Imagine: In just the next few years, IP traffic is expected to total more than half a zettabyte. (That is a trillion gigabytes, or a one followed by 21 zeroes!).

A further challenge is new sources of data generation. The nature of information today is different than information in the past. Because of the profusion of sensors, microphones, cameras, and medical and image scanners in our lives, the data generated from these instruments is the largest segment of all information available, and about 80% of this information is unstructured. One advantage of this shift is that these non-traditional data sources are often available while they are enroute from their source to their final destination for storage, which means that they are basically available the moment they happen (before they are stored).

Let us consider data feeds from local or global news agencies or stock markets. Throughout their active hours of operation, these data feeds are updated and communicated as events happen. Most of us have at least one of the addresses (URLs) for these live data feeds bookmarked on our browser. We are confident that every time we access this address we will be able to review current data. Conversely, we probably do not have any idea where the historic information goes to finally be stored. More importantly, there may be no need to know where it is stored because we are able to analyze the data before it comes to rest.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000024

This data that is continuously flowing across interconnected communication channels is considered streaming data or *data in motion*.

To be able to automate and incorporate streaming data into our decision-making process, we need to use a new paradigm in programming called *stream computing*. Stream computing is the response to the shift in paradigm to harness the awesome potential of data in motion. In traditional computing, we access relatively static information to answer our evolving and dynamic analytic questions. With stream computing, you can deploy a static application that will continuously apply that analysis to an ever changing stream of data. As you look at Figure 1-2, think of the difference between doing research for a report on a current event from the printed material in a library verses the online news feeds.



Figure 1-2 Comparing traditional and stream computing

To help you better understand this concept, consider for the moment that you are standing on the bank of a rapidly flowing river into which you are planning to launch a canoe. You have a wetsuit but are not sure if you need to bother with wearing it. You first need to determine how cold the water is to decide whether or not you should put on the wetsuit before getting into the canoe. There is certainly no chance of capturing all of the water and holding it while you determine the average temperature of the full *set* of the water. In addition, you want to leave your canoe while you go to retrieve a newspaper that might have a listing of the local water temperatures. Neither of those options are practical. A simpler option is to stand on the edge of the creek and put your hand or foot into the water to decide whether it was warm enough to forego the wetsuit. In fact, you could also insert a thermometer for a more accurate reading, but that level of accuracy is probably not necessary for this scenario. The water's journey was not interrupted by our need to analyze the information; on the contrary, someone else could be testing those same waters farther downstream for their own informational needs. Much like its watery counterpart in this example, streaming data can be accessed from its edges during its travels without deterring further use downstream or stopping its flow to its final repository.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000025

Looking at the different programming approaches to analysis, you can see correlations to the river, although historically you have been able to capture all of the information prior to analysis. In traditional computing, you could produce a report, dashboard, or graph showing information from a data warehouse for all sales for the eastern region by store. You could take notice of particular items of interest or concern in your high level analysis and drill down into the underlying details to understand the situation more clearly. The data warehouse supplying information to these reports and queries is probably being loaded on a set schedule, for example, weekly, daily, or even hourly, and not being modified much after the data is committed. So the data is growing over time, but historical data is mostly static. If you run reports and queries today, tomorrow, or a month from now on the same parameters, the results will basically be the same. Conversely, the analysis in this example is evolving and changing from a summary to details. You could, and surely would, take action based on this analysis. For example, you could use it to project replenishment and merchandising needs, but because the analyst must wait until the data is loaded to use, their ability to take those actions in real time is limited. With some analyses and actions, this is a perfectly acceptable model, for example, making plans to update the production process or design a new line based on the success of another line. With stream computing, you can focus on an analysis that results in immediate action. You could deploy an analytic application that will continuously check the ever-changing inventory information that is based on the feeds from the cash registers and the inventory control system and send an alert to the stock room to replenish item(s) more quickly before loading the data.

1.1.1 Business landscape

As previously stated, the potential for data acquisition for analysis is taking place in real time than ever before. Still, most of our decisions, either as business leaders or even as individuals, are made based on information that is historic in nature or limited in scope. Even though most decision makers believe that more current data leads to better decisions, stored data has been the only source of data readily available for analysis. This situation has driven expectations about how the analytic process and the tools that support analysis should work. Businesses historically have to wait until after the actual data has been committed to storage before they can run any analysis. This analysis is then limited to using historical, and usually structured, data to predict the best actions for the future. By being able to acquire actual data in real time, businesses hope to be able to analyze and take action in real time, but they need new products and tools designed to do so.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000026

Figure 1-3 shows the results of a 2011 IBM study of 3,018 CIOs worldwide. The source of this data is the IBM Institute for Business Value Global CIO Study 2011. The survey found that there are four CIO mandates: Leverage, Expand, Transform, and Pioneer. Nearly one-quarter of the CIOs IBM had a conversation with support organizations that operate with a Transform mandate. Organizations with this mandate see IT primarily as a provider of industry-specific solutions to change the business.

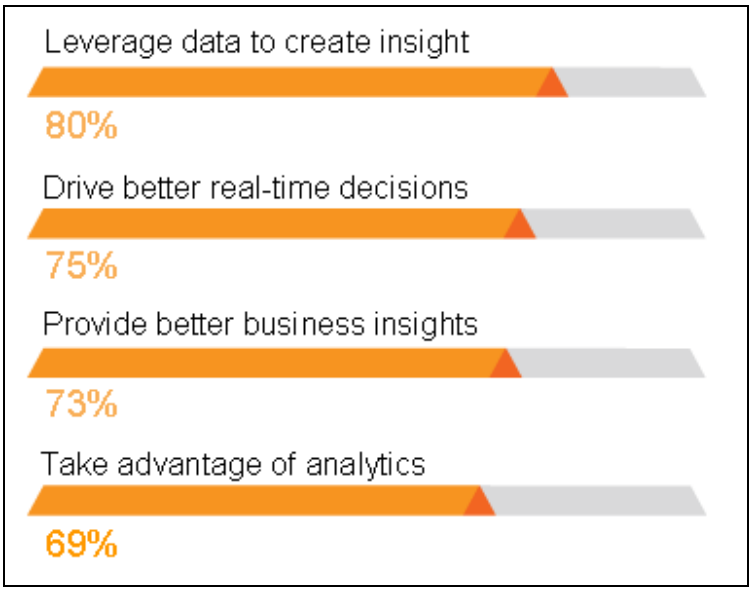


Figure 1-3 Enabling the intelligence enterprise

Beyond the delivery of basic IT services and business process improvements, Transform mandate CIOs are helping their public and private sector organizations fundamentally rethink the way they understand and interact with customers and partners. One key to extending the enterprise's reach is the use of *Big Data*, that is, the vast volumes captured and analyzed, such as data from sensors, RFID tags or real-time, web-based transactions that can be analyzed to drive better, real-time decisions. Big Data, what to do with it, and how to use it are top issues for Transform mandate CIOs.

Why is it that businesses continue to feel an urgency to seek an effective use of analytics? Because before you can determine an effective use strategy, the definition of what it takes to be effective changes. The rapid shift in the complexion of the data causes companies, governments, and even individuals to continually find themselves trying to determine how best to use the new data landscape.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000027

An excess of good data may sound like a good problem to have, but it means that with enterprise data projected to double every 18 months, it will be hard to keep up. And, with 80% of data growth driven by unstructured and non-traditional data sources, such as email, text, VoIP, and video, it is difficult to even know for which types of data to plan. With the fast pace of today's business environment, industry leaders are constantly being called upon to make innovative decisions in real time to gain a competitive edge, or to simply remain competitive.

The time is certainly near (if not already now) when these images and streaming data will be far more prevalent than their historic structured counterparts. Businesses are challenged by not only the volume and velocity of available data, but also by the ability to interpret the broad range of sources.

Note: A city the size of London could have tens of thousands of security cameras. Roads are often equipped with thousands of sensors for an area as limited as one bridge. Medical diagnostic equipment that creates digitized medical images make up, or will soon make up, almost a third of all the data in the world. Most if not all of these devices are connected to the World Wide Web, where they continually produce data.

The list of sources of available data for analysis just keeps growing. The increased reach of instrumentation brings with it the availability of a volume of data that businesses could never have dreamed would exist. We have smart appliances that can keep smart meters aware of energy usage in small increments for all appliances, which allows the utility company to make capacity adjustments at any level, even within a single home. Even something as basic as a tractor can capture and transmit soil condition, temperature, and water content and directly relay that information to a specific location to be used to monitor and manage water usage for irrigation systems, while simultaneously alerting the local dealership of the fact that the machine needs service. From the potential of smart grids, smart rail, smart sewers, and smart buildings, we can see significant shifts in the way decisions are made in the fabric of our lives.

Simultaneously, as individuals, we are becoming increasingly aware of the value of real-time information, whether to just feel more connected in a society where being in the same physical locality is not always possible between family members and friends, or as a forum to have our opinions heard. The rise in popularity of the social networks indicates that there is also a considerable personal value to real-time information. In this forum, millions of people (and therefore customers, students, patients, and citizens) are voicing their opinion about everything from good and bad experiences they have had with products or companies to their support for current issues and trends.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000028

Businesses that can analyze what is being said and align themselves with the popular desires will be a powerful force.

In some situations, businesses have tried to build applications to provide functionality beyond traditional analytics tools to use non-traditional or high volume information. Unfortunately, many of these businesses find that their in-house built applications struggle to scale well enough to keep up with the rapidly growing data throughput rates. The reality is that even as the old obstacles to real-time analytics are removed, business leaders are still finding themselves limited in their ability to make truly, real-time decisions. Unless they embrace a new analytical paradigm, the dynamic growth in data volumes will result in increasing the time needed to process data, potentially leading to missed opportunities and lowering, or losing, competitive advantage.

Businesses are keenly aware that knowing how to effectively use all known sources of data in a way that allows future sources to be incorporated smoothly can be the deciding factor that serves to fuel competitive, economic, and environmental advantages in real time. Those companies and individuals that find themselves positioned to be able to use data in motion will certainly find themselves with a clear competitive edge. The time it takes to commit the data to persistent storage can be a critical limitation in highly competitive marketplaces. More and more, we see that the effective key decisions need to come from the insights available from both traditional and non-traditional sources.

1.1.2 Information environment

You can already see that the changes in the business landscape are beginning to blur the lines between the information needs of business and the needs of the individual. Data that was once only generated, collected, analyzed, enriched, and stored in corporate data centers is now transmitted by some of the most innocuous devices. Data that was once only attainable by using the power provided by a business's information environment can now be accessed by devices as personal as intelligent cellular phones and personal computer tablets. With greater availability of data comes the potential for a new breed of analyst to enrich the information content. Thanks to the increases in instrumentation and interconnectivity, these new analysts and their new content can now be heard worldwide in mere seconds. The lines that historically have delineated the information environment into its use by industries, business, government, or individuals are becoming all but invisible. Not only is the world becoming more instrumented and interconnected, but the entire planet is becoming more intelligent, or at least has that potential.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000029

We are in the dawn of the IBM Smarter Planet™ era in information evolution. As you can see in Figure 1-4, the Smarter Planet era builds from earlier styles of computing, and is being driven by the rapid rise in the level of sophisticated computing technology being delivered into hands of the real world.

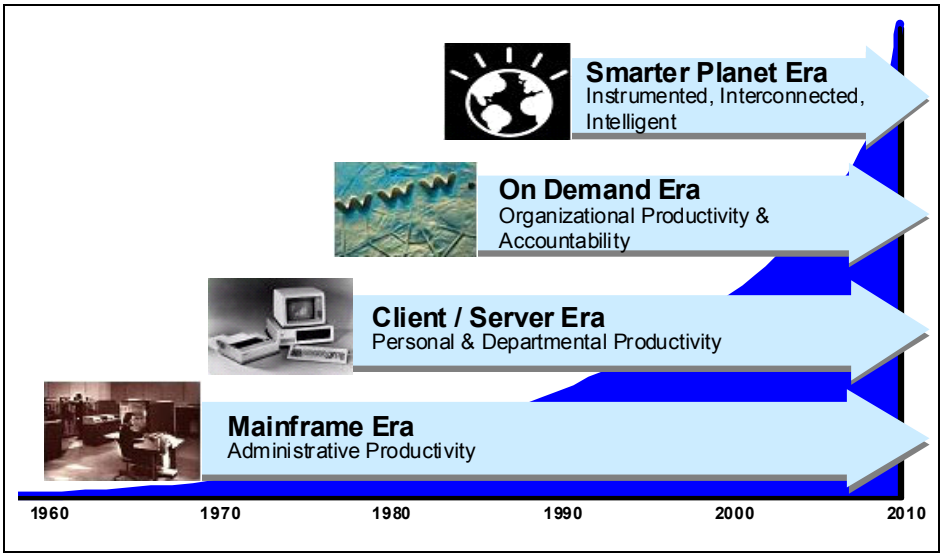


Figure 1-4 Eras of information technology evolution

The need for an automated information environment began in the mid 1960s with the Mainframe Era. A wonderful environment for automating the back office functions, it is still widely in use today. Because all businesses have back offices, the information environment of the Mainframe Era became prolific in all businesses and all industries. Not designed to do things such as plant floor manufacturing, departmental computing, or personal computing, this environment and the data it creates and stores are solely designed for and used by the specific enterprise. The purposes of the applications in this environment are to conduct the day-to-day business. The data it generates is only truly used by the processes of those day-to-day operations, such as billing, inventory control, accounting, and sales analysis.

The awareness of the value of analysis came from the departmental units within the business. The information environment of the Mainframe Era was well suited for creating and storing data, but not as well suited to analyzing that information to make improvements in the way to do business. As departmental decision makers were pressed to improve their bottom lines, they found themselves needing an information environment flexible enough to support any question they needed to ask.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000030

The Client/Server Era was the first information environment fashioned to support the business analyst. Unfortunately, as it rose from departmental demands, it was an information environment isolated from the data needs of the particular departmental unit. Still, it was quick and effective, and like its predecessor, it still exists in organizations today for targeted needs, such as campaign management.

But soon analysis could no longer be isolated to the departmental domain. The On Demand Era was born out of the need to blend (and validate) the departmental analysis to provide insight and recommendations at the enterprise level. The On Demand Era started integrating these islands of automation into an information environment that could serve the analytical needs of the entire enterprise. The rise of the worldwide Internet, with its open, standards-based, and widely available access, provided a framework to share departmental information within the business and the foundation for sharing information throughout the world at a pace we could barely imagine.

The growth of the environments to support information has been relatively steady over the years and gradually growing. Still, the sources of data and the results of analysis were primarily the property of industry, government, and academic science. These enterprise data centers controlled the information because only they had the computing power to acquire and analyze it. As such, they reviewed and verified information before communicating it to the rest of the world. But with computing power prevalent outside of the walls of industry, we must re-think what a computer really is and who might be performing analysis on the data.

Consider: There were 4.6 billion mobile phone subscribers in February 2010, and there will be 1.3 billion Radio Frequency Identification (RFID) tags produced globally in the next year. Sensors are being embedded across entire ecosystems, such as supply chains, health care networks, electric grids, cities, and even natural systems such as rivers. These sensors and phones can all be considered computers that have the ability to generate and communicate data.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000031

In the past, the information available on the Internet was only generated and communicated by those large data centers. Now it is estimated that not only will there likely be over 2 billion people connected to the Internet, but there will also be as many as a trillion objects connected as well. This Internet of things, show in Figure 1-5, shows some examples of how computers are moving out of the data centers and into our everyday world.

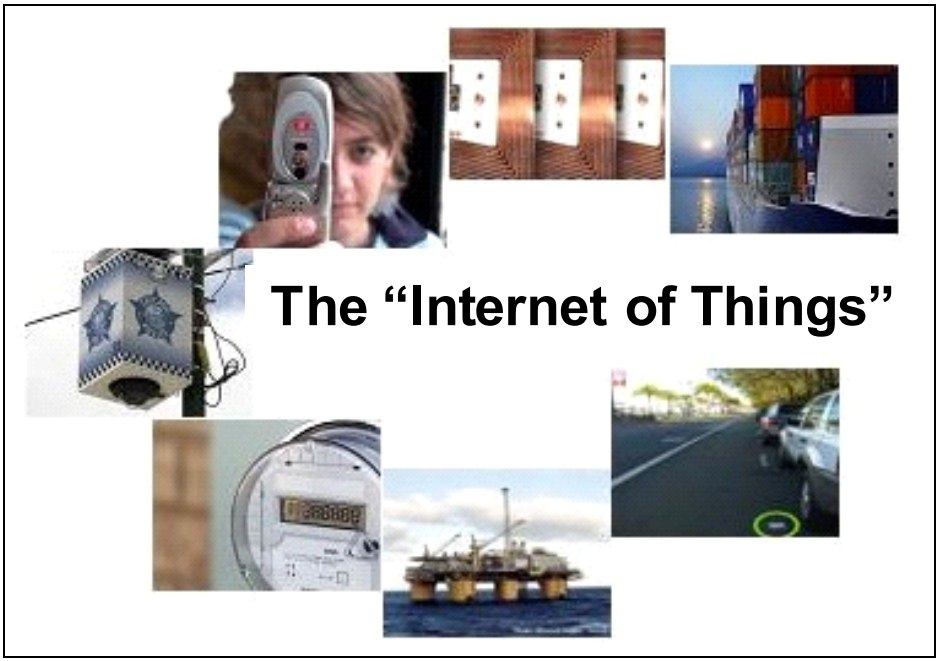


Figure 1-5 Computers are moving out of the data center into the world

The proliferation of these sensors and devices, embedded in everything from smart grids to rail cars, and even chickens, will help drive as much as a 10 times increase in the amount of data in the world. The ability to tap into that data for a new kind of intelligence can help drive vast improvements in our systems.

Welcome to the Smarter Planet Era. Driven by the surge in instrumentation, interconnectivity and intelligence being infused in all aspects of our lives, this new information environment holds the potential to infuse even complex analysis in the routine aspects of our lives. This new environment will enable situations such as having traffic and security cameras in major cities help alert police and other first responders to the logistics of an incident far faster and more precisely than ever before. The information from sensors in your car could alert your garage-of-choice that your car needs maintenance so they can schedule an appointment for you and prompt you to confirm the appointment through a text message on your cell phone.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000032

In-home monitors that transmit key health factors could have the capability that allows an elderly relative to continue living in their own home. Automated quality analysis installed on a factory line could identify defects in products before they leave the line to allow operators to take corrective measures.

While having access to data sooner may save lives, lower crime, save energy, and reduce the cost of doing business and therefore products, one of the by-products of the widespread instrumentation is the potential for feeling like we live in a *surveillance society*. Although it is easy for people and businesses to see the value of access to the wide variety of data and smarter systems, it is also easy to see how they may be increasingly uncomfortable having so much information known about them. Individuals and businesses alike find themselves concerned that this essential environment might only be as secure and reliable as the average person's mobile computer or PDA.

Note: You may have read an article a few years ago that reported that the London flat in which George Orwell wrote *1984* has 32 closed-circuit cameras within 200 yards, scanning every move. Those cameras were not put there specifically to spy on that flat; they were installed to scan traffic and provide security for local businesses. Still, they are there and as they are most likely connected to the internet, the irony, and the potential concern, is pretty easy to see.

Along with the sheer mechanics of analyzing the massive volumes of information in real time comes the additional challenge of how to provide security and privacy. The information environment of the Smarter Planet Era gets its reaction speed from accessing data in motion, data outside of the usual framework of data repository authorization and encryption features. The sheer volume of the available data in motion requires a high performance environment for stream computing, but this environment also needs to be able to employ in-stream analysis to determine the credibility of sources and protect the information without adversely compromising the speed. When the information environment uses and allows widespread access to analyze and enrich data in motion, it also takes on some of the responsibility to protect the results of that analysis and enrichment of the data and determine the analysis is based on credible and trusted sources.

The potential of the information environment for this new Smarter Planet Era is great, but so are its challenges. Providing an environment capable of delivering valuable and insightful information for real-time analytics without compromising on quality and security is the standard with which we will measure the success of this era.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000033

1.1.3 The evolution of analytics

Just as the business landscape and information environments are evolving, the approaches to effective analysis are at the dawn of a major evolution. We might think we are drowning in data, but in fact we now have the ability to turn that data into useful information. Advanced software analytic tools and sophisticated mathematical models can help us identify patterns, correlations of events, and outliers. With these new tools, we can begin to anticipate, forecast, predict, and make changes in our systems with more clarity and confidence than ever before. We stand on the brink of the next generation of intelligence: analysis of insightful and relevant information in real time, which is the real value of a Smarter Planet. The evolution of the analytic process is shown in Figure 1-6.

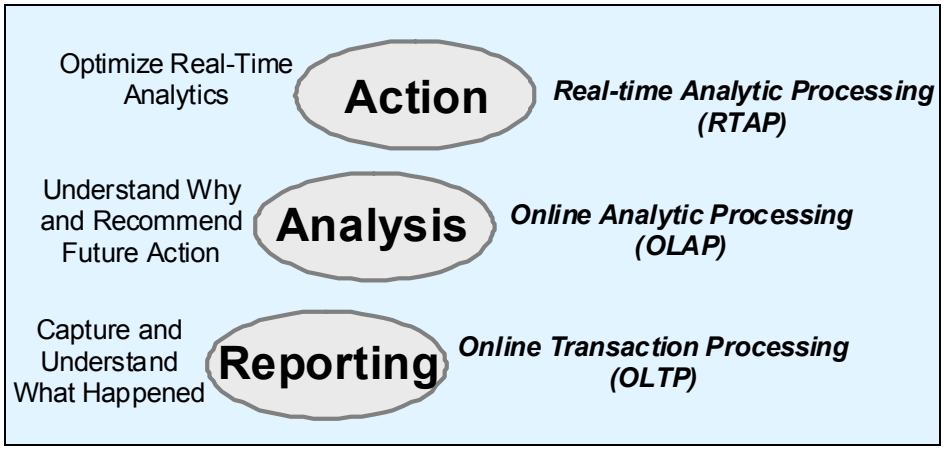


Figure 1-6 The next generation of business intelligence

Hierarchical databases were invented in the 1960s and still serve as the foundation for online transaction processing (OLTP) systems for all forms of business and government driving trillions of transactions today. Consider a bank as an example. It is quite likely that even today in many banks that information is entered into an OLTP system, possibly by employees or by a web application that captures and stores that data in hierarchical databases. This information then appears in daily reports and graphical dashboards to demonstrate the current state of the business and to enable and support appropriate actions. Analytical processing here is limited to capturing and understanding what has happened.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000034

Relational databases brought with them the concept of data warehousing, which extended the use of databases from OLTP to online analytic processing (OLAP). Using our example of the bank, the transactions captured by the OLTP system were stored over time and made available to the various business analysts in the organization. With OLAP, the analysts could now use the stored data to determine trends in loan defaults, overdrawn accounts, income growth, and so on. By combining and enriching the data with the results of their analysis, they could do even more complex analysis to forecast future economic trends or make recommendations on new investment areas. Additionally, they could mine the data looking for patterns to help them be more proactive in predicting potential future problems in areas such as foreclosures. The business could then analyze the recommendations to decide if they should take action. The core value of OLAP is focused on understanding why things happened to make more informed recommendations.

A key component of both OLTP and OLAP is that the data is stored. Particularly now, some new applications require faster analytics than is possible when you have to wait until the data is retrieved from storage. To meet the needs of these new dynamic applications, you must take advantage of the increase in the availability of data prior to storage, otherwise known as streaming data. This need is driving the next evolution in analytic processing called real-time analytic processing (RTAP). RTAP focuses on taking the proven analytics established in OLAP to the next level. Data in motion and unstructured data may be able to provide actual data where OLAP had to settle for assumptions and hunches. The speed of RTAP allows for the potential of action in place of simply making recommendations.

So, what type of analysis makes sense to do in real time? Key types of RTAP include, but are not limited to, the following analyses:

- Alerting
  - The RTAP application notifies the user(s) that the analysis has identified that a situation (based on a set of rules or process models) has occurred and then optionally provides recommendations and options for the appropriate actions.
  - Alerts are useful in situations where the application should not be automatically modifying the process or automatically taking action. They are also effective in situations where the action to be taken is outside the scope of influence of the RTAP application.

  Some examples are:

  - A market surveillance application that is monitoring a local exchange for suspect activity would notify someone when such a situation occurs.
  - A patient monitoring application would alert a nurse to take a particular action, such as administering additional medicine.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000035

- Feedback
  - The RTAP application identifies that a situation (based on a set of rules or process models) has occurred and makes appropriate modifications to the processes to prevent further problems or to correct the problems that have already occurred.
  - Feedback analysis is useful, as an example, in manufacturing scenarios where the application has determined that defective items have been produced and takes action to modify components to prevent further defects.

  An example is a manufacturer of plastic containers may run an application that uses the data from sensors of the production line to check the quality of the items throughout the manufacturing cycle. If defective items are sensed, the application generates instructions to the blending devices, for example, to adjust the ingredients to prevent further defects from occurring.
- Detecting failures
  - The RTAP application is designed to notice when a data source does not respond or generate data within a prescribed period of time.
  - Failure detection is useful in determining system failure in remote locations or problems in communication networks.

  Some examples are:
  - An administrator for a critical communications network deploys an application to continuously test that the network is delivering an adequate response time. When the application determines that the speed drops below a certain level, or is not responding at all, it alerts the administrator, wherever they happen to be.
  - An in-home health monitoring application determines that the motion sensors for a particular subscriber have not been activated today and sends an alert to a caregiver to check on the patient to determine why they are not moving around.

When you look at the big picture, you can see that if you consider the improvements that have been made in chip capacity, network protocols, and input / output caching with the advances in instrumentation and interconnectivity and the potential of stream computing, we stand poised and ready to be able to present appropriate information to the decision makers in time for proactive action to be taken. The days of solely basing our decisions on static data are yielding to being able to also use streaming data or *data in motion*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000036

1.1.4 Relationship to Big Data

With the growing use of embedded chips in sensors and smart phones, the volume of data generated annually is exploding into the exabyte data size range. With the pervasive deployment of sensors to monitor everything from environmental processes to human interactions, the variety of digital data is rapidly encompassing structured, semi-structured, and unstructured data. Finally, with better and faster networks to carry the data, from wireless to fiber optic, the velocity of data is also exploding. We call data with these characteristics Big Data. Examples include sources such as the Internet, web logs, chats, sensor networks, social media, telecommunications call detail records, biological sensor signals (as examples, EKG and EEG), astronomy, images, audio, medical records, military surveillance, and eCommerce.

With the ability to generate all this valuable data from their systems, businesses and governments are grappling with the problem of analyzing the data for two important purposes: to be able to sense and respond to current events in a timely fashion, and to be able to use predictions from historical learning to guide the response. This situation requires the seamless functioning of data-in-motion (current data) and data-at-rest (historical data) analysis, operating on massive volumes, varieties, and velocities of data. Bringing the seamless processing of current and historical data into operation is a technology challenge faced by many businesses that have access to Big Data.

IBM has introduced a Big Data platform, based on two products: IBM InfoSphere Streams and IBM InfoSphere BigInsights, which are designed to address this class of problems. Both products are built to run on large-scale, distributed systems designed to scale from small to large data volumes, while handling both structured and unstructured data analysis. In this book, we describe various scenarios where data analysis can be performed across the two platforms to address the Big Data challenges.

1.2 IBM InfoSphere Streams

In April of 2009, IBM made available a revolutionary product named IBM InfoSphere Streams (Streams). Streams is a product architected specifically to help clients continuously analyze massive volumes of streaming data at extreme speeds to improve business insight and decision making. Based on ground-breaking work from an IBM Research team working with the U.S. Government, Streams is one of the first products designed specifically for the new business, informational, and analytical needs of the Smarter Planet Era.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000037

As seen in Figure 1-7, Streams is installed in over 150 sites across six continents. Hundreds of applications have been developed, helping IBM understand more about client requirements for this new type of analysis.

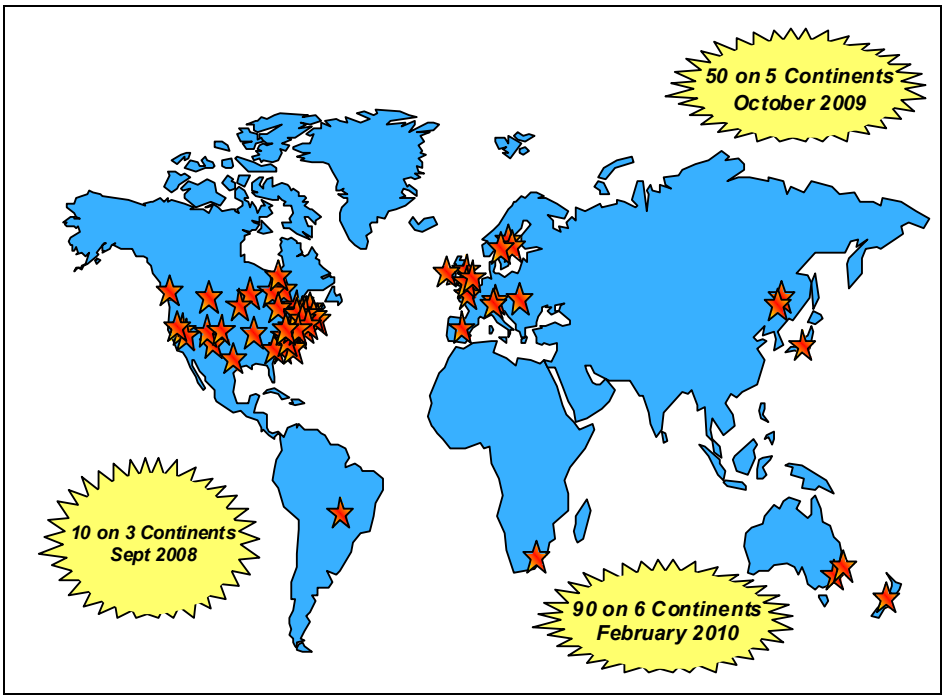


Figure 1-7 Streams installations as of February 2010

Streams is based on nearly a decade of effort by that IBM Research team to extend computing technology to handle advanced analysis of high volumes of data quickly. How important is their research? Consider how it would help crime investigation to be able to analyze the output of any video cameras in the area surrounding the scene of a crime to identify specific faces of any “persons of interest” in the crowd and relay that information to the unit that is responding. Similarly, what a competitive edge it could provide by being to be able to analyze 6 million stock market messages per second and execute trades with an average trade latency of only 25 microseconds (far faster than a hummingbird flaps his wings). Think about how much time, money, and resources could be saved by analyzing test results from chip-manufacturing wafer testers in real time to determine if there are defective chips before they leave the line.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000038

Note: While at IBM, Dr. Ted Codd invented the relational database. In the defining IBM Research project, it was referred to as System R, which stood for Relational. The relational database is the foundation for data warehousing that launched the highly successful Client/Server and On Demand informational eras. One of the cornerstones of that success was the capability of OLAP products that are still used in critical business processes today.

When the IBM Research division again set its sights of developing something to address the next evolution of analysis (RTAP) for the Smarter Planet evolution, they set their sights on developing a platform with the same level of world changing success, and decided to call their effort System S, which stood for Streams. Like System R, System S was founded on the promise of a revolutionary change to the analytic paradigm. The research of the Exploratory Stream Processing Systems team at T.J. Watson Research Center, which was set on advanced topics in highly-scalable stream-processing applications for the System S project, is the heart and soul of Streams.

Critical intelligence, informed actions, and operational efficiencies that are all available in real time is the promise of Streams. InfoSphere Streams will help us realize the promise of a Smarter Planet.

1.2.1 Overview of Streams

For the purposes of this overview, it is not necessary to understand the specifics, as they will be discussed in more detail in subsequent chapters. The purpose here is only to demonstrate how Streams was designed and architected to focus on the ability to deliver RTAP of exceedingly large volumes of data using a flexible platform positioned to grow with the increasing needs of this dynamic market.

As the amount of data available to enterprises and other organizations dramatically increases, more and more companies are looking to turn this data into actionable information and intelligence in real time. Addressing these requirements requires applications that are able to analyze potentially enormous volumes and varieties of continuous data streams to provide decision makers with critical information almost instantaneously. Streams provides a development platform and runtime environment where you can develop applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams based on defined, proven, and analytical rules that alert you to take appropriate action, all within an appropriate time frame for your organization.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000039

The Streams product goes further by allowing the applications to be modified dynamically. Although there are other systems that embrace the stream computing paradigm, Streams takes a fundamentally different approach to how it performs continuous processing and therefore differentiates itself from the rest with its distributed runtime platform, programming model, and tools for developing continuously processing applications. The data streams that are consumable by Streams can originate from sensors, cameras, news feeds, stock tickers, or a variety of other sources, including traditional databases. The streams of input sources are defined and can be numeric, text, or non-relational types of information, such as video, audio, sonar, or radar inputs. Analytic operators are specified to perform their actions on the streams. The applications, once created, are deployed to the Streams Runtime.

A simple view of the components of Streams is shown in Figure 1-8:

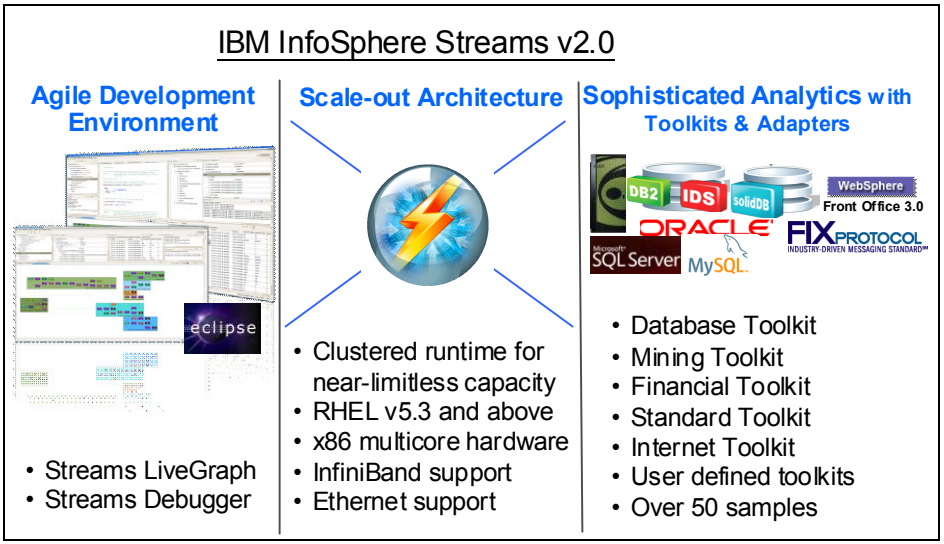


Figure 1-8 The components of InfoSphere Streams V2.0

Applications are developed in the InfoSphere Streams Studio using IBM Streams Processing Language (previously called Streams Processing Application Declarative Engine), which is a declarative language customized for stream computing. Once developed, the applications are deployed to Streams Runtime environment. Streams Live Graph then enables you to monitor performance of the runtime cluster, both from the perspective of individual machines and the communications between them.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000040

Virtually any device, sensor, or application system can be defined using the language. But there are also predefined source and output adapters that can further simplify application development. As examples, IBM delivers the following adapters:

- TCP/IP, UDP/IP, and files
- IBM WebSphere Front Office, which delivers stock feeds from major exchanges worldwide
- IBM solidDB® includes an in-memory, persistent database using the Solid Accelerator API
- Relational databases, which are supported using industry standard ODBC

Applications, such as the one shown in the application graph in Figure 1-9, almost always have multiple steps.

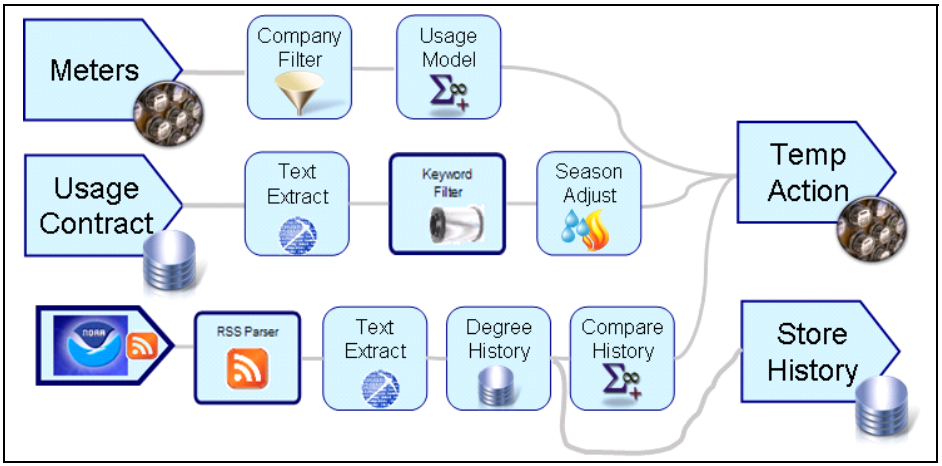


Figure 1-9 Application graph of a Streams application

For example, some utilities have begun paying customers who sign up for a particular usage plan to have their air conditioning units turned off for a short time, allowing the temperature to be changed An application to implement this would collect data from meters, and might apply a filter to only monitor for those customers who have selected this service. Then, a usage model must be applied that has been selected for that company. Up-to-date usage contracts then need to be applied by retrieving them, extracting text, filtering on key words, and the possibly applying a seasonal adjustment. Current weather information can be collected and parsed from the U.S. National Oceanic & Atmospheric Administration (NOAA), which has weather stations across the United States. After parsing for the correct location, text can be extracted and temperature history can be read from a database and compared to historical information.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000041

Optionally, the latest temperature history could be stored in a warehouse for future use. Finally, the three streams (meter information, usage contract, and current weather comparison to historical weather) can be used to take actions.

Streams delivers an Integrated Development Environment (IDE) based on the Open Source Eclipse project, as shown in Figure 1-10. Developers can specify operators to be used in processing the stream, such as filter, aggregate, merge, transform, or much more complex mathematical functions such as Fast Fourier Transforms. The developer also specifies the data input streams to be used (source operators) and data output streams (sink operators). Some of the source and sink adapters are included within the Streams product and the environment also allows the developer to define custom adapters, including custom analytics or operators written in C++ or Java. Existing analytics can also be called from Streams applications. More details about these functions can be found in the subsequent chapters of this book.

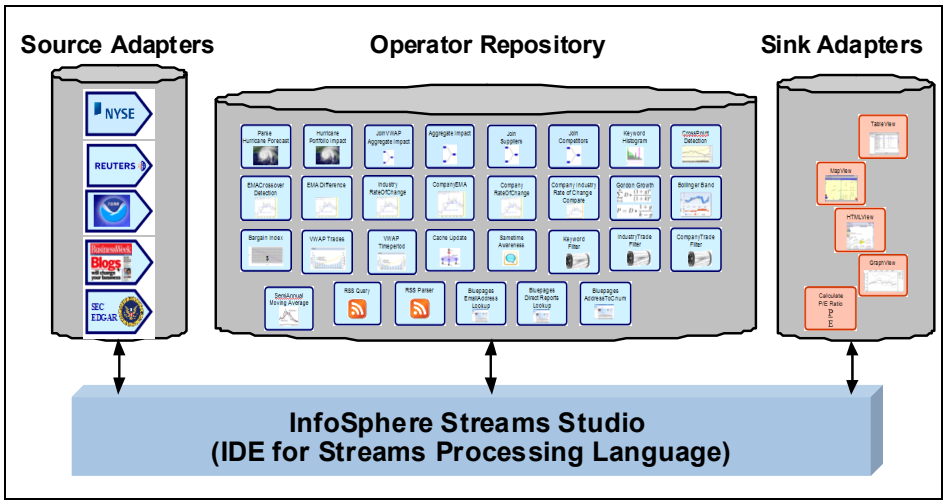


Figure 1-10 The Streams Integrated Development Environment (IDE)

InfoSphere Streams Studio not only helps you develop applications, but includes the powerful Streams Debugger tool. As with many IDEs, you can set breakpoints and stops in your application to facilitate testing and debugging. In addition, the Streams Debugger works with the Streams Runtime and the Streams Live Graph tool so that you can find a break point even after the application has been deployed. You can inspect or inject new data to ensure the application is doing what you want it to do.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000042

An optimizing compiler translates the application to executable code. The compiler makes many decision to optimize application performance, such as buffer sizes required, which functions should be run on a single machine, and when new execution threads can be initiated. When deployed, the Streams Runtime works with the optimizing compiler to allocate the application across the runtime configuration, similar to the example in Figure 1-11.

The Streams Runtime establishes communications for the streams of data as they are being processed by the application. This task is accomplished by setting up connections to route data into the Streams Runtime, between the various operators, and out of the Streams Runtime cluster. Because the Streams Runtime is all handled in memory across systems within a high speed infrastructure, the overall application experiences extremely low latency.

The Streams Runtime monitors the environment's performance and if it detects poor performance, or hardware failure, the administrator can take corrective actions. For example, the administrator can move an analytic function to another processor in the cluster. The Streams Runtime allows you to add or remove applications without bringing down the cluster. You can also add or remove computer nodes from the cluster while it is running.

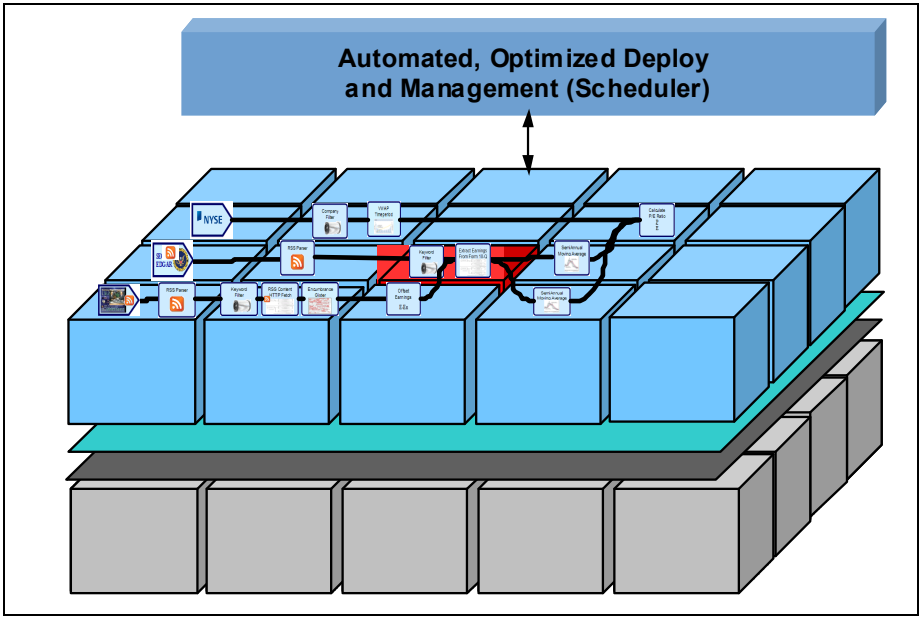


Figure 1-11 An illustration of a typical runtime deployment of a Streams application

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000043

1.2.2 Why use Streams

The architecture of Streams represents a significant change in the organization and capabilities of a computing platform. Typically, stream computing platforms are designed with the lofty goal of achieving success of the following objectives:

- Respond in real time to events and changing requirements
- Continuously analyze data at volumes and rates that are orders of magnitude greater than existing systems
- Adapt rapidly to changing data formats and types
- Manage high availability, heterogeneity, and distribution for the new stream paradigm
- Provide security and information confidentiality for shared information

Although there have been academic and commercial initiatives focused on delivering on the above technical challenges in isolation, Streams attempts to simultaneously address all of them. Streams is on a mission to break through a number of fundamental barriers to enable the creation of solutions designed to deliver on all of these critical objectives.

To do so, the Streams development environment provides a graphical representation of the composition of new applications to make them easier to understand. New applications, or new rules for analysis, can be created dynamically, mapped to a variety of hardware configurations, and then deployed from the development environment to provide the desired flexibility and agility of administration needed within this new paradigm. As analytical needs or rules change, or the priorities and value of data sources shift, the affected portions of the applications can be modified and re-deployed incrementally with minimal impact to the rest of the application. As hardware resource availability grows within the data center, Streams is designed to be able to scale from a single node to one or more high performance clusters having no limit on the number of processing nodes within a cluster. The runtime component continually monitors and adapts to the “state” and utilization levels of its computing resources and the data velocity.

There have been other products engineered to focus on some of the above objectives, many of which center around the analysis of events. You should think of an event as something that happens, such as a temperature reading, a purchase, or an insurance claim. Events might be simple events, such as a purchase transaction, or complex events, such as a combination of purchases with a specific time span.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000044

In comparison, a data stream is somewhat different. Streaming data is more of a continuous set of never ending readings, for example, the data from an EKG, the output of a video camera, or the sound captured by a microphone. Data streams can be viewed as a continuous stream of events happening quickly and as such there are often correlations and comparisons between stream processing and complex event processing.

IBM has been aware of the potential of using business event processing (processing of both simple and complex events) for quite some time. To get a better idea of exactly how to use this dynamic, IBM has classified seven major opportunities for business event analysis and determined the likely corresponding entry points to consider business event processing systems such as Streams, which are:

- Business Activity Monitoring: Communicating key performance indicators (KPIs) and alerts to business stakeholders
- Information-Derived Events: Identifying correlating events across data stores and data streams
- Business Logic Derived Events: Identifying correlating events across transactions and business processes
- Active Diagnostics: Identifying potential issues and taking action
- Predictive Processing: Identifying future issues based upon business events in context and over time
- Business Service Management: Ensuring IT health so service level agreements are met
- High Volume Stream Analytics: Analysis on high volume data streams in real time

Complex Event Processing (CEP) is an evolution of business event processing. It is primarily a concept that deals with the task of processing multiple business events with the goal of identifying the meaningful characteristics within the entire scenario. In addition, Business Event Processing (BEP) has been defined by IBM and handles both simple and complex events within a business context.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000045

The goal is always to be able to take advantage of an appropriate action plan in real time. Because of a similar goal within CEP, CEP tools often have the same, or similar, marketing messages as Streams. However, the proof points behind those messages are typically quite different. At the core, these goals of ultra-low latency and real-time event stream processing in real time are true for both. While CEP tools usually achieve response times of under a millisecond, Streams has achieved response times below one hundred microseconds. Both types of tools are able to use parallel processing platforms and techniques, such as partitioning and pipelining, to achieve performance, but the extent of that achievement is far from the same, when comparing a typical CEP tool to Streams.

As we just mentioned, Streams has some similarities to other CEP tools on the market, but it is built to support higher data volumes and a broader spectrum of the data sources used for input. Streams can successfully process millions of messages per second while other CEP tools typically only speak about processing hundreds of thousands of messages per second. Although CEP tools are able to handle discrete events, Streams can handle both discrete events and real-time data streams, such as video and audio. As we have previously described, Streams also provides an infrastructure to support the need for scalability and dynamic adaptability, by using scheduling, load balancing, and high availability techniques that will certainly be key as this information environment continues to evolve.

Like Streams, CEP tools and other products use techniques such as the detection of complex patterns within many events, event correlation and abstraction, event hierarchies, and relationships between events to identify events of events. These complex events allow for analysis across all the layers of information to determine their impact. CEP tools often employ if / then / else-type rules-based analysis. With Streams, you are able to incorporate powerful analytics such as signals processing, regressions, clustering algorithms, and more.

In Table 1-1, we have listed a few characteristics of Streams and CEP to help clarify, and differentiate, them.

Table 1-1 Characteristics comparison of Streams and CEP tools

Complex Event Processing	**InfoSphere Streams**
Analysis on discrete business events.	Analytics on continuous data streams.
Rules-based processing (using if / then / else) with correlation across event types.	Supports simple to extremely complex analytics and can scale for computational intensity.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000046

Complex Event Processing	InfoSphere Streams
Only structured data types are supported.	Supports an entire range of relational and non-relational data types.
Modest data rates.	Extreme data rates (often an order of magnitude faster).

As you can see, Streams is fully aligned to the challenges of the Smarter Planet Era. The optimized runtime compiler is architected to manage the extremely high volume of data that makes up the current challenges and will make up the future business landscape by providing the low latency required by real-time analytical processing. The development environment and toolkits are focused on using a client's existing analytics. In addition, the environment will easily allow for the creation of new and complex analytics to find credibility and value for multiple traditional and even non-traditional information sources that are becoming the key elements in our ever-broadening information environment.

1.2.3 Examples of Streams implementations

Even throughout its research and development phase, Streams has been demonstrating success by delivering cutting edge commercial and scientific applications. Many of these applications have shown clients a way to capture the formerly unattainable competitive edge. In addition, many of these early applications are playing a significant role in the evolution of the Smarter Planet.

From the beginning, the intent for the Streams infrastructure was to provide a solid foundation for this broad spectrum of practical applications by being designed specifically to address the fundamental challenges of analysis on streaming data. By being able to continually deliver exceptional performance (for all types of analyses across massive amounts of data) while remaining flexible enough to offer interoperability (with the existing application infrastructures), Streams is positioned for any solution that would benefit from analyzing streaming data in real time. The uses are as varied as the imagination of the analysts involved in almost every industry and initiative.

Some of the early adopters of Streams have developed solutions with life changing potential. To demonstrate some of the types of solutions you can create with the Streams platform, the following sections provide brief examples.

Over the past several years, hundreds of applications have been developed using InfoSphere Streams. The following sections provides a summary of a few applications and highlights the types of usage supported by InfoSphere Streams.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000047

Example 1: Telecommunications - Call Detail Record mediation and analytics

The challenge of closing certain technology and business gaps has been especially apparent for cellular service providers in Asia. Chips that are embedded in cell phones enable email, texting, pictures, videos, and information sharing using social sites such as Facebook. For each phone call, email, web browse, or text message, cellular phone switches emit call detail records (CDRs). To ensure no data is lost, the switches emit two CDRs for each transaction, which must later be deduplicated for the billing support systems. A rising volume of data caused mediation of CDRs to become ever more difficult to perform in a timely manner. Phone number portability enabled subscribers to move to a competitor at any time. Some sophisticated users even switch between multiple providers at different times within a given day to take advantage of certain promotions. Providers not only needed to reduce the window of processing CDRs to near real time, but also perform real-time analytics in parallel, to predict which customers might leave for a competitor, known in the industry as *churn*. With this real-time insight into customer behavior, providers could take action to retain a higher percentage of customers.

InfoSphere Streams, with its agile programming model, has enabled customers to handle their huge volume of CDRs with low latency, while providing churn analysis on the data at the same time. At one company, a peak rate of 208,000 CDRs per second is being processed with an end-to-end processing latency of under 1 second. Each CDR is checked against billions of existing CDRs with duplicates eliminated in real time, effectively cutting in half the amount of data stored in their databases.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000048

This example illustrates a key Streams use case: simultaneous processing, filtering, and analysis in real time. High Availability, automated fault tolerance and recovery, along with real-time dashboard summaries, are also currently in use and improving IT operations. Real-time analysis of CDRs is leading to improved business operations by performing things like churn prediction and campaign management to improve the retention rate of their current customers, as shown in Figure 1-12.

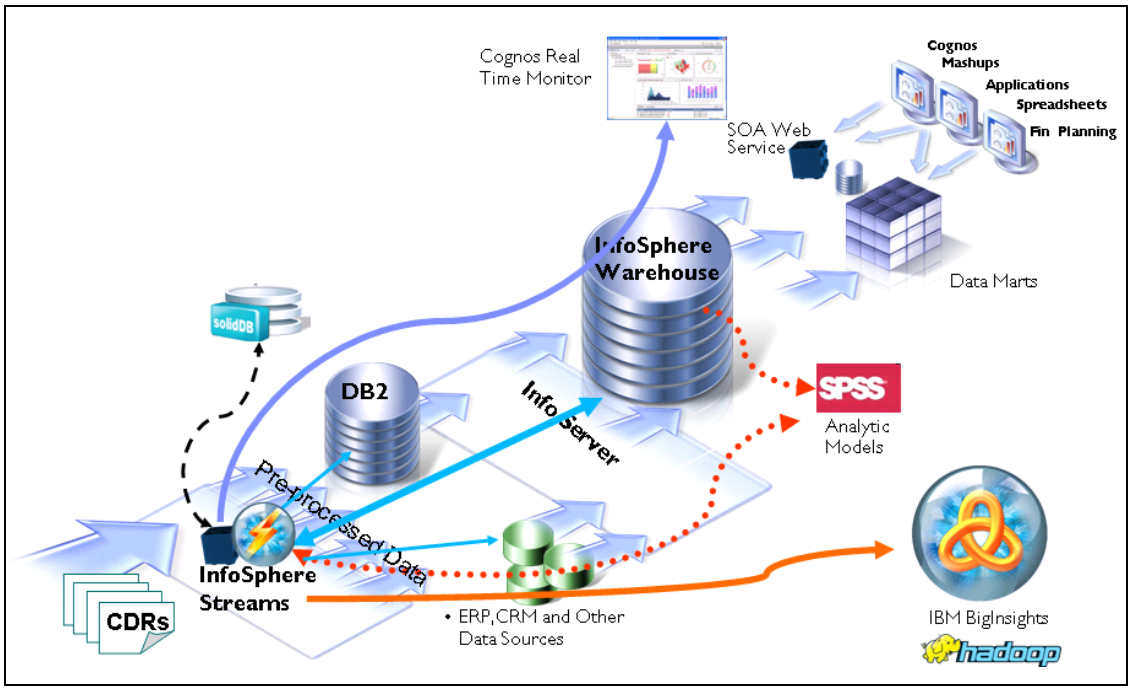


Figure 1-12 InfoSphere Streams Call Detail Record processing architecture

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000049

Example 2: Government - Maintaining cybersecurity in a hostile environment

A key strength of Streams is the ability for Streams to perform analytics on data-intensive streams to quickly and accurately identify the typically small number of items that merit deeper investigation. One example of this use case can be found in the domain of cyber security. A *botnet* is a network of software agents, or robots, that run autonomously and automatically. Botnets respond to Command and Control (C&C) machines, where an underground economy has sprung up to provide per pay access to the botnets for criminal activity. These botnets are evolving rapidly to make it difficult to detect the bots due to fast fluxing networks and encryption. The Shadowserver Foundation tracks known bots, and they estimate that there are thousands of C&C machines, and tens of thousands of bots currently in action today.

InfoSphere Streams was used to analyze IP traffic at a data rate of over 100 megabytes per second of IP traffic and over 10 million domain name server (DNS) queries per hour to generate fast fluxing botnet alerts. InfoSphere Streams not only uses machine learning models, but also models created using historic data in IBM InfoSphere Warehouse for botnet alerts. SPSS Modeler is used to create these historic models. InfoSphere Streams also monitors for model drift. When the attack patterns change, Streams will issue requests to have models updated against the historic data to ensure all parts of the solution are using up-to-date detection models. This use case shows the value of continuously updating the data mining models for both historic and real-time analysis of the data.

Example 3: Financial Services - Finding arbitrage opportunities faster than the competition

Many segments of the financial services industry rely on rapidly analyzing large volumes of data to make near-real time business and trading decisions. Today these organizations routinely consume market data at rates exceeding one million messages per second, twice the peak rates they experienced only a year ago. This dramatic growth in market data is expected to continue for the foreseeable future, outpacing the capabilities of many current technologies. Industry leaders are extending and refining their strategies by including other types of data in their automated analysis; sources range from advanced weather prediction models to broadcast news. IBM developed an InfoSphere Streams based trading prototype running on a single 16 core x86 computer that could processing OPRA data feeds at up to 5.7 million options messages per second, with latencies of under 30 microseconds. Recompiling the application allows it to scale even further when deployed across a cluster of computers.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000050

Example 4: Health monitoring - Predicting the onset of illness earlier

Stream computing can be used to better perform medical analysis while reducing the workload on nurses and doctors. Privacy-protected streams of medical-device data can be analyzed to detect early signs of disease, correlations among multiple patients, and efficacy of treatments. There is a strong emphasis on data provenance within this domain, which is the tracking of how data are derived as they flow through the system. The “First of a Kind” collaboration between IBM and the University of Ontario Institute of Technology uses InfoSphere Streams to monitor premature babies in a neonatal unit. Data has been collected for more than 18 months from a hospital in Toronto, Canada. Remote telemetry from a U.S. hospital has been operational for a year using the same analytic routines. And earlier this year, additional hospitals in China and Australia began implementation of this solution.

Example 5: Transportation - Getting from here to there faster

IBM is working on an application in the IBM Smarter Cities™ Technology Centre in Dublin, Ireland where InfoSphere Streams receives GPS data once per minute from buses. A real-time display shows all buses as they move through the city. The pilot is working to extend this solution by providing real-time predictions related to arrival times for each bus at each bus stop, which will enable bus riders to better plan when to arrive at the bus stops, reducing waiting times. In the future, a personal travel planner could allow riders to receive recommendations based on real-time traffic monitoring.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000051

Figure 1-13 provides a picture of such a system.



Figure 1-13 Smarter transportation

In addition, other use cases of InfoSphere Streams are fast emerging in domains such as environmental monitoring and control (wildfire detection and water flow monitoring), energy and utilities industry (synchrophasor monitoring of smart grids and prediction of wind power generation), radio astronomy, x-ray diffraction using synchrotrons, fraud prevention, and many, many more areas.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000052

2

Streams concepts and terms

In Chapter 1, “Introduction” on page 1we introduced IBM InfoSphere Streams (Streams) from a business and strategic perspective. In this chapter, we look at the next level of detail by discussing and describing the primary concepts of Streams along with many of the terms typically used in a Streams environment. This chapter provides a high-level perspective of Streams that will better enable you to understand the lower levels of detail presented in the subsequent chapters.

We have organized this chapter contents into four major sections. First we discuss the technical challenges found with the types of business problems that Streams is designed to address. However, if you already can easily see where and how Streams can solve those types of problems with which common relational databases and standard extract-transform-load software packages would struggle, you may then choose to advance to 2.2, “Concepts and terms” on page 39, and begin immediately with the ideas that are specific to Streams.

In 2.3, “End-to-end example: Streams and the lost child” on page 48, we give details about one sample Streams application as a means to reinforce the concepts and terms introduced in this chapter, as well as a brief introduction to the numerous tools included with Streams.

Then in 2.4, “IBM InfoSphere Streams tools” on page 58, we introduce the Streams administrative and programming interfaces, some of which were used as we created our end-to-end example.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000053

2.1 IBM InfoSphere Streams: Solving new problems

The types of business problems that IBM InfoSphere Streams is designed to address are fundamentally different than those regularly addressed by common relational databases and standard extract-transform-load software packages. Consider the following fictitious example scenario.

After scheduling and receiving planned maintenance on your automobile, you pick up your automobile at 7:00 a.m. on your way to work. But then, as soon as your enter the multi-lane highway and proceed to the lane closest to the center of the highway (typically considered the lane for vehicles desiring to travel at higher speeds), your automobile begins running roughly, starts slowing down, and subsequently the engine quits and the automobile comes to a halt.

This is a scenario in which none of us want to find ourselves. It is not only frustrating, but, in this case, is extremely dangerous. Even staying in your automobile cannot be considered safe.

So, you decide to exit the automobile to attempt to get away from the traffic flow. However, the only way you can get off the highway is on foot and by crossing several lanes of high speed traffic, as shown in Figure 2-1. But, there appears to be an endless number of high-speed data points to examine as you begin your decision-making processing. So, how can you leave the highway safely?



Figure 2-1 Examining a stream of traffic

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000054

For the moment, let us think of the situation you are experiencing in terms of a data processing problem. How can you find a safe path to use (time and route) to cross the width of the highway where there are numerous vehicles travelling at various levels of high speed? As is typical of any highway, there are also many types of vehicles of different shapes and sizes. Some of the oncoming vehicles are higher and wider than others, thus obscuring your vision as to what is behind them. How can you make a decision as to when it is safe to cross?

If you did not have a good, continuous live view of the traffic moving on the highway as it is occurring, would you feel comfortable deciding when to cross the road? Even if you had, say, a satellite photo of the roadway that was 60 seconds old, would you feel safe then? Before you answer, consider the fact that satellite images are static. As such, you would need several satellite images to even try to extrapolate the relative speed of each automobile. And, in the time you are performing these extrapolations, the data would have already changed, introducing a measure of latency and resultant inaccuracy into your decision making process.

So what you really need is the ability to see a continuous view of the traffic flowing on the highway, with accurate representations of all the vehicles and their properties (length, size, speed, acceleration, and so on). Can you get that data with traditional data processing capabilities?

What if, for example, you modeled the above situation as an application using a common relational database server (database)? You would discover the following items:

- A database could easily model and record how many and which types of vehicles had passed through given points of the roadway at certain times in the past. The database could easily retrieve this data by location or by most other stored attributes, for as far back as you had chosen to capture and store this data on hard disk (a historical record).
- If you needed to track something resembling real-time (live) data, you would need to make a design choice. Does your model track data by vehicle (its current location and speed), or by section of roadway?

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000055

Note: In a relational database, the data model is generally static. Even if not, certainly the data model is not changing every few seconds or sub-second.

While the data in a standard relational database changes, you do not change the whole of the data frequently. Instead you insert or update a small percentage of the entire contents of the database, where that data resides forever (or at least until it is deleted).

In a relational database, it is the queries that are dynamic. Queries can even be ad hoc (completely unanticipated), but as long as the answer is inside your database (where you have modeled the data to support a given set of questions and program requirements), you will eventually receive a response.

Modeling to track data by vehicle has the advantage that it is finite, that is, there are only so many vehicles that you need to track. If you need to determine conditions for a given section of road, you would use the product of vehicles and vehicle data at a given geographic location. Also, you could create a vehicle record, and update that record on a frequent basis (as frequently as you can afford to from a hardware and performance standpoint).

You could model to track data by location on the roadway. But again, that is a huge amount of data to have to update and record, which may be beyond the limits that your hardware can support. Highways and locations that are modeled and recorded inside databases generally have to give focus to the relatively small number of known exits and merge points. Everything else is extrapolated, which does not help you if your automobile is stranded between recorded points.

However, even with all this modeling, can you satisfy the requirements of the application and get yourself safely off the highway? The short answer is *no*. You need a new application paradigm to satisfy these new requirements. That new paradigm is available, and is know as *stream computing*. As we discuss and describe this new paradigm, you see how using it could satisfy these new types of application requirements.

IBM InfoSphere Streams: A new and dynamic paradigm

As previously stated, a standard database server generally offers static data models and data, while supporting dynamic queries. So, the data is stored somewhere (most often on a computer hard disk), and is persistent. Queries can arrive at any time and answer any question that is contained inside the historical data and data model.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000056

However, in an IBM InfoSphere Streams environment, you do not have to store data before analyzing it. In fact, with Streams it is generally expected that the observed data volume can be so large that you just could not afford to make it persistent.

In simple terms, Streams supports the ability to build applications that can support these huge volumes of data that have the critical requirement to be analyzed and acted upon with low latency in close to or in real time. For this situation, static models are not sufficient; you need to be able to analyze the data while it is in motion. That is the value of Streams.

Note: The types of questions asked of the data stored in a database (data at rest) are typically quite different than those asked of real-time data that is in flight (data in motion).

For example, a standard relational database could easily answer a question about the retail price of every item sold for the past 6 months. Streams could answer questions about the current sub-second pricing at a world wide auction per sale item by location. The database is appending new data to the database for subsequent queries, while Streams is observing a window of real-time data in flight as it flows from the source.

Streams operates on (ingests, filters, analyses, and correlates) data while the data is still moving. By not having to store data before processing it, Streams addresses the types of applications with requirements for high data volumes and low latency responses.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000057

Standard database servers generally have a static data model and data, as well as dynamic (albeit often long running) queries. IBM InfoSphere Streams supports a widely dynamic data model and data. The Streams version of a query runs continuously without change. In Figure 2-2 we provide an illustration of these two approaches to data analysis.

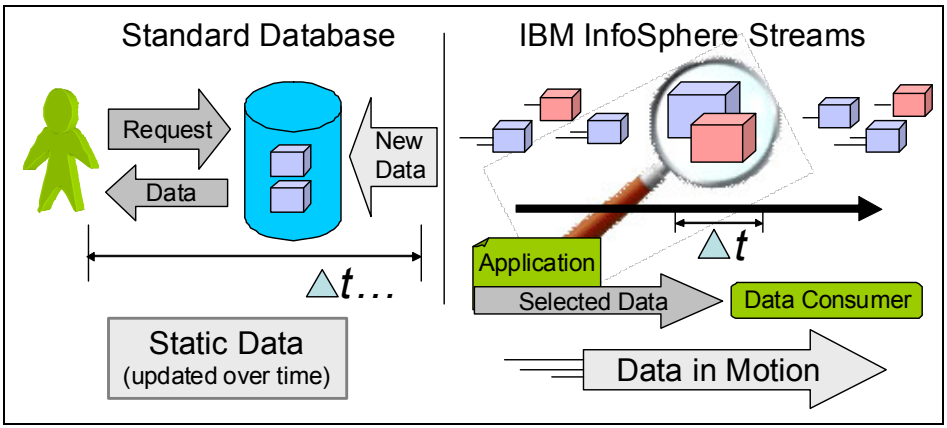


Figure 2-2 A standard relational database compared to IBM InfoSphere Streams

The traditional approach, shown on the left, involves storing data in a standard database, updating that static data over time, and periodically executing queries against that data. By contrast, in the Streams approach shown on the right, the data of interest within the stream is defined, and then processed as it flows by the application. The data is processed while it is in motion and due to its volume or velocity, it is typically not stored.

Queries in IBM InfoSphere Streams are defined by the Streams application, and run continuously (or at least until someone cancels them). In a database environment, a hard disk typically holds the data and the requester is the consumer of that data. In a Streams environment, data meeting the defined criteria is selected as it flows past the application, which then sends it to the appropriate consumer of that data.

Because Streams applications are always running, and continually sending selected data that meets the defined criteria to the consumers, they are well suited for answering *always-on* or *continuous* types of questions. These are questions such as, what is the rolling average, how many parts have been built so far today, how does production at this point in time today compare with yesterday, and other continuous, sub-second response time statistics. Because of the nature of the questions that Streams is designed to answer, it provides join-like capabilities that are well beyond those found in standard SQL.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000058

Streams easily and natively handles structured and unstructured data, both text and binary formats.

Note: With a standard relational database, the data is stored by persisting it to disk. With IBM InfoSphere Streams, the query (actually a Streams application) sends the appropriate data directly to the consumer of that data. To run a new query (ask a new question) with Streams, you run a new Streams application.

All software environments are composed of three major components: process, memory, and disk. With Streams, it is the process component that consumes the data and develops the response to the queries.

2.2 Concepts and terms

At the simplest level, Streams is a software product. It is well integrated with the operating system for high performance. In a world of database servers, LDAP servers, and J2EE compliant application servers, Streams is a platform with runtime and development components. As such, Streams contains:

- A runtime environment

  The runtime environment includes platform services and a scheduler to deploy and monitor Streams applications across a single host or set of integrated hosts.

- A programming model

  Streams applications are written in the Streams Processing Language (SPL), a largely declarative language where you state what you want, and the runtime environment accepts the responsibility to determine how best to service the request.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000059

Note: There are many adjectives used to describe computer languages, including the adjective *declarative*. In a declarative computer language, you describe the results (the computation) you desire, without defining how to gather those results (without defining the program flow and control).

SQL, and more specifically SQL SELECT statements, are probably the most widely known example of declarative programming. With an SQL SELECT, you specify only the resultant data set you want to receive. An automatic subsystem to the database server, the query (SQL SELECT) optimizer, determines how to best process the given query, that is, whether to use indexes, or sequential scans, what the table join order should be, the table join algorithm, and so on. The query optimizer decides how to return the resultant query in the fastest time, using the least amount of resources.

In Streams, a Streams application is written in the Streams Processing Language (SPL), which is a declarative language. A Streams application defines the data processing result that you want. The *how to* portion of that execution is determined by the Streams Runtime environment (which includes the scheduler), the SPL Compiler, and to a lesser extent, a number of property and resource files.

The Streams equivalent to a query optimizer arrives in the form of these three entities.

- Tools and monitoring and administrative interfaces

  Streams applications process data at rates higher than the normal collection rate of operating system monitoring utilities can handle. As an example, Streams can process twelve million messages per second with results returned in just 120 microseconds. To handle these types of speeds, a new processing paradigm is required. That paradigm is delivered by InfoSphere Streams.

2.2.1 Streams instances, hosts, host types, and admin services

A Streams instance is a complete, self-contained Streams Runtime environment. It is composed of a set of interacting services executing across one or more host computers.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000060

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 00006

Figure 2-3 displays a Streams instance and its member hosts. As with most software systems, the terms found within a Streams-based discussion are either physical or logical. Physical terms exist, and have mass and are measurable, whereas logical terms are typically definitions, and contain one or more physical terms.

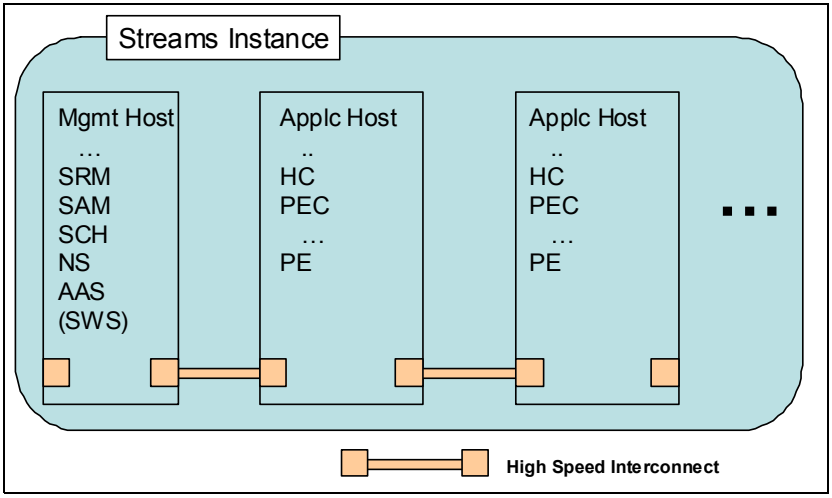


Figure 2-3 Streams instance, hosts, and more

The following comments are relative to Figure 2-3:

- As a term, a Streams instance is logical; a Streams instance serves as a container for the physical entities or host computers.

  Streams instances are started, and then stopped, and the entities contained inside a Streams instance are then modified, compiled, submitted, deployed, and so on.

  From an administrative standpoint, a Streams instance is created and then can be deleted. Creating a Streams instance is a relatively low-cost procedure.

- As a term, a Streams host is a physical term, and usually equates with a single operating system host. In certain contexts, a Streams host is also called a node.

  In order for it to exist, a Streams instance (a logical term) must be composed of one or more hosts (host being a physical term).

- A Streams host is exclusively one of three types: a Management host, an Application host, or a Mixed-Use host.
  - A Management host executes the resident services that make up the Streams Runtime environment proper; services such as the Streams Authentication and Authorization Service (AAS), which, among other responsibilities, verifies user identity and permission levels.
  - An Application host is dedicated to executing the actual Streams applications that were previously described as being the queries that run continuously inside Streams.
  - A Mixed-Use host executes both resident services (all or a subset of these services) and Streams applications, and are perhaps more common in development environments than in production environments.

    Note: Although the same Streams host could support two or more concurrently running operating Streams instances, that condition is probably not optimal from a performance standpoint.

    There is currently no coordination or communication of any kind between Streams instances. Because high-end Streams instances can easily make full use of system resources, locating two or more Streams instances per host is likely to produce performance bottlenecks.

- Management hosts or Mixed-Use hosts may provide one or more of the following Streams services:
  - Streams Resource Manager (SRM)

    The SRM service initializes a given Streams instance, and aggregates system-wide performance metrics. The SRM Service is also the Service that interacts with the host controller of each application host.
  - Streams Application Manager (SAM)

    The SAM service receives and processes job submission and cancellation requests, primarily through its interaction with the Scheduler (SCH) service.
  - Scheduler (SCH)

    The SCH service gathers runtime performance metrics from the SRM service, and then feeds information to the SAM service to determine which Applications host or hosts are instructed to run given a Streams application (Processing Element Containers).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000062

Note: The Streams Scheduler Service determines which Application host or Mixed-Use host runs a given Streams application's Processing Elements.

The default Scheduler mode is Balanced. The Balanced Mode Scheduler distributes a Streams application's PEs to the currently least active host or hosts.

The Predictive Mode Scheduler uses resource consumption models (CPU and network) to locate a Streams application's PEs on a given host or hosts, as different PEs are known to consume differing amounts of resources.

  - The Name Service (NS) provides location reference for all administrative services.
  - The Authorization and Administrative Service (AAS) authenticates and authorizes user service requests and access.
  - Streams Web Service (SWS)

    The SWS Service is the first optional Service listed, and provides web communication protocol access to a given Streams instance's administrative functions.

- Application hosts or Mixed-Use hosts may provide one or more of the following Streams services;
  - Host Controller (HC)

    The HC service runs on all Application hosts and Mixed-Use hosts, and starts and monitors all Processing Elements.
  - Processing Element Container (PEC)

    The PEC exists as an operating system binary program. Processing Elements exist as custom shared libraries, and are loaded by PECs as required by given the Streams application definitions.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000063

Note: Secure operating systems generally disallow one operating system program from having interprocess communication with another program. They also typically disallow sharing concurrent file or device access. Although observed functionality is generally reduced, an upside to this capability is greater process isolation and greater fault tolerance.

To accommodate secure operating systems, IBM InfoSphere Streams supports the concepts of Domains (groups of Application hosts, or nodes).

Other terms in this discussion include the following: Confined Domains, Vetted Domains, or Node Pools, Streams Policy Macros, and Policy Modules. However, these terms are not expanded upon further here.

- The last items related to Figure 2-3 is that it includes the presence of high speed network interconnects, as well as any optional shared file system.

2.2.2 Projects, applications, streams, and operators

In this section, we discuss and describe several of the key terms associated with Stream computing. A few examples are as follows:

- The data is stored in databases, which are used in typical database applications, as called records. In relational terms, the data is stored in a row and column format. The columns are the individually defined elements that make up the data record (or row). In Streams, these data records are called *tuples* and the columns are called *attributes*.

  The primary reason these items are called tuples versus records is to reflect the fact that Streams can process both structured (rows and columns) data and unstructured data (variable data representations, flat or hierarchical, text or binary).
- A stream is the term used for any continuous data flow from a data source. That data is in the form of tuples, which are composed of a fixed set of attributes.
- An operator is a component of processing functionality that takes one or more streams as input, processes the tuples and attributes, and produces one or more streams as output.
- A Streams application is as a collection of operators connected together by streams. A Streams application defines how the run time should analyze a set of stream data.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000064

The image displayed in Figure 2-4 is taken from IBM InfoSphere Streams Studio. This is the primary developer's workbench for Streams and contains the following items:

- The top portion of the image displays a Streams Processing Language (SPL) source file within the Source Editor View. The bottom portion displays the Application Graph View.
- This Streams application represents the smallest functional Streams application one can create. As such, the following are true:
  - The composite section defines this application's main composite operator, which encapsulates the stream graph that forms the whole program. Main composite operators are self-contained in the sense that their stream graph has no output or input ports. They also have no mandatory parameters.
  - Within the composite section, the graph section defines the stream graph of the application. This consists of one or more operator invocations.
  - The minimum functional Streams application defines at least one Source operator (called a Source) and one destination operator (called a Sink).

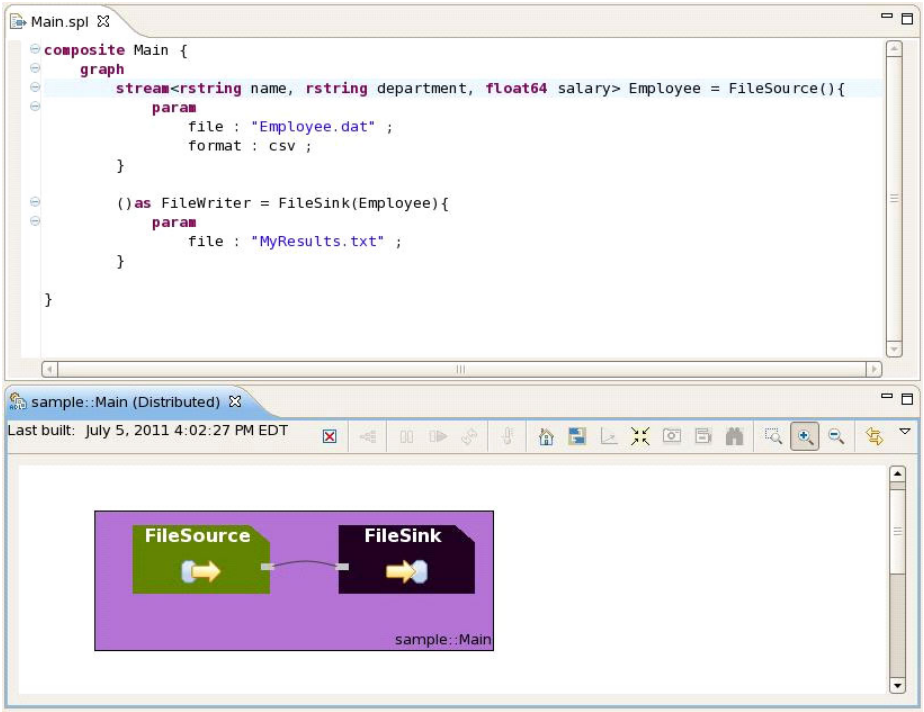


Figure 2-4 Inside Streams Studio - An example Streams application

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000065

Note: Technically, a Streams application does not require a Sink operator, and minimally only requires a Source operator. However, that style of Streams application may not be useful.

The Streams application shown in Figure 2-4 has its own Source operator (which refers to an operating system file) and its own Sink operator (which also refers to an operating system file). As such, it provides a self-contained Streams application.

As more complex Streams applications gain more and more operators, both Sources and Sinks, as well as intermediary operators (not pictured), you might optionally split these larger Streams applications into two or more Streams application Source Files. You then can introduce the Import and Export operators to enable these subset Streams applications to explicitly define their end points.

- The bottom portion of Figure 2-4 displays the Application Graph, which is essentially a logical view of a given Streams application, its operators, and resultant streams.
- A Streams project (not pictured) serves as a container to a Streams application (there is a one-to-one correspondence of project to application). A Streams project contains other properties and configuration settings related to a Streams application.
- As a term, the word *stream* represents the actual flow of tuples from operator to operator.

  Generally, Source operators output one stream, which may be consumed by one or more downstream operator(s).
- The definition of the data attributes (count and associated data types) being consumed by, or output by, a given operator is referred to as a schema definition (schema).

  Generally, most operators can accept only one input (schema definition) and produce only one output (schema definition). An operator can accept multiple input streams, as long as each is the same in its schema definition.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000066

Note: Additional Source and Sink operators are available in the Database, Financial Services, Internet, Mining, and SPL Standards toolkits. These operators include ODBC connectivity to relational databases, stock market data feeds, and support for HTTP, RSS, FTP, and other communication protocols.

If you still have not found support for your given data source or data target, Streams also allows you to create user-defined sources and sinks.

2.2.3 Applications, Jobs, Processing Elements, and Containers

We have previously defined an InfoSphere Streams application. However, this represents only the definition (the declaration) of a given request for service from inside the IBM InfoSphere Streams platform. We need further terms to define the instantiation and execution of these objects.

A Streams application can be submitted for execution in the form of a Streams Job. Generally, there is a one-to-many relationship of applications to jobs. As examples, an application may not be running (zero jobs), running only once (one job), or there may be several copies of the same application running (many concurrent applications, or more accurately, many concurrent jobs). Applications can be defined to accept runtime parameters, so that each of the concurrently executing jobs are performing distinct processing.

Job is a logical term, and exists in the form of one or more Processing Elements (PEs). Where the Processing Element Container (PEC) service runs on each Application host or Mixed-Use host, the PEC exists as an operating system binary program. PEs exist as shared libraries that are loaded by PECs, and then run.

Note: Ultimately, the discussion of Processing Element Containers (PECs) and Processing Elements (PEs) leads to a discussion of process architecture.

Rather than throwing dozens or even hundreds of executable processes at the operating system, and asking the operating system to perform process scheduling (with all of the operating system's associated and costly process context switching), the Streams Runtime environment provides this task, which is critical towards achieving ultimate performance.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000067

2.3 End-to-end example: Streams and the lost child

In this section, we provide a sample IBM InfoSphere Streams (Streams) application. The business problem used in this end-to-end example is described as follows:

At upscale alpine ski resorts, you may find that your room key not only opens the door to your guest room, but also serves as a charge card at the resort retail outlets, game rooms, and so on. This card also allows you access to the ski lifts.

Besides tracking customer preferences, this system can also be used to locate lost or missing children.

In the example application, we receive three input data feeds:

- Door scans, including the guest rooms, pool and other areas
- Retail purchases, which also indicate location, and entry to the ski lift lines

The Streams application we detail keeps the last known location of each child on the resort property, and matches that data with requests to find lost or missing children.

In the real world, these input data feeds would arrive from various live systems. To better enable you to recreate this application in your environment, the example below uses ASCII text files. The sample contents of these files are shown in Example 2-1.

Example 2-1 Sample data used in this example - Input and output

```
                    4th Col
                    here is
Door Scans          Boolean
                    1=Minor
Lname,Lname,x,x,ifIsMinor,Timestamp,Location
Morgenson,Sally,5,0,0,2010-02-14 10:14:12,Pool
Tanner,Luis,12,1,0,2010-02-14 10:14:12,Patio
Irwin,Mike,2,2,1,2010-02-14 10:14:12,Pool
Davidson,Bart,0,0,1,2010-02-14 10:14:13,Club House
Thomas,Wally,0,0,0,2010-02-14 10:14:14,Exercise Room
Williams,Eric,0,0,0,2010-02-14 10:14:13,Club House
Ignacio,Juan,3,1,0,2010-02-14 10:14:14,Game Room
Ryne,Paul,0,0,0,2010-02-14 10:14:13,Club House
Irwin,Mike,2,2,1,2010-02-14 10:14:14,Laundry Room

Lift Lines
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000068

```
Lname,Fname,Timestamp,Location,Minor
Davidson,Frederick,2010-02-14 10:13:55,Garfield Lift,N
Samuel,Corky,2010-02-14 10:13:55,Pioneer Lift,N
Ignacio,Juan,2010-02-14 10:13:55,Pioneer Lift,N
Stevens,Sam,2010-02-14 10:13:55,Tumbelina Lift,N
Bartolo,Izzy,2010-02-14 10:13:55,Breezeway Lift,N
Edwards,Chuck,2010-02-14 10:13:55,Pioneer Lift,N
Sylvestor,Hugo,2010-02-14 10:13:55,Tumbelina Lift,Y
Thomas,Wally,2010-02-14 10:13:55,Tumbelina Lift,N

Cash Register Swipes

Lname,Fname,Location,Timestamp,Minor
Green,Roxanne,Soda Shop,2010-02-14 09:23:56,N
Balmos,Fred,Gift Shop,2010-02-14 10:11:51,Y
Morgenson,Sally,Cafeteria,2010-02-14 10:14:08,N

Lost Children

Lname,Fname,Who-called
Irwin,Mike,Mum
Dewey,Ian,Dad

Results File

"Irwin","Mike","Mum","Laundry Room","2010-02-14 10:14:14"
"Dewey","Ian","Dad","",""
```

The following items relate to Example 2-1 on page 48:

- Five total files are displayed.
- The three input files called Door Scans, Lift Lines, and Cash Register Swipes, represent normal guest activity at the alpine ski resort, such as passing through a secure doorway, purchasing something, or entering the lift line.

  Each of these three files is modeled differently, with different attribute order and data types. A multi-attribute primary key is present in the form of a combination of last name (Lname) and first name (Fname) combined.

  A mixture of one/zero, and Y/N, represents the identification of children versus adults. This system is designed only to locate missing children.
- A fourth input file (Lost Children) contains requests by parents to locate their missing child.
- The fifth file represents the application result set.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000069

In the case of Mike Irwin, the system reported a last known location, while the system did not have a known location for Ian Dewey.

2.3.1 The Lost Child application

Example 2-2 displays the self-contained Streams application that is used as an example throughout the remainder of this section.

Example 2-2 Self-contained Streams application

```
composite Main {
   type
      DoorScanSchema = tuple< //
      rstring lastName, //
      rstring firstName, //
      int32 unused1, //
      int32 unused2, //
      int32 ifIsMinor, //
      rstring timeStamp, //
      rstring location> ;
      //
      LiftLineSchema = tuple< //
      rstring lastName, //
      rstring firstName, //
      rstring timeStamp, //
      rstring location, //
      rstring ifIsMinor> ;
      //
      RegisterSwipeSchema = tuple< //
      rstring lastName, //
      rstring firstName, //
      rstring location, //
      rstring timeStamp, //
      rstring ifIsMinor> ;
      //
      MergedChildSightingsSchema = tuple< //
      rstring lastName, //
      rstring firstName, //
      rstring timeStamp, //
      rstring location, //
      boolean isMinor> ;
   graph
      stream<DoorScanSchema> DoorScans = FileSource(){
         param
            file : "DoorScans.txt" ;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000070

```
            format : csv ;
            hasHeaderLine : true ;
            hasDelayField : false ;
            parsing : strict ;
      }

      stream<MergedChildSightingsSchema> MergeDoorScan =
Functor(DoorScans){
         output
            MergeDoorScan : isMinor = (ifIsMinor == 1) ;
      }

      stream<LiftLineSchema> LiftLines = FileSource(){
         param
            file : "LiftLines.txt" ;
            format : csv ;
            hasHeaderLine : true ;
            hasDelayField : false ;
            parsing : strict ;
      }

      stream<MergedChildSightingsSchema> MergeLiftLine =
Functor(LiftLines){
         output
            MergeLiftLine : isMinor = (ifIsMinor == "Y") ;
      }

      stream<RegisterSwipeSchema> RegisterSwipes = FileSource(){
         param
            file : "CashRegisterSwipes.txt" ;
            format : csv ;
            hasHeaderLine : true ;
            hasDelayField : false ;
            parsing : strict ;
      }

      stream<MergedChildSightingsSchema> MergeRegisterSwipe =
Functor(RegisterSwipes){
         output
            MergeRegisterSwipe : isMinor = (ifIsMinor == "Y") ;
      }

      stream<MergedChildSightingsSchema> MergedChildSightings =
Functor(MergeDoorScan, MergeLiftLine, MergeRegisterSwipe){
         param
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000071

```
            filter : isMinor == true ;
      }

      stream<rstring lastName, rstring firstName, rstring whocalled>
LostChildren = FileSource(){
         param
            file : "LostChildren.txt" ;
            format : csv ;
            hasHeaderLine : true ;
            hasDelayField : false ;
            parsing : strict ;
            initDelay : 2.0 ;
      }

      stream<rstring lastName, rstring firstName, rstring whocalled,
rstring location, rstring timeStamp> JoinedRecords =
      Join(LostChildren as LHS ; MergedChildSightings as RHS){
         window
            LHS : sliding, count(0);
            RHS : partitioned, sliding, count(1);
         param
            algorithm : leftOuter ;
            partitionByRHS : RHS.lastName, RHS.firstName;
            equalityLHS : LHS.lastName + LHS.firstName ;
            equalityRHS : RHS.lastName + RHS.firstName ;
         output
            JoinedRecords : lastName = LHS.lastName, firstName =
LHS.firstName, whocalled = LHS.whocalled, location =
            RHS.location, timeStamp = RHS.timeStamp ;
      }

      ()as FileWriterJoinedRecords = FileSink(JoinedRecords){
         param
            file : "JoinedRecords.txt" ;
            format : csv ;
      }

}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000072

2.3.2 Example Streams Processing Language code review

The following items are part of a code review related to Example 2-2 on page 50:

- The composite section provides the name for this Streams application, which is titled “Main”.
- The type section defines the schemas of the streams within this application.
- The graph section contains Streams Processing Language declarative commands proper, and makes up the majority of this file.
- There are five schema definitions within the type section. Any of these schema definitions could have been placed inside a given operator and could have been functionally equivalent. The purpose of defining them once at the start of the application is to increase source code readability, and increase code re-use. As you can see, some of the schema definitions are used multiple times inside this file.

  A schema definition has the general form of an attribute data type and then attribute name (comma and repeat). The attribute data types presented in Example 2-2 include String, int32, and Boolean.

  Note: The Streams Processing Language includes 22 or more simple data types, as well as collection types such as list, set, map, and tuples.

Reading data: Source stream operator

The first real statement of SPL source code in Example 2-2 on page 50 begins with the text `stream<DoorScanSchema> DoorScans = …` and ends with the trailing curly brace `}`.

- This statement defines a source (input) stream for this Streams application. This FileSource() operator is an ASCII text file reader.
- The syntax begins as “stream”, a schema definition, and then a stream name (with the stream name DoorScans).

  If any further operator in this application wants to consume the output of this operator, it could do so by using the stream name DoorScans. Generally, any operator can automatically support multiple consumers of the output stream that it produces.
- The stream name is followed by the output attributes specification to this operator. Here we could explicitly specify attribute names and attribute data types, but optionally make reference to a type declaration above and call it DoorScanSchema.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000073

- The text following the “param” keyword specifies modifiers to this operator; specifically, we set the file name, input file values are delimited with commas, the file has a header line, and the operator does not include a delay interval and should be parsed for strict adherence to data types.

Filtering and mapping attributes: Functor operator

The second statement of SPL source code in Example 2-2 on page 50 begins with the text `stream<MergedChildSightingsSchema> MergeDoorScan = …` and ends with a trailing curly brace `}`.

- This Streams operator is of type Functor(). In its simplest form, an SPL Functor() operator is similar to a Transform operator in a standard extract-transform-load software package.
- This Streams operator receives five input attributes, as determined by its source stream (DoorScans). The Source operator (and source stream) sends five attributes, so we receive five attributes.

  The Functor() operator compares the value of the int32 input attribute named “ifIsMinor” to the literal value of “1”.

  If the value of this input attribute equals 1, then the Boolean output attribute isMinor is assigned the value of “true”.

Note: This Streams operator also outputs five Attributes, as determined by the Schema Definition named MergedChildSightingsSchema.

These source and output Schema Definitions for this operator are equal in Attribute count, and also in the collection of most attribute names. These two Schema Definitions do differ, however, in the attribute data type for the last attribute, which is named ifMinor. We change (recast) that data type from Int32 to Boolean to demonstrate this technique (and also to make the definition of this data stream consistent with the other data streams we manipulate in a related manner in the following sections).

Note: Algorithmically, what we are doing here is recasting this input Attribute data type. Because computers cannot assume that a numeric one should translate to a Boolean “true”, we have this block of statements.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000074

Four more operators

The next four Streams operators in this application are as follows:

```
stream<LiftLineSchema> LiftLines = …
stream<MergedChildSightingsSchema> MergeLiftLine = …
stream<RegisterSwipeSchema> RegisterSwipes = …
stream<MergedChildSightingsSchema> MergeRegisterSwipe = …
```

These operators provide similar functionality to the two Streams operators we previously described, but demonstrate no new techniques. The next significant operator we describe is named *MergedChildSightings*.

Merging (input) Streams: (Any) operator

The Streams Processing Language statement that begins with the text `stream<MergedChildSightingsSchema> MergedChildSightings = …` is also an operator of the Functor type. We need an operator of the Functor type to both merge the sources of location data and also to apply a filter condition (isMinor = “true”). This Functor operator gives us our three main, merged input data streams as a single output stream.

Reading the list of lost children

Consider the following SPL statement:

```
"stream<rstring lastName, rstring firstName, rstring whocalled>
LostChildren = …"
```

In this statement, we read the file containing the names of the children who have been reported as lost. Note the additional “initDelay” parameter used with the FileSource operator.

Note: In a real situation, this Streams application would start and run for a long time. In this simple example, we are using operating system text files with only a few lines of data. The challenge in testing or developing in this manner is the sequencing of multiple operating system programs and multiple Streams PECs.

In this example, we are reading from four small input files, and it could happen that we read from the single file containing the request to find a lost or missing child (and perform the join) before we ever load the three files that contain the current child location (and thus return zero joined tuples).

To better accommodate development and testing on such a small sample, the operator that reads the missing children names includes the modifier named initDelay. This modifier causes this Source stream to wait two seconds before completing its initialization.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000075

Joining, aggregating, or sorting static (normal) data

The next requirement in our Streams application is to compare the tuples of the known child locations with a request to locate a child. This task is completed using an SPL Join operator. However, joining, aggregating, or sorting streamed data is fundamentally different than joining or aggregating data in a static data repository. In this section, we describe that challenge.

When sorting a standard list of data, you cannot output any of the tuples until you have first received all of those tuples.

However, based on that statement, what if you are receiving a list of tuples to be sorted alphabetically, but the last tuple to be received starts with the letter “A” (and should be output first)? Sorting data has this requirement, and so does aggregating data, which has an implied sort. For example, when aggregating by department, you first sort the list of company-wide data by department.

Joining data also implies this condition, because you nearly always sort at least one side of the joined lists to achieve better performance.

The assumption here, however, is that you know where or when the end of the data is located or received. Remember, you need to receive the last row before you can complete a sort. Because streaming data never terminates, how then do you sort (or aggregate, or join)? The answer is to create windows, or marked subgroups of the data received.

With InfoSphere Streams, an input data stream may never terminate, and yet you still need to join data, aggregate data for given subgroups, or sort data within given subgroups. To accomplish this program requirement, Streams defines the concept of (moving) windows of data.

Joining, aggregating, or sorting streamed data

In an endless stream of data, IBM InfoSphere Streams allows data to be promoted into subgroups, called windows (of data), by defining a number of conditions. Here is a list of some of the possible conditions:

- By row count
- By elapsed time
- By a marker in the stream sent by an upstream operator (this marker is referred to as punctuation)
- By an evaluation expression against the attribute values
- A combination of the above conditions

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000076

With Streams, there are two types of windows: tumbling and sliding. A tumbling window fills with data, performs or supports whatever operator is specified, and then moves to an entirely new window (which is an entirely new sub grouping) of data. It tumbles forward through the data, end over end.

For example, a tumbling window could be used to aggregate streaming data, and report every time a new key value pair has changed its descriptive attributes. From our alpine ski resort and lost or missing child example, the windows would report every time a child's location changes. This window would be tumbling because it dumps one key-value-pair tuple in favor of another.

A sliding window fills with its first sub-grouping of data, performs or supports whatever operator it is an element of, then continues with this same data, adding it incrementally, or sliding, as new data is received.

Note: As previously stated, Streams windows are used with at least three Streams operators, including Aggregate, Sort, and Join. Often you might choose to model a window using Aggregate or Sort, and then apply this definition to the Join operator that simultaneously manages two Streams, which are the two sides of the join pair.

Much like standard relational databases, Streams supports full joins, and left and right outer joins. Streams also allows joins on more than just relational algebra, meaning that a left side expression can evaluate to true, and it will join with all right side expressions that are also true, without either side having to be equal to each other.

Further, Streams supports full intersections of data, which means that, by default, Streams attempts to join all tuples from the left side of the join pair to the right side of the join pair, and then joins all right side tuples with the left side tuples.

In our example use case, we want an outer join and return the child's location if it is known; otherwise we return null. We still want a response, even if it is null. However, we want only to join a child identifier with location. In Streams terminology, this is referred to as a *one sided join*.

Joining tuples: Join operator

At this point, we have met all of the prerequisites for our Streams Join operator, and continue with Example 2-2 on page 50. This discussion continues with the source line that begins as follows:

```
stream<rstring lastName.., > JoinedRecords = Join(…"
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000077

Where:

- This operator uses an explicit schema definition. We could have used a predefined schema definition, but instead we decided to show an explicit schema definition.
- The windows expression proper is in the parentheses following the Join keyword. It states the following:
  - LostChildren lists a window size of zero tuples. MergedChildSightings is listed with a window size of 1 tuple per child.
  - That condition makes this a one-sided Streams Join, with the behavior that the join is only triggered on the arrival of LostChildren tuples, and not MergedChildSightings tuples.

All Join operators use only sliding windows, as is the nature of a join.

Writing data: Sink operator

The last Streams operator in Example 2-2 on page 50 is as follows:

```
() as FileWriterjoinedRecords = …
```

It writes the observed result to an operating system ASCII text file, with sample results displayed at the bottom of Example 2-1.

Although we write to a file here to simplify this example, in a real situation we could have several concurrent subscribers to this output stream, including electronic pagers, event alerts, text message generators, and so on.

2.4 IBM InfoSphere Streams tools

IBM InfoSphere Streams offers a combination of command-line tools and graphical tools (web-based and fat client), as well as tools to perform administrative and monitoring functions, and application development functions, as examples. In this section, we detail two of those tools:

- IBM InfoSphere Streams Studio (Streams Studio)

  An Eclipse based developers and administrator's workbench.
- streamtool

  A command-line interface program that is used to perform many tasks, such as listing Streams instances, and starting and stopping instances.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000078

2.4.1 Creating an example application inside Streams Studio

In this section, we provide the steps used to develop the sample Lost Child application using the Streams Studio developer's workbench. This example should give you a better understanding of IBM InfoSphere Streams Studio. In the development of the sample application, the following assumptions are made:

- The complete IBM InfoSphere Streams product is installed, including a base Eclipse installation with Streams Studio plug-ins.

  The operating system platform in the example is Red Hat Enterprise Linux Version 5.x.

- You have created the four sample input files displayed in Example 2-1. Add them to a folder in Eclipse later.

Complete the following steps:

1. IBM InfoSphere Streams Studio exists as a collection of plug-ins to the Eclipse developers workbench. To run Streams Studio, launch Eclipse.

   Note: Eclipse is an open source and extensible developer's workbench. As IBM InfoSphere Streams Studio is based on Eclipse, it inherits many Eclipse terms and concepts.

   In Eclipse, a given parent directory contains all of your Projects, and any metadata related to work bench preferences. In Eclipse, this parent directory is called your Workspace. Project is also an Eclipse term, and is thus inherited by Streams. In Streams, there is a one to one relationship of Streams project to Streams application.

   If you are prompted to select a Workspace when launching Eclipse, select the default value and click the **OK** button.

   If this is your first time that you are running Eclipse, you may receive a Welcome window. Feel free to close it.

2. You need to create a small number of Streams objects before you can begin entering Streams Processing Language statements. To accomplish this task, complete the following steps:

   a. From the Studio menu bar, select, **File** → **New** → **Project**. A dialog box opens.

   b. Select **InfoSphere Streams Studio** → **SPL Project** → **Next**.

   c. Give this new Streams project a name and click **Finish**. If you are prompted to change Perspective, click **Yes**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000079

Note: Perspective and View are additional Eclipse terms.

A View in Eclipse is a distinct area of the window, and in other environments might be called a panel or frame. There are so many Views in Eclipse that they are organized into groups of Views called Perspectives.

Streams Studio provides two Perspectives with its Eclipse plug-ins, and a number of Views.

If this is the first time you are running Eclipse and creating a Streams Project, you may receive a prompt to change Perspectives. If so, click, **Yes**.

3. At this point you are ready to enter Streams Processing Language (SPL) statements.

   To begin, you may first create either the sample Streams application shown in Figure 2-4 on page 45 or Example 2-2 on page 50. To do so, create an SPL source file by selecting **File** → **New** → **Other** → **SPL Source File** → **Next**.

   In the dialog box that opens, pick your newly created SPL project. Leave the other fields unchanged and click **Finish**.

   In the resulting Main.spl editor, enter the SPL source code. As you do, be aware of the following items:

   a. If you produce an error in your syntax, a red marker highlights the source of the problem. For a more detailed description of the error, you may also move to the Problems View.

   b. While entering statements in the Editor View, you can enter Ctrl+Spacebar to produce context sensitive help. You are prompted with a next option or continuation of the current command or modifier.

   c. You are done when your SPL Source code matches the statements you have been entering from the example, you have zero errors after saving the file, you have copied your four input files into the data folder, and your Project Explorer View is what is shown in Figure 2-5 on page 61.

4. To run the Streams application, complete the following steps:

   a. Click **Streams Explorer View** tab, and in the list shown, right-click **Instances** and select **Make instance…**. Enter the name MyInstance and click **OK**.

   b. Navigate to the MyInstance entry under Instances, right-click it, and pick **Start instance...**. Allow the instance to start before continuing.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000080

c. Return to the **project explorer view**, right-click **Distributed [Active]**, and select **Launch**. Use the default values shown in the resulting dialog box and click **Continue**. Click **Yes** if you are prompted to save any unsaved changes.

d. After 5 or 10 seconds, select the data folder under Resources and press F5. You should see a new file named `JoinedRecords.txt`. Open the file to confirm that it contains the expected output.

These step execute your Streams application. As Streams jobs never terminate, you should navigate to the 0:Main entry under Jobs on the Streams Explorer tab, right-click **0:Main**, and select **Cancel job**.

In the following pages, we show a number of images from the IBM InfoSphere Streams Studio. For example, Figure 2-5 displays the Project Explorer View.

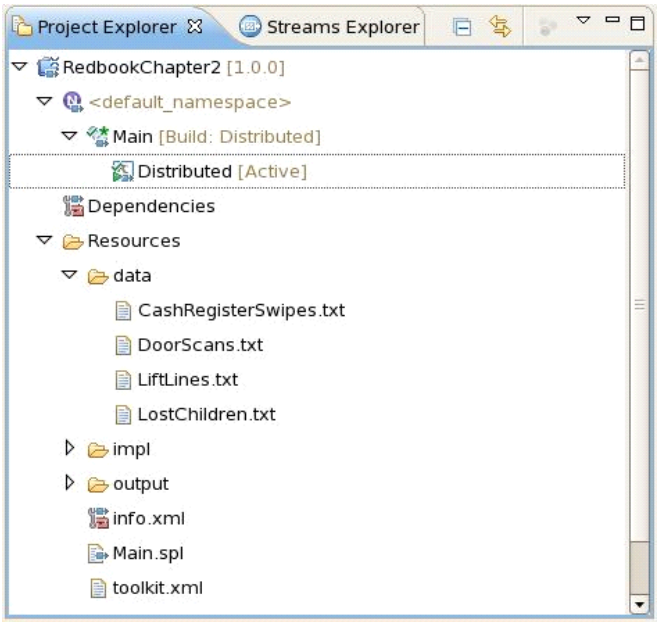


Figure 2-5 Project Explorer View from IBM InfoSphere Streams Studio

Where:

- The Project Explorer View operates like most file explorer interfaces. The highest level object displayed in this View are Streams projects, followed by the given contents of the project, the SPL source file, and so on.
- Figure 2-5 displays a data subdirectory where all source and target operating system data files are located.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000081

- Other files displayed in Figure 2-5 on page 61 contain configuration settings, and commands to start and stop the given Streams instance, and to submit and cancel the associated Streams application (job).

Figure 2-6 shows the Streams Application Graph View from InfoSphere Streams Studio.

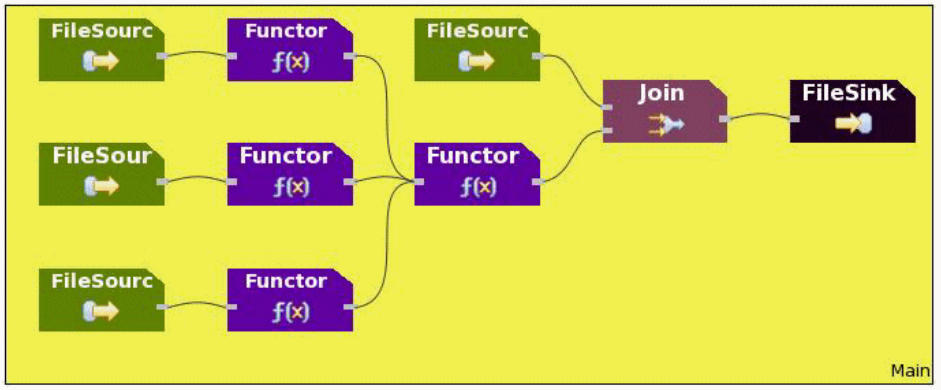


Figure 2-6 Streams Application Graph View from InfoSphere Streams Studio

Where:

- The graphic in Figure 2-6 shows the logical design of the given Streams application from an operator and Stream perspective.

  If you right-click and select **Open Source** any of the operators, your cursor will move to that point in the Streams application source file.

- You can use the Streams application Graph to ensure that all of your operators are connected as you intend them to be. An operator with no input/output Stream most likely indicates you have misspelled a given Stream in your SPL source file.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000082

Figure 2-6 on page 62 displays the logical design of a given Streams application, but Figure 2-7 shows it executing.

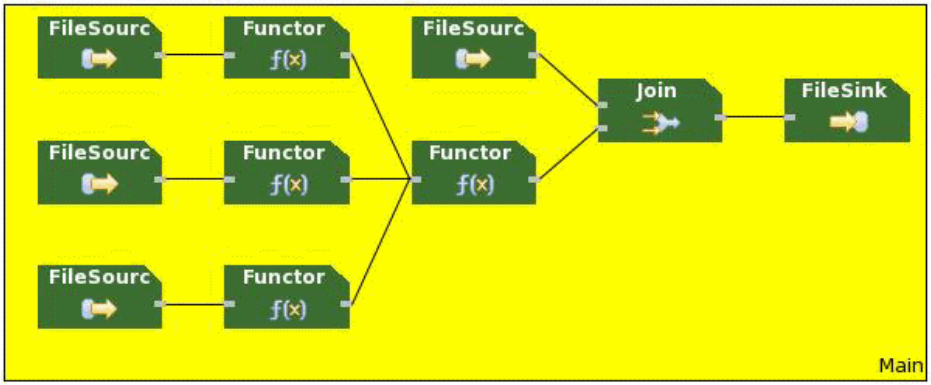


Figure 2-7 Streams Live Graph View from IBM InfoSphere Streams Studio

Where:

- The colors of the displayed operators change as the operator state changes. If you are not seeing the color changes, this is most likely due to the fact that this application runs quickly.
- Clicking an operator displays metrics about the number of tuples processed and other information.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000083

The Metric View is shown in Figure 2-8, which shows the most detailed and low-level performance statistics.

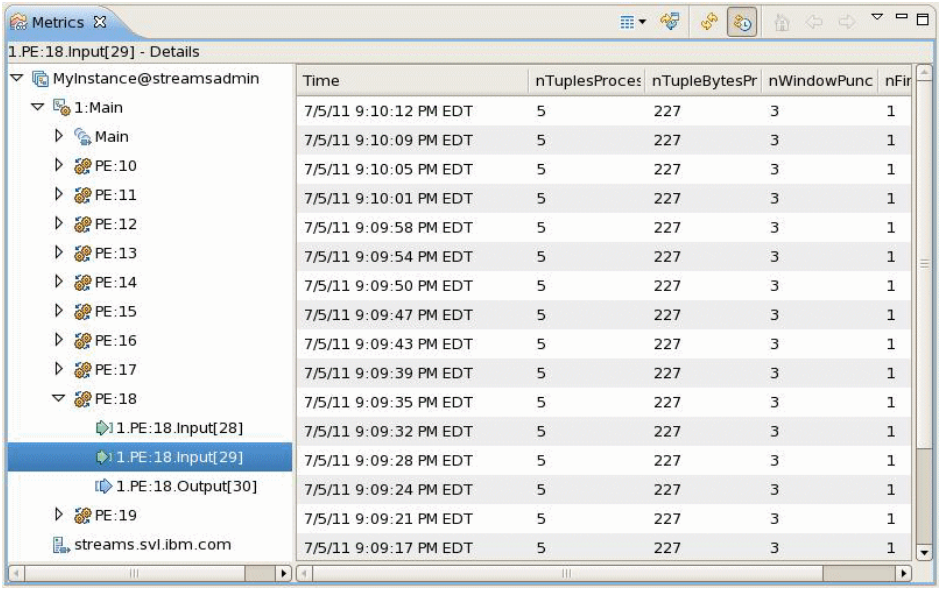


Figure 2-8 Streams Live Graph Outline View from IBM InfoSphere Streams Studio

The Streams Metric View displays rows counts, bytes, and so on, operator by operator. This View is handy for debugging, as well as for monitoring, your Streams application.

You can also concurrently run this View with the Streams Application Graph, moving back and forth as you click various elements.

Using streamtool

The IBM InfoSphere Streams streamtool offers a command-line (character based) interface that can also be used to accomplish all the steps we performed in previous sections. Most invocations in the streamtool require that you supply a Streams instance id-name. If you do not know your Streams instance id-name, you can run the following from the command line:

```
streamtool   lsinstance
```

(For clarity, lsinstance is “L S Instance” in lowercase and is one word.)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000084

Note: If your Streams instance id-name contains a special character, for example, the @ sign, then you must always enter this id-name inside a pair of double quotes. Otherwise, your operating system command interpreter will observe and process those special characters.

The above command reports all currently running Stream instances on the given server. If you do not know your Stream Instance id-name and that instance is not currently running, you may then start the instance inside InfoSphere Streams Studio and then return to this point.

To start or stop a given Streams instance, run the following commands:

```
streamtool   startinstance   -i   id-name
streamtool   stopinstance   -i   id-name
```

Note: It is common for new developers to exit the Streams Studio design tool and leave their Streams instance running, and then not know how to terminate or even restart the Streams instance.

To force a shutdown of the Streams instance, run **`streamtool stopInstance`** and append the -f flag.

By default, the Streams instance SWS service is not running, but is still required to run the IBM InfoSphere Streams Console, which is a web-based application. To start the Streams SWS service, complete the following steps:

1. To obtain a list and location of all running Streams Services, run the following command:

   ```
   streamtool   lshost   -i   id-name  -long
   ```

   The sample output below, indicates that SWS is not running:

   ```
   host          Services
   host-x        hc, aas, nsr, sam, sch, srm
   ```

2. To start the SWS Service, run the following command:

   ```
   streamtool   addservice   -i   id-name    --host    host-x   sws
   ```

3. To get the fully qualified URL that allows access to the Streams Console, run the following command:

   ```
   streamtool   geturl   -i   id-name
   ```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000085

Enter the return value inside a supported web browser, as shown in Figure 2-9.

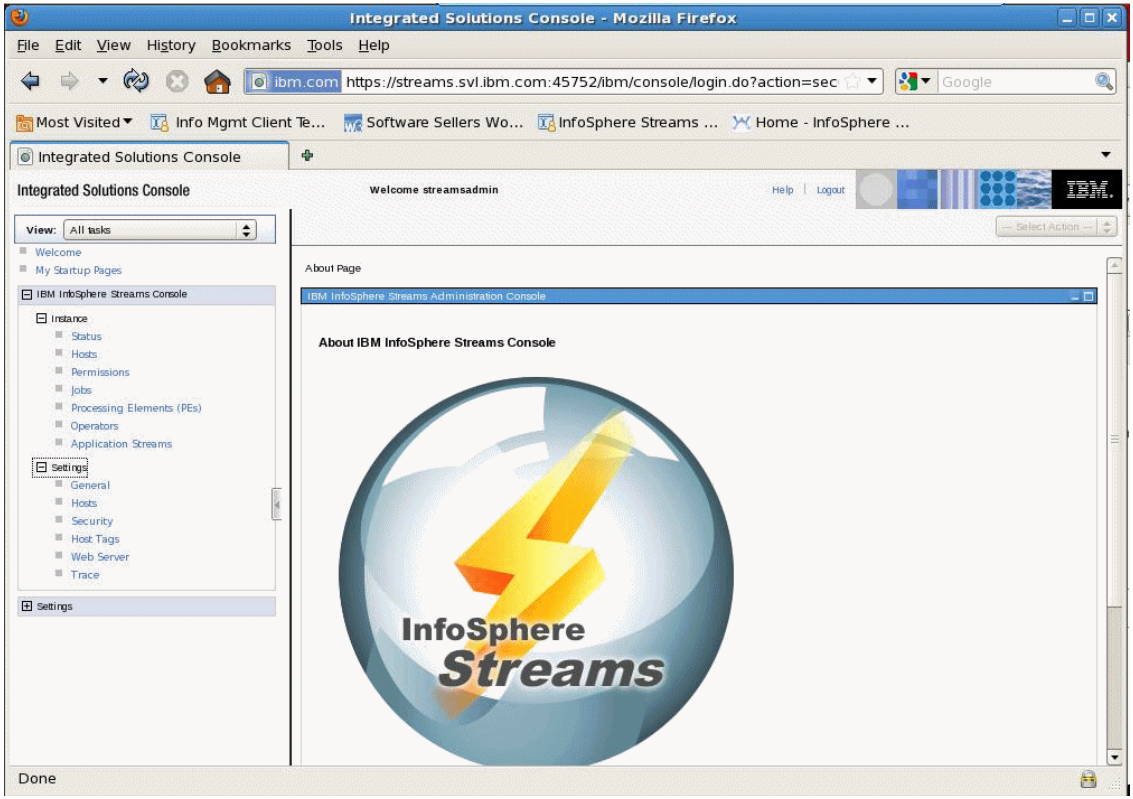


Figure 2-9 IBM InfoSphere Streams Console - A web application

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000086

3

Streams applications

In previous chapters of this book, we introduced you to Streams applications, which are composed of a combination of data streams and operators on these streams that allow InfoSphere Streams to ingest, process, and output data.

In this chapter, we provide a closer look at Streams applications, and outline the activities required to design a well performing application. The design process will lead you to consider the purpose, inputs, outputs, and dependencies of the system in question. We also outline features to address the performance requirements that should be considered during application design. In addition, we contrast Streams application development with traditional application development and highlight a number of novel attributes of Streams applications.

From early experience with implementing Streams applications, we list a number of use cases and example design patterns that should help you understand the best positioning and use of the Streams platform and enable new developers to better relate their requirements for established systems.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000087

The scalable, continuous processing model of InfoSphere Streams makes it ideal for the following types of applications:

- RIGHT NOW applications: In RIGHT NOW applications, an action needs to be taken as soon as there is enough information to predict or detect an opportunity. Examples of RIGHT NOW applications include preventing cyber security attacks, providing a better experience for customers at the point of contact, taking advantage of a trading opportunity in capital markets, and stopping a security threat.
- MORE CONTEXT applications: In MORE CONTEXT applications, there is a desire to take advantage of data that would have been too time sensitive or too expensive to incorporate using previous computing platforms. By automating the analysis in a continuous way without storing data, the Streams platform can deliver insights from this data that give a competitive advantage. Examples of MORE CONTEXT applications are those that enhance customer interactions with context from social media, such as blogs and tweets, or security applications that use computers to simultaneously examine numerous video streams, saving the streams and alerting humans only when something unusual happens.
- DYNAMIC INSIGHT ASSEMBLY LINE applications: In assembly line applications, the application naturally can be structured much like a manufacturing assembly line operating on data items element by element as the elements stream by. These applications can be dynamically reconfigured based on data or application needs. One example of this kind of application is Call Detail Record (CDR) processing for telecommunications where a well-defined set of steps are performed to mediate and remove duplicates from the records. CDR applications can be extended by providing features such as real-time summary statistics or immediate offers for customers based on usage patterns. In the past, these applications were often implemented in suboptimal ways because Streams did not exist. In real customer applications that use the Streams platform, there has been 10x reductions in servers, storage, and processing time by rebuilding the applications in Streams.

Most successful Streams applications take advantage of one or more of these application types. There are many considerations for determining when to use Streams for your applications. The following sections provide more detail about issues such as application design and performance. There are standard practices that should be considered with all types of Streams applications, such as those in the following list:

- Process data as soon as it enters the enterprise, or as it is generated in the enterprise.
- Process data before it is written to disk (it is okay to use disk as an intermediate step if you plan to change your processes later to be more real time).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000088

- Streaming analysis should typically come before, or in parallel with, a database or Hadoop infrastructure.
- Streaming analysis should be used in conjunction with your database or Hadoop infrastructure to obtain models and reference data.
- Streaming analysis should keep as much information as possible in memory. Memory is getting big enough to keep large collections of data active across a distributed cluster. Streams provides windowing functions to help manage data in memory.
- Do not use Streams for transactional workload. You can make a Streams application highly reliable, but if you need 100% transactional guarantees, you should use a database. When a database is too slow, you can use Streams, but have to compromise by giving up some level of guarantees or by providing more application logic.
- Do not use Streams when there are multipass algorithms that are highly complex or need to process more data than will fit in memory; use a database or Hadoop instead.
- Use Streams when there are a variety of data types to analyze, such as voice, video, network packets, audio, and waveforms.

3.1 Streams application design

When implementing a business requirement using the IBM InfoSphere Streams platform, the first thing you will need to do is to develop a high-level system design identifying the main components of the system.

As shown in Figure 3-1 on page 70, there are two main components to this design:

1. A logical design identifying the application components to be produced. The logical application design is discussed in the subsequent sections of this chapter.
2. A physical component design identifying the runtime components and hardware platform required to support your application and deliver the level of performance that is required. This physical component design is discussed in Chapter 4, “InfoSphere Streams deployment” on page 139.

At run time, the Streams Scheduler determines how to efficiently segment the application across the hardware hosts. The application should be segmented to produce suitably sized deployable processing elements for the Streams Scheduler to deploy. In 4.5, “Application deployment capabilities” on page 179, we discuss the aspects of appropriate application segmentation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000089

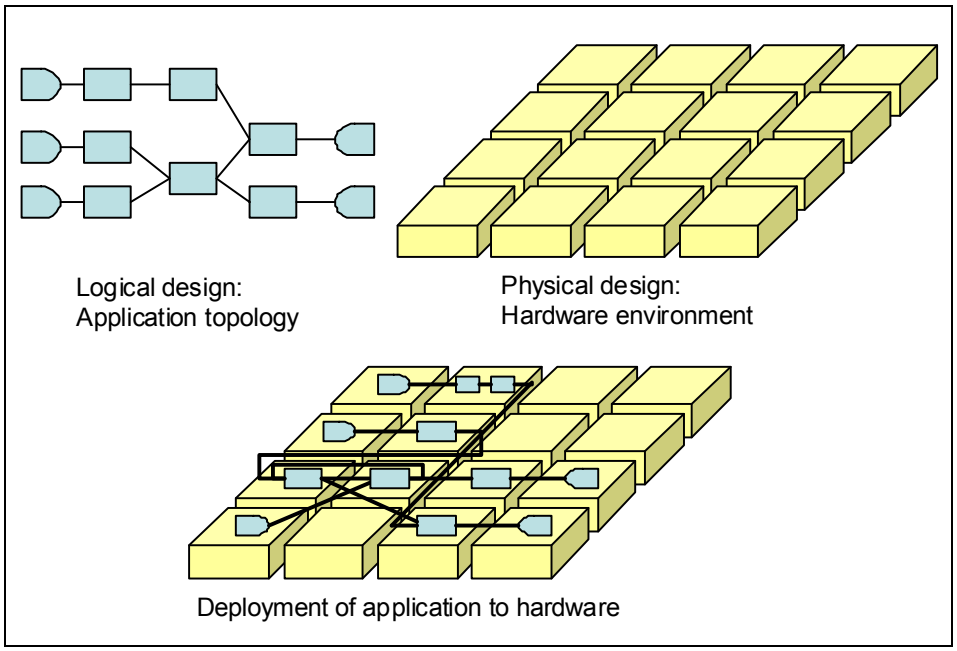


Figure 3-1 Application design, physical design, and deployment

3.1.1 Design aspects

Begin the design of the Streams application by considering the requirements that the system should satisfy and determine if Streams is a natural fit. Next, explore the interfaces of the system, such as what are the inputs that you can use for the application and what are the known outputs that you need to generate. Some of these interfaces are likely to be existing automated systems, so you should be able to determine the technical details of these interfaces. Some of the inputs and outputs may be new, predicted, or unknown, in which case you need to build in the flexibility to handle these less well-defined interfaces. A more detailed discussion of the design of data sources is contained in 3.1.2, “Data sources” on page 72 and outputs are further discussed in 3.1.3, “Output” on page 75.

In cases of complex analytic requirements, it may be necessary to enhance the standard Streams operators by calling out to external software packages or to compile and link the application with existing code. If this is the case, then the design needs to consider the nature of the existing analytics to be utilized. This is discussed further in 3.1.4, “Existing analytics” on page 78.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000090

As Streams is specifically built for high performance, the success of the application relies on identifying, estimating and forecasting the performance that is required from the application. This is covered in 3.1.5, “Performance requirements” on page 79 and includes a discussion on the volumes of data and the processing speed required.

Figure 3-2 graphically summarizes the steps for this initial stage of application design:

1. Identify the purpose of the proposed system and determine whether or not it aligns with one of the application classes.
2. Define the inputs and outputs required.
3. Identify existing analytics (that is, Streams operators and external libraries) to use.
4. Assess performance, time, and reliability requirements.

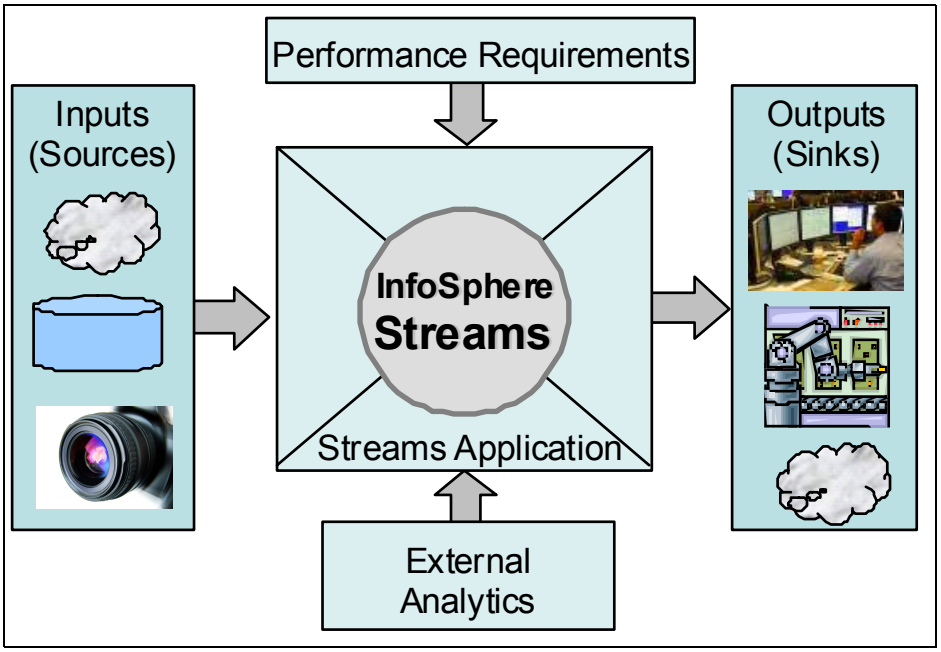


Figure 3-2 Streams application high level design aspects

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000091

3.1.2 Data sources

Identifying and understanding the data sources available to the Streams application is an essential aspect of the design. Some may be well known and well defined, and you may need to ensure that you have flexibility to accommodate additional sources when they become available.

Information content

What is the information content of each source? Regardless of the format and structure of the data source, what are the relevant data items that you expect to extract? Some examples are:

- A feed of video from a traffic camera may allow extraction of information about which vehicles are travelling on a road. The same camera may allow identification of accidents and alert relevant authorities to react.
- Data sources from medical sensors can give specific readings of a patient's bodily condition.
- Feeds of data from web logs (blogs) may allow a quick and appropriate response to the reaction of a business announcement.

Method of access

How can you access the data of each source? A data source can be accessed using a number of flexible, configurable options. As examples, they may be:

- File based, for use when data is being deposited on a locally accessible file system.
- TCP based, so you can access a data stream through a TCP network connection, providing reliable communication of the stream contents from the originator. The Streams Source can be configured to be the server or client side of a TCP connection depending on the requirements.
- UDP based, so you can access a data stream through a UDP network connection. Unlike TCP communications, UDP does not support reliable message transfer, so should only be used in situations where it would be acceptable to lose some of that data to gain additional speed of transfer. The Streams Source can only be the server side of a UDP connection, so the system originating the data needs to connect to the Streams UDP socket.
- Databases, for use when the data resides in a relational database. The Database Toolkit packaged with Streams contains operators for accessing a database.
- Sensors, which may use TCP or UDP for transmitting data, or may use other types of communication methods and libraries to access the sensor data.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000092

Because Streams applications can use existing libraries to access data or perform operations, sensor data can easily be accommodated.

These options provide different levels of reliability. In some cases, reliability may be essential, in other cases, missing some tuples from an input source may be an acceptable alternative:

- Is the input data being pushed from a remote system? If you miss the opportunity to observe and consume this data, will it be lost?
- Is the data, in some way, buffered or stored so that you can request the contents when you are ready?

Variety of sources

How different is the information content of the various sources?

- You may consume a large number of data sources that all contain the same type of data, such as a huge number of temperature sensors from a manufacturing process.

  Even if the streams are in different formats, it should be possible to convert to a common format, extract the common components, or merge the streams and process the streams in a similar way.
- You may consume a number of different types of data, such as news reports (textual and TV video), stock market feeds, and weather reports to predict stock price movement.

  When the data is of different types, you need more complex comparison, aggregation, and join logic to be able to correlate the streams of data. In this case, it is useful to identify where there are common identities (or keys, in database terminology) that will allow us to relate one stream to another by using, for example, the built-in Join operator.

Structure of data

How well organized is the information that you receive from the data source and how much of the information is likely to be useful? Some data sources are more structured because they are in a concise form and have perhaps been preprocessed, and other data sources are less structured. Although there is not a precise definition, many people in the industry talk about structured and unstructured data:

- Structured data is typically well organized. It is likely to have originated from another computer system, and it will already be processed into a refined form. Therefore, the data received will be known and it is possible that the data will be filtered to provide only those data elements of interest.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000093

An example of a structured data source is the feed of data that can be acquired from stock markets. The data will be well defined in terms of the attributes that you will receive and the meaning of the data (such as the trading prices for particular stocks) will be well understood.

- Unstructured data is typically less well organized and will need to be identified and extracted from the source form. Discerning information of relevance could require the use of specialized algorithms that need to be integrated with the Streams application.

  An example of an unstructured data source is the wealth of blog data that is published on the Internet. This data is written in free-form text, can be about a wide range of topics, and may be reliable or unreliable sources of information and opinion. As an example, a market research company may be looking for opinions on products in the blogs of consumers, but will have to process a significant volume of data to enable the extraction of the relevant portions.

What is the data format of the information received from the data source? Streams supports two main forms of input data:

- Unicode text as a default: Each line of text is read by the Streams data source and turned into a tuple. A tuple is the Streams term for the individual element of data. Think of a tuple as a row in database terminology, and an object in C++ and Java terminology.
- Binary: As examples, this is the type of data format that comes from video and audio inputs.

These form of input data are translated into the rich type system supported by Streams.

Error checking and optional data

How consistent is the information in the source data stream? In the situation where you have highly structured data, each field in the incoming data will be present, and will have a valid and meaningful value.

However, you may find that frequently the data has values that are omitted from the input, or those values may be invalid. For example, a field that you expect to be a number might be empty, it might include non-numeric values, or it might be a valid number but far larger or smaller than you expected.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000094

You should identify the level of error checking required and specify what to do when errors or omissions are observed:

- You might need to detect errors in the input stream and either correct or discard these elements.
- Certain attributes in the stream tuples may be optional, and therefore not present in some cases. Appropriate default values should be set in this case.

Control of the data source

In some cases, the data sources may be controllable. that is, they have some mechanism by which you can modify the behavior of the system generating the data. In this case, the results of the Streams application can be used to affect the data produced.

One example of this situation is in the manufacturing use cases where you may control the conditions and ingredients of the production process to fine-tune the results and improve the quality of the product. Another example of data source control is in the use of security cameras, where you may control the direction and zoom of a camera to observe an event of interest.

3.1.3 Output

There will be downstream systems and applications that consume the data output by the Streams application. The Sink operators of the Streams application need to be configured and customized to provide the data to these systems in a format suitable for their use.

When selecting and customizing Sink operators, consider the following items when evaluating how to best interact with systems that consume the output of a Streams application:

- Can the downstream systems read data from files accessible from the Streams run time, such as in a distributed file system? If this is the case, then you can write output files to suitable directories on the Streams file system and the downstream systems can take the data from there. The FileSink operator that is contained in the Streams Standard Toolkit provides an immediate solution for outputting tuple data to a file.

  For high volume data, persistence to disk may be a performance limitation. Therefore, you may need to configure a high speed file system, or send the output using network communication.
- You can output a data stream over a TCP network connection, providing reliable communication of the stream contents to the recipient. The Streams TCPSink can be configured to be the server or client side of a TCP connection depending on the requirements of the receiving system.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000095

- You can output a data stream over a UDP network connection. Unlike TCP communication, UDP does not support reliable message transfer, so use UDP only in cases where some loss of that data is acceptable to gain additional speed of transfer. The Streams UDPSink can only be the client side of a UDP connection, so the Sink will stream data to a specified IP address and port over a UDP socket.
- Using an ODBC connection, you can insert tuple data into a database table. The ODBCAppend operator that is packaged with the Database Toolkit invokes a SQL INSERT to append data from a stream to a specified table.
- Using an HDFS sink, you can write data into a Hadoop file system for batch analysis by Hadoop that might require more data than can fit within a window.
- There might be another transfer protocol required to send messages over some specific messaging protocol. In this situation, there could be pre-written adapter operators available to support these protocols or you may need to produce a user-defined Sink operator.

You need to identify the format and structure of the data that is required by the external system:

- The required format could be either text based or binary. Having these options means a different configuration will need to be defined in the Streams Sink operator.
- You need to know the required structure of the data, as well as what data items and data types are required.

The nature of the output is determined by the purpose and requirements of the Streams application. In the following sections, we provide some examples of different types of output based on the purpose of the application.

Characterization of metadata

If an application is dealing with high volumes of data, one output may be a description that characterizes aspects of the data stream in a more concise representation. This more concise representation can then be used without having to know the details about the original streams.

For example, when receiving sensor readings from health sensors, you can summarize the individual readings into aggregate values such as the average value, the maximum and minimum, or some statistical measure of the data variation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000096

As another example, consider a traffic control system that takes feeds from road traffic cameras. This system will be receiving high volumes of binary video data from the camera and can then output a concise description of relevant events, such as the registration number of any identified vehicle, the location, and the time of the event. The original video data may be needed as an audit trail of the processing, but is not actually required for further analysis processing.

Opportunities

The fast processing that can be achieved with InfoSphere Streams enables you to identify and act on specific situations, thereby gaining a competitive advantage.

A prime example of opportunity identification is that of algorithmic trading in financial markets. Many areas of the financial services industry analyze larger and larger volumes of data to make their trading decisions. The value of a trading decision is highly time sensitive and so faster processing of this data can result in making trades at lower prices, which can result in greater profits.

Novel observations

Combining data from different sources and performing statistical analyses of the correlations allows Streams to uncover patterns and observations in the data streams that might not be evident.

An example of is using radio astronomy to find gamma ray signals among the data from thousands of telescope antennae.

Warnings and alerts

Streams applications can be used to monitor a stream of data and highlight when attributes in the data change state, particularly when an action of some sort should be taken to rectify an error or investigate suspicious activity. An output of warnings and alerts can be generated and fed into downstream systems to prompt action of a suitable nature.

This is the case in the health monitoring example, where Streams can be used to perform analysis of medical sensors, bringing significant changes in status to the attention of appropriate medical practitioners for further examination.

Controls and feedback

The Streams application may be able to directly impact the systems being observed. It can generate outputs that can then be fed as control signals to the automated machines controlling the monitored system, thus providing feedback to the system.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000097

One example is in the manufacturing use cases where you may control the conditions and ingredients of the production process to fine-tune the results, thereby improving the quality of the product.

Another example of data source control is in the use of security cameras, where you may control the orientation and zoom of a camera to observe an event of interest.

3.1.4 Existing analytics

Using the toolkits provided with InfoSphere Streams, you should be able to develop many useful applications without implementing custom logic in C++ or Java. However, in cases of complex analytic requirements, it might be necessary to extend the standard capabilities by implementing new operators and functions that invoke external software packages. In this case, the design needs to consider the nature of these external packages (for example, libraries) to ensure they are consistent with the requirements and well integrated.

For example, when dealing with unstructured data, you might use specialized analysis systems, such as video processing, audio processing, natural language processing, or language translation systems to analyze the incoming data stream and extract items of relevance. The relevant metadata can then be used as a data stream within the rest of the Streams application.

If the analysis system is packaged as a library, or remote API, you may wrap the API calls that interact with the package in SPL Primitive operators or native functions. Primitive operators may be implemented in either C++ or Java and invoked from your Streams application like any of the operators that are provided with the Streams product. Native functions enable you to package common C++ utilities so that they may be reused in your streams applications.

A related scenario is the case where you are migrating to Streams from a traditional platform. If suitable software packaging and APIs are available, the algorithms implemented using earlier code may be reused by invoking the APIs from Primitive operators and native functions.

You need to consider the implications of the new Streams application on any truly external systems. The Streams application will generate extra load on the system and you should confirm that this does not result in problems.

The design needs to determine what you should do if the external system fails or is unavailable. Compensation actions may be required, such as skipping a section of processing or alerting a human operator of the technical problem.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000098

3.1.5 Performance requirements

One of the main goals of Streams is to deliver high throughput and low latency computation. To achieve this task, the performance requirements of the system need to be well understood. In this section, we discuss a number of aspects of the performance requirements that should be addressed.

You should determine the expected volumes for each data source, and consider the following measures:

- What is the average throughput over a sustained period of time?
- What is the peak expected throughput in any short period of time?
- What growth in data volumes is anticipated in the future?

Knowing the average and peak throughput, for example, allows you to determine the power and quantity of physical hosts required by your target application. Streams has a low impact on the data path in exchange for the scale up and scale out features it provides as well as the application agility. When sizing an application for Streams, the best possible performance from a hand coded distributed application is a good starting point. Sizing of the physical environment is discussed further in 4.2.2, “Sizing the environment” on page 151.

There are cases when it is not possible to process all data by parallelizing or distributing because the data rates are too high or the required processing is too complex. In those cases, you need to determine an acceptable alternative when the volume of data from a data source exceeds what the system can handle. Some alternatives are:

- You can discard excess data. The Source and associated operators can be designed to act as a throttle, filtering the incoming data to a specific rate. It might also be possible to combine multiple records into aggregates (reducing the number of records processed downstream) or to discard excess tuples, either by a random process of sampling, by identifying tuples of lower potential value, or by shedding the most or least recently received tuple using a threadedPort.
- You can buffer excess data in the expectation that this is a peak load period and as the load declines, the system will be able to process that excess data. Note that using this approach will increase the average latency (time to process each individual entry), as latency will also include the time that a tuple is buffered and waiting to be processed. As such, this approach may be undesirable in an extreme low latency focused system, but many applications are not so sensitive and will still provide better latency than alternative approaches.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000099

- You can control the throughput of select flows by using the Throttle operator. The Throttle operator is provided in the Standard toolkit and is capable of pacing a stream given a rate specified in tuples per second.

Extreme performance requirements may lead you to segment the application to enable processing of smaller chunks, which may then be deployed to separate hosts and processed in parallel. The impact of sizing and extreme volumes on the application structure is discussed further in 4.5, “Application deployment capabilities” on page 179.

3.2 SPL design patterns

In this section, we present a set of SPL design pattern examples. These examples capture strategies and techniques that Streams developers have used to build SPL applications.

The patterns described here implement a specific capability. Typically, an application will be composed of numerous combinations of patterns, each of which provide a particular and common function. For example, applications may use a Data Parallel pattern (3.2.3, “Data Parallel” on page 90), where each parallel data flow executes an instance of the Enrich Stream pattern (3.2.6, “Enriching streams from a database” on page 107). By adapting these examples to your application, you may reuse standard solutions and thereby reduce the effort required to build common functions.

The design patterns described here showcase both general approaches for processing streaming data, as well as specific coding solutions. For example, the Simple Filter (3.2.1, “Reducing data using a simple filter” on page 82) and Simple Schema Mapper (3.2.2, “Reducing data using a simple schema mapper” on page 86) patterns explain the fundamentals of selecting tuples in a data stream, and a method for converting between stream schema. Almost all SPL applications perform these functions, and usually multiple times. A more elaborate type of filtering is given by the Outlier Detection pattern (3.2.5, “Outlier detection” on page 101). This pattern selects tuples containing data that is numerically distant from other tuples in a stream, and may be adapted to generate alerts for exceptional conditions. The Enrich Stream pattern (3.2.6, “Enriching streams from a database” on page 107) provides a detailed example of how to use operators in the Streams Database Toolkit to enrich a data stream with information from an external database. Other common code solutions, such as how to obtain runtime metrics using the standard Custom operator, or how to implement a Primitive operator that dynamically updates stream subscription properties are described by the Adapting to Observations (3.2.11, “Adapting to observations of runtime metrics” on page 134) and the Updating Import Stream

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000100

Subscriptions patterns (3.2.9, “Updating import stream subscriptions” on page 120), respectively.

In addition to distinguishing patterns by the level of detail described, the pattern examples implement different types of functions. The Processing Multiplexed Stream (3.2.8, “Processing multiplexed streams” on page 116), Simple Filter (3.2.1, “Reducing data using a simple filter” on page 82), and Simple Schema Mapper (3.2.2, “Reducing data using a simple schema mapper” on page 86) patterns perform common transformations on the streaming data. Techniques for improving latency and throughput are addressed by the Data Parallel (3.2.3, “Data Parallel” on page 90) and Data Pipeline (3.2.4, “Data Pipeline” on page 97) patterns. These examples explain methods for exploiting the distributed support provided by Streams to execute operations of the data flow concurrently, and thereby enhance performance. Another mechanism for adjusting application performance is exemplified by the Adapting to Observations (3.2.11, “Adapting to observations of runtime metrics” on page 134) and Tuple Pacer (3.2.7, “Tuple Pacer” on page 111) patterns, which use runtime metrics and processing latency, respectively, to change application behavior. The Updating Import Stream Subscription (3.2.9, “Updating import stream subscriptions” on page 120) and complementary Updating Export Stream Properties (3.2.10, “Updating export stream properties” on page 127) show how to establish stream connections at run time.

The documentation for each design pattern explains how to use and implement the pattern. Additionally, references are provided to related patterns and recommended reading. More specifically, the pattern documentation includes the following subsections:

- When to use: Describes scenarios where the example may be used.
- How to use: Identifies changes to be considered when adapting the pattern example for your application.
- Inputs, Outputs, and Parameters: A short description of each input stream, output stream, and parameter.
- Implementation: Shows how the pattern is implemented. This subsection may include SPL code, as well as implementations of the primitive operators, where relevant.
- Walkthrough: A step-by-step execution of the data flow for the implementation or example code.
- Example: Demonstrates how to invoke the pattern from an SPL application. Example input and output data may also be provided.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000101

Although the set of example designs presented here are a limited sample of the many patterns that will be used by Streams applications, they should provide a good start for developing your SPL code. Becoming familiar with these patterns will help you learn good practices for building applications for Streams, as well as design patterns that you can use and share.

3.2.1 Reducing data using a simple filter

When you are processing large volumes of data, much of the data may be considered to be irrelevant to your analysis. For example, you might only be interested in data that occurs infrequently, or is represented by a small fraction of the data. This design pattern describes a rudimentary filter for selecting tuples that contain a particular value for an attribute. The following sub-sections further describe the simple filter.

When to use

This pattern is useful for reducing the volume of data consumed by an SPL application. Consider applying this pattern when downstream operators require a subset of the streaming data, and filtering the steam yields a material reduction in latency or system resources consumed.

How to use

Insert one or more filters as far upstream in your application as possible. In general, when processing high volume streams, you should consider filtering the streams to reduce their volumes as soon as possible after they are ingested. This can reduce the total processing workload of the application, allowing a higher volume of throughput, with lower latency and potentially using less hardware. The example design pattern discussed below filters a stream of stock transaction data.

Input ports

Transactions: A full take of the stock transactions.

Output ports

SelectTransactions: Only the transactions containing the selected ticker symbol.

Parameters

$tickerSymbol: The selected ticker symbol (for example, IBM).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000102

Implementation

The example implementation of the filter selects stock transaction records for a particular ticker symbol. The operator takes as input a set of transaction records, and outputs the transaction records for a particular stock symbol. The stock symbol to select is specified as an input parameter (tickerSymbol) to the operator. This pattern is implemented as a Composite operator called TickerFilter. Given the input stream of transaction data, and a stock ticker symbol to select, the Composite operator invokes a Filter operator, which is provided in the standard toolkit for SPL.

The code for the Composite operator, sample.TickerFilter, is shown in Example 3-1.

Example 3-1 Composite operator TickerFilter

```
type
  TransactionRecord = rstring ticker, rstring date, rstring time,
                      rstring ttype, decimal64 price, decimal64 volume,
                      decimal64 bidprice, decimal64 bidsize,
                      decimal64 askprice, decimal64 asksize;

composite TickerFilter(output SelectTransactions; input Transactions) {
      param
           expression<rstring> $tickerSymbol;
      graph
           stream<TransactionRecord> SelectTransactions = Filter(Transactions) {
                param
                  filter : ticker == $tickerSymbol;
      }
}
```

Figure 3-3 shows the TickerFilter operator.

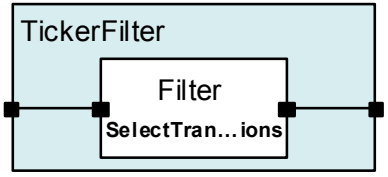


Figure 3-3 TickerFilter

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000103

Example

In Example 3-2, we show how you may invoke the TickerFilter example design pattern. Note that in this example, the ticker symbol to select is specified using a submission time parameter. The value of the tickerSymbol parameter is defined as a key-value pair when you launch the job.

Example 3-2 Invoking the TickerFilter

```
composite SimpleFilterMain {
     param
        // #1 The ticker symbol to select is defined by the tickerSymbol
        // submission time parameter
        expression<rstring> $tickerSymbolValue:
           getSubmissionTimeValue("tickerSymbol");
     graph
        // #2 Read the transaction data from a file
        stream<TransactionRecord> allTickers = FileSource() {
            param
                file : "SmallTradesAndQuotes.csv";
                format : csv;
        }
        // #3 Invoke the TickerFilter operator to filter out all but the
        // specified ticker symbol
        stream<TransactionRecord> selectedTicker = TickerFilter (allTickers) {
            param tickerSymbol : $tickerSymbolValue;
          }
        // #4 Write the filtered stream to a file.
        () as sinkPrefixSink = FileSink(selectedTicker)   {
            param
                file : "FilteredTradesAndQuotes.csv";
        }
}
```

Figure 3-4 shows the TickerFilter design pattern.

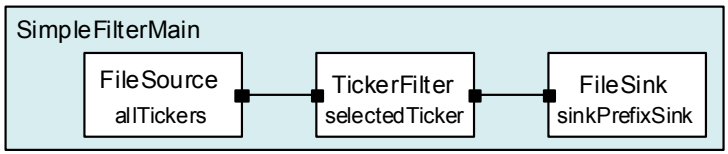


Figure 3-4 TickerFilter design pattern

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000104

Walkthrough

The following steps provide a walkthrough for Example 3-2 on page 84:

1. Obtain the stock symbol to select from the tickerSymbol submission time value. Assign the value of the symbol to the tickerSymbolValue variable.
2. Read the full take of transaction data from the SmallTradesAndQuotes input file, where the records are formatted as comma separated variables. The FileSource operator produces an allTickers stream.
3. For each tuple in the allTickers stream, invoke the TickerFilter operator to filter the input stream. If the transaction record references the input ticker symbol, the TickerFilter operator forwards the transaction record. Otherwise, the transaction record is not forwarded.
4. For each tuple in the selectedTicker stream, invoke the FileSink to output the selected transaction record. Given an input file `SmallTradesAndQuotes`, where the first few records are as follows:

   ```
   "ASA","27-DEC-2005","14:30:07.458","Quote",0,0,53.13,2,53.21,1
   "AVZ","27-DEC-2005","14:30:07.458","Quote",0,0,15.62,1,15.7,100
   "AYI","27-DEC-2005","14:30:07.458","Quote",0,0,32.38,1,32.49,13
   "BGG","27-DEC-2005","14:30:07.520","Quote",0,0,39.79,2,39.81,13
   "IBM","27-DEC-2005","14:30:07.521","Trade",83.48,74200,0,0,0,0
   "GSL","27-DEC-2005","14:30:07.582","Quote",0,0,43.65,1,43.71,1
   "JNS","27-DEC-2005","14:30:07.582","Quote",0,0,18.84,1,18.86,38
   "DLB","27-DEC-2005","14:30:07.585","Quote",0,0,16.84,23,16.95,1
   ```

   The SimpleFilterMain started with a tickerSymbol of IBM yields an output that includes the following records:

   ```
   "IBM","27-DEC-2005","14:30:07.521","Trade",83.48,74200,0,0,0,0
   "IBM","27-DEC-2005","14:30:08.117","Quote",0,0,80.43,18,83.49,10
   "IBM","27-DEC-2005","14:30:13.283","Quote",0,0,83.43,18,83.46,10
   "IBM","27-DEC-2005","14:30:13.518","Trade",83.45,200,0,0,0,0
   "IBM","27-DEC-2005","14:30:13.718","Quote",0,0,83.42,4,83.46,10
   "IBM","27-DEC-2005","14:30:14.731","Quote",0,0,83.42,4,83.45,1
   "IBM","27-DEC-2005","14:30:15.751","Quote",0,0,81.42,4,83.45,3
   ```

Note that the output file represents the results of filtering thousands of transaction records, where only the first one (that is, `"IBM","27-DEC-2005","14:30:07.521",...`) is listed in the extract of the input file previously shown.

Refer to 3.2.2, “Reducing data using a simple schema mapper” on page 86, where we provide a design pattern example that describes an alternative for reducing data volume.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000105

3.2.2 Reducing data using a simple schema mapper

A simple Streams application may be used to map between stream schema. Streams applications commonly map between schema for the purpose of selecting only relevant attributes, or translating between different schema. This design pattern addresses the former scenario and describes how to use SPL to reduce a schema.

When to use

This pattern is useful when the input data contains attributes that are not used by downstream operators. In this case, excluding unused attributes at the appropriate point in the data flow reduces the data volume. Related scenarios include combining attributes by mapping one or more input types to fewer output types.

How to use

This pattern may be used as a stand-alone utility, where it would ingest an input file and produce an output file with fewer attributes. More commonly, this pattern may be embedded in an application whenever you need to translate one schema to another.

Input ports

BigSchema: The stream whose schema is to be reduced.

Output ports

SmallSchema: The output stream with fewer schema attributes.

Parameters

SmallSchemaType: The schema of the output stream.

Implementation

This pattern is implemented as a Composite operator that uses a Functor to select a subset of the attributes in the input stream to produce an output stream with fewer attributes. Without additional parameterization, the Functor operator will match like named attributes between the input and output streams. The composite (SimpleSchemaReducer) is type generic, and assumes that the schema for the input and output streams are defined by the application that invokes this Composite operator. To provide ongoing feedback to the user, the Functor outputs a progress message for every 1000 input records processed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000106

The code for the Composite operator sample SimpleSchemaReducer is shown in Example 3-3.

Example 3-3 Composite operator SimpleSchemaReducer

```
composite SimpleSchemaReducer(output SmallSchema; input BigSchema) {
      param
        // The schema of the output stream.  Set this parameter to the name
        // of the SPL type that defines the output schema.
        type $SmallSchemaType;
      graph
        stream<TransactionRecordOutput> SmallSchema = Functor(BigSchema) {
            logic
               // Since this is implemented as a utility, provide the user with
               // periodic feedback on progress, and finally, the total number of
               // records processed.
               state : {
                 mutable int32 i = 0;
               }
               onTuple BigSchema : {
                   if ((++i % 1000) == 0) {
                      println("processing record#: " + (rstring)i);
                   }
               }
               onPunct BigSchema: {
                  println ("total records: " + (rstring)i);
               }
      }
```

Figure 3-5 shows the SimpleSchemaReducer.

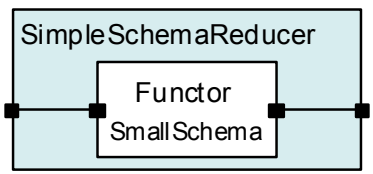


Figure 3-5 SimpleSchemaReducer

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000107

Walkthrough

The following steps provide a walkthrough for Example 3-3 on page 87.

1. It is assumed that the SimpleSchemaReducer Composite operator is being invoked from an application. In this case, the invoking application is feeding the SimpleSchemaReducer with stream BigSchema and is consuming the resulting SmallSchema stream. The SimpleSchemaReducer is parameterized with the TransactionRecordOutput type, which is the reduced schema used by the output stream.
2. For each tuple received on the BigSchema stream, the Functor produces a tuple with just those attributes of the input stream that are also defined in the schema of the output stream. Note that because no attributes are assigned in the output clause, the Functor implicitly assigns the input attributes to the output attributes with the same name.
3. For each 1000 tuples, the Functor provides a progress indicator to the user and reports the total number of tuples that have been processed.

Example

The example implementation of the schema mapper simplifies the schema used by the Vwap sample application. (The Vwap application is packaged with the InfoSphere Streams product in the `samples/spl/application/Vwap` directory). In particular, the TransactionDataSchemaMapper example composite reduces the 29 attributes (that is, columns) contained in the input stream to nine attributes that are used by one of the other design patterns. This particular example is implemented as a utility, which takes the TradesAndQuotes file as input (which contains 29 attributes) and produces SmallTradesAndQuotes, a smaller file that contains just nine attributes. The example code for the SimpleSchemaMapper is shown in Example 3-4. For more information, see “SPL schema” in *Streams SPL Language Reference*.

Example 3-4 shows an example TransactionDataSchemaMapper Composite operator.

Example 3-4 TransactionDataSchemaMapper

```
composite TransactionDataSchemaMapper
   type
   // Input record of transaction data with 29 attributes
   TransactionRecordInput
     = rstring ticker,rstring date, rstring time, int32 gmtOffset,
       rstring ttype, rstring exCntrbID, decimal64 price,
       decimal64 volume, decimal64 vwap, rstring buyerID,
       decimal64 bidprice, decimal64 bidsize, int32 numbuyers,
       rstring sellerID, decimal64 askprice, decimal64 asksize,
       int32 numsellers, rstring qualifiers, int32 seqno,
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000108

```
    rstring exchtime, decimal64 blockTrd, decimal64 floorTrd,
    decimal64 PEratio, decimal64 yield, decimal64 newprice,
    decimal64 newvol, int32 newseqno, decimal64 bidimpvol,
    decimal64 askimpcol, decimal64 impvol;

// Output record of transaction data with reduced schema, containing
// only 9 attributes
TransactionRecordOutput
  = rstring ticker, rstring date, rstring time,
    rstring ttype, decimal64 price, decimal64 volume,
    decimal64 bidprice, decimal64 bidsize, decimal64 askprice,
    decimal64 asksize;

graph
  // #1 Read full take of transaction data from comma separated input file
  stream<TransactionRecordInput> TradeQuote = FileSource() {
   param
     file : "TradesAndQuotes.csv";
     format : csv;
   }

   // #2 Map large record used by TradeQuote to reduced TransactionRecordOutput
   // schema.
   stream<TransactionRecordOutput> TradeQuoteSmall =
      SimpleSchemaReducer(TradeQuote) {
        param SmallSchemaType : TransactionRecordOutput;
   }

   // #3 Output reduced transaction to file, format as comma separated variables
   () as sinkPrefixSink = FileSink(TradeQuoteSmall)   {
         param
             file : "SmallTradesAndQuotes.csv";
             format : csv;
   }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000109

Figure 3-6 shows the TransactionDataSchemaMapper.

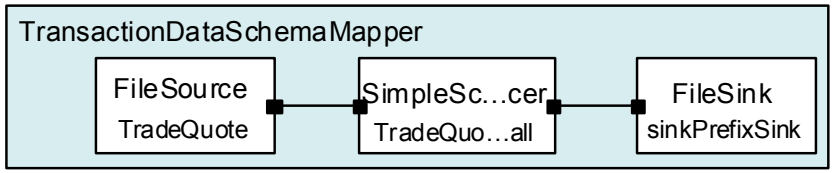


Figure 3-6 TransactionDataSchemaMapper

The `TradesAndQuotes.csv` input file contains records such as:

```
ASA,27-DEC-2005,14:30:07.458,-5,Quote,,,,,53.13,2,,,53.21,1,,R[PRC],,,,,,,,,,,
AVZ,27-DEC-2005,14:30:07.458,-5,Quote,,,,,15.62,1,,,15.7,100,,OQ[PRC],,,,,,,,,,,
AYI,27-DEC-2005,14:30:07.458,-5,Quote,,,,,32.38,1,,,32.49,13,,R[PRC],,,,,,,,,,,
BGG,27-DEC-2005,14:30:07.520,-5,Quote,,,,,39.79,2,,,39.81,13,,R[PRC],,,,,,,,,,,
IBM,27-DEC-2005,14:30:07.521,-5,Trade,,83.48,74200,83.4800,,,,,,,,Open|High|Low[GEN],46
19,14:30:07,1,,,,,,,,
GSL,27-DEC-2005,14:30:07.582,-5,Quote,,,,,43.65,1,,,43.71,1,,R[PRC],,,,,,,,,,,
JNS,27-DEC-2005,14:30:07.582,-5,Quote,,,,,18.84,1,,,18.86,38,,R[PRC],,,,,,,,,,,
DLB,27-DEC-2005,14:30:07.585,-5,Quote,,,,,16.84,23,,,16.95,1,,OQ[PRC],,,,,,,,,,,
```

The TransactionDataSchemaMapper utility outputs the same records to SmallTradeAndQuote, but with only nine of the 29 columns from the input file:

```
"ASA","27-DEC-2005","14:30:07.458","Quote",0,0,53.13,2,53.21,1
"AVZ","27-DEC-2005","14:30:07.458","Quote",0,0,15.62,1,15.7,100
"AYI","27-DEC-2005","14:30:07.458","Quote",0,0,32.38,1,32.49,13
"BGG","27-DEC-2005","14:30:07.520","Quote",0,0,39.79,2,39.81,13
"IBM","27-DEC-2005","14:30:07.521","Trade",83.48,74200,0,0,0,0
"GSL","27-DEC-2005","14:30:07.582","Quote",0,0,43.65,1,43.71,1
"JNS","27-DEC-2005","14:30:07.582","Quote",0,0,18.84,1,18.86,38
"DLB","27-DEC-2005","14:30:07.585","Quote",0,0,16.84,23,16.95,1
```

Refer to 3.2.1, “Reducing data using a simple filter” on page 82; that design pattern example describes an alternative for reducing data volume.

3.2.3 Data Parallel

This design pattern partitions a data stream into multiple separate (or parallel) data flows. The operators in each of the parallel data flows implement the same logic. This pattern assumes that each of the parallel data flows execute in a separate thread. When assigned to different processors, the parallel data flows may execute simultaneously, thereby yielding a potential improvement in both latency and throughput.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000110

To illustrate this pattern, we implement a simple application that aggregates revenue from a stream of food orders for a chain of restaurants.

When to use

Consider using this pattern when the inbound data stream can be partitioned into multiple outbound streams and subsequently recombined to yield a single outbound stream. To realize throughput improvements, the systems on which this application is run should have enough processors to execute the parallel data flows simultaneously.

Note that this pattern does not provide a mechanism for preserving the tuple order. As a result, this pattern should be used only when tuple order need not be preserved. Alternatively, the pattern may be extended to include an operator to add a sequence number prior to splitting and another operator to reorder the tuples subsequent to merging the parallel streams.

Additional considerations should be given to handling cases where ordering of tuples may be discontinuous as a result of tuples being dropped due to filtering operators or congestion on a stream.

How to use

To adapt the basic fan-out, fan-in structure of this pattern to your application, consider the following items:

- Identify a key in the inbound tuple attributes that may be used to partition the data streams.
- Count or estimate the unique values of this key. Depending on the data stream, you may want to use a separate operator for computing the distribution of key values.
- Count or estimate the number of processors available on the cluster on which the data parallel pattern will execute.
- Determine the number of parallel paths by determining how the operators in each path would be placed on processes.
- Define a mapping between keys and ports (output streams) that balances the volume of tuple data across the parallel paths.

Note that this pattern may be further abstracted by using a separate composite to implement the parallel data flows.

Input ports

Orders: Food orders from all restaurants.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000111

Output ports

RevenueTot: Snapshot of aggregate revenue for food orders.

Parameters

$snapshotPeriod: expression<int32> period (in seconds) for computing revenue snapshot.

Implementation

Implementations of this pattern are characterized by complementary Split and Union operators. The upstream Split operator demultiplexes the tuples in the inbound data stream by assigning each tuple to one of the operator's output ports. The assignment may be determined by a function on a tuple attribute that maps a value to the operator's output port number, or by a mapping table loaded by the operator. The Union operator, which is positioned downstream of the Split, multiplexes the partitioned data flow, yielding a single outbound stream.

Between the Split and Union operators are parallel sub-graphs that perform identical processing on the partitioned inbound data. Instead of a Union operator, patterns of this type may employ a Barrier or a Join to synchronize and combine the results of the parallel data flows.

This example pattern computes a periodic snapshot of revenue for a large restaurant chain. For each item on the menu, a tuple is created whenever a customer orders the product (burger, fries, drink, or dessert) from the menu. Because there are potentially thousands of franchises, and millions of orders per day, we can improve throughput by parallelizing the work of aggregating revenue. For this example, the ParallelAggregator composite operator contains the code that performs the parallel computation. Looking inside this composite, the Split demultiplexes the inbound tuples by mapping the value of the enumeration representing the menu item (that is, category) to an output port. Each of the streams output by the Split is consumed by a separate instance of a ComputeRevenueSnapshot operator. This operator is intended to serve as a placeholder for the computation performed on each parallel data flow. The Union operator multiplexes the tuples produced by each of the ComputeRevenueSnapshot operators and routes the tuples to an Aggregator, which computes a total revenue for each menu item over period specified by the $snapshotPeriod parameter.

The ComputeRevenueSnapshot consists of a single Aggregate operator, which sums the cost of the menu item at a frequency specified by the submission time $period parameter. For a commercial application, this composite may contain a large sub graph, each of which is executed in parallel.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000112

The code for ComputeRevenueSnapshot and the types used by this example are shown in Example 3-5.

Example 3-5 ComputeRevenueSnapshot

```
type OrderCategoryT = enum {burgers, fries, drinks, deserts, other};
type OrderRecordT = OrderCategoryT category, int64 cost;
type RevenueRecordT = int64 totalCost;

composite ComputeRevenueSnapshot(output RevenueSnapshot; input Orders) {
       param
         // The period, in seconds, for computing a revenue snapshot
         expression<int32> $period;
       graph
        // Put parallel data computation here.
        // Aggregate cost across orders received during specified period
        stream<RevenueRecordT> RevenueSnapshot = Aggregate(Orders) {
            window
                Orders: tumbling, time($period);
            output
                RevenueSnapshot : time = getTimestamp(),
                                  totalCost = Sum(cost);
        }
}
```

The ParallelAggregator Composite operator is shown in Example 3-6.

Example 3-6 ParallelAggregator Composite operator

```
composite ParallelAggregator(output RevenueTot; input Orders) {
       param
         // #1 Period for computing revenue snapshots.
         expression<int32> $snapshotPeriod;
       graph
          (stream<OrderRecordT> OrderBurgers;
           stream<OrderRecordT> OrderFries;
           stream<OrderRecordT> OrderDrinks;
           stream<OrderRecordT> OrderDeserts;
           stream<OrderRecordT> OrderOther) = Split(Orders) {
            param
             index : (int64)category;
        }
        stream<RevenueRecordT> RevenueBurgers = ComputeRevenueSnapshot(OrderBurgers)
           {
               param period : $snapshotPeriod;
        }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000113

```
        stream<RevenueRecordT> RevenueFries = ComputeRevenueSnapshot(OrderFries) {
               param period : $snapshotPeriod;
        }
        stream<RevenueRecordT> RevenueDrinks = ComputeRevenueSnapshot(OrderDrinks) {
               param period : $snapshotPeriod;
        }
        stream<RevenueRecordT> RevenueDeserts = ComputeRevenueSnapshot(OrderDeserts)
           {
               param period : $snapshotPeriod;
        }
        stream<RevenueRecordT> RevenueOther = ComputeRevenueSnapshot(OrderOther) {
               param period : $snapshotPeriod;
        }
        stream<RevenueRecordT> RevenueUnion = Union(RevenueBurgers;RevenueFries;
                                                    RevenueDrinks;RevenueDeserts;
                                                    RevenueOther) {
        }
        stream<RevenueRecordT> RevenueTot = Aggregate(RevenueUnion) {
            window
                RevenueUnion:  tumbling, time($snapshotPeriod);
            output
                RevenueTot: totalCost = Sum(totalCost);
        }
}
```

Figure 3-7 shows the ParallelAggregator.

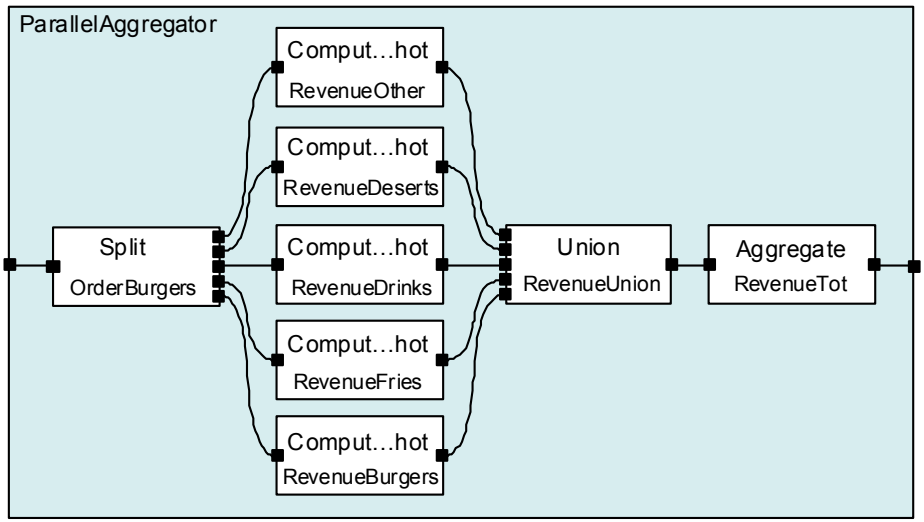


Figure 3-7 ParallelAggregator

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000114

Walkthrough

The following steps provide a walkthrough for Example 3-6 on page 93:

1. When invoked, the ParallelAggregator is initialized with the value of the $snapshotPeriod parameter. This parameter specifies the number of seconds to wait before computing another snapshot of the revenue.
2. For each tuple received by the ParallelAggregator, the value of the category attribute maps to the index of the output port on which the Split operator will route the tuple.
3. The tuple routed by the Split is passed to the instance of ComputeRevenueSnapshot operator that handles the category. The ComputeRevenueSnapshot is also initialized with the $snapshotPeriod.
4. The output from the ComputeRevenueSnapshot operator is re-multiplexed onto the RevenueUnion stream output by the Union operator.
5. A total for the revenue snapshots for each order category is computed periodically, as specified by the $snapshotPeriod parameter. Tuples containing the sum of revenue snapshots is output to the RevenueTot stream.

Example

The code listing shown in Example 3-7 shows a main composite that invokes the ParallelAggregator. The Beacon operator generates a value from 1 to 5 to represent the category of the food order (burgers, fries, and so on), and value from 1 - 6 to represent the cost. Each generated order is processed by the ParallelAggregator, and the results are exported using stream with the RevenueSnapshots ID.

Example 3-7 ParallelAggregator

```
composite ParallelDataFlowMain {
    graph
       // Use a beacon to generate sample orders for food.
            stream<OrderRecordT> Orders = Beacon() {
                   param period : 0.5;
                   output
                      Orders : category = (OrderCategoryT) (int32)(random() * 5.0),
                               cost = (int64)((random() * 5.0) + 1.0);
      }
      stream<RevenueRecordT> RevenueSnapshots = ParallelAggregator(Orders) {
             // Produce a snapshot for orders every 36 seconds.  In practice, the
             // period would be greater - hours or days.  36 seconds is useful for a
             // demo.
             param snapshotPeriod : 36;
      }

      () as eRevenueSnapshots = Export(RevenueSnapshots) {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000115

```
            param streamId : "RevenueSnapshots";
        }
}
```

Figure 3-8 shows a graph of the ParallelDataFlowMain.

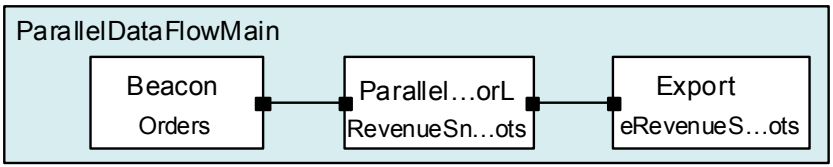


Figure 3-8 TransactionDataSchemaMapper

The following records show an extract of the input data generated by the Beacon in the main composite:

- other,1
- burgers,1
- burgers,2
- drinks,5
- other,4
- drinks,5
- other,5
- deserts,2
- drinks,2
- burgers,2
- drinks,4

A sampling of the tuples captured from the RevenueSnapshots stream is shown below. Each tuple represents a snapshot of the aggregated revenue.

```
(1305817600,400524000,0),832
(1305817636,402356000,0),860
(1305817672,403847000,0),797
(1305817708,405234000,0),810
(1305817744,406678000,0),905
(1305817780,408331000,0),841
```

The left column displays the SPL time stamp of the snapshot (in parenthesis), and the right most column contains the revenue collected during that period from all the chain restaurants. From left to right, the time stamp is encoded as seconds since the epoch, nanoseconds, and a machine identifier.

Refer to 3.2.4, “Data Pipeline” on page 97, where we have a design pattern that describes an alternative approach to executing Streams operators in parallel.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000116

3.2.4 Data Pipeline

This design pattern describes how to speed up processing by arranging a sequence of operators in a pipeline. The processing improvement is realized by positioning the operators in a linear sequence, or pipeline, so that each operator may execute in a separate thread. By assigning the operator's threads to different processors or hosts, consecutive tuples may be processed concurrently.

When to use

The Pipeline design pattern is useful for increasing throughput. The important considerations include:

- Tuples should not be dependent on one another, that is, the data values in one tuple should not be derived from another. This enables operations on different tuples to be executed concurrently.
- The target environment should provide multiple processors or hosts on which the multiple threads may be assigned.
- The distribution of execution time across the operators should be relatively even. Otherwise, if the latencies are not balanced, a slower operator might interfere with the smooth flow through the pipeline, and thereby limit the degree of concurrent execution.

How to use

The typical uses of the pipeline graph design pattern include the progressive (and concurrent) enrichment of tuple data as the tuple moves down the pipeline. To achieve the benefits of concurrency, the environment should enable the placement of operators on different processors or hosts. In the case that multiple operators are fused in the same processing element (PE), and are connected through threaded ports, ensure that the PE runs on a host with a sufficient number of processors to achieve an acceptable degree of concurrency across the fused operators. If the application consists of multiple PEs, consider how the operators should be assigned to different hosts. Typically, the pipeline is embedded in a Composite operator and provides one input and one output port. Because the pipeline is processing the series of tuples in order, an implementation will respect a FIFO discipline.

Input ports

In: The data stream input to the pipeline.

Output ports

Out: The data stream output by the pipeline.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000117

Implementation

The Pipeline pattern is implemented by a linear data flow of Functors. Note that the input port for each operator in the pattern is threaded. The threadedPort configuration declares that each operator executes on a separate thread. This thread dequeues the inbound tuple and invokes the operator's process method. This enables each operator, even if used in the same processing element, to execute concurrently. The configuration of the threaded port in this implementation specifies that the port will use a queue depth of 1 (the rightmost parameter), and will block when the queue is full (indicated by the Sys.Wait parameter). The first Functor in the pipeline uses the getPricefunction to retrieve the price for the order's part number. Given the price, the second operator verifies the customer has the credit required to purchase the part at the specified price. If the creditOK function returns true, implying that the customer has sufficient credit, this operator sets the orderOK attribute to true, and forwards the tuple. The last operator in the pipeline checks the inventory for the part number and sets the orderOK to false if the inventory is short. The implementations of the functions that get the price and inventory and run the credit check are not shown here. In practice, we would expect that various optimizations would be used, such as caching prices, and checking inventory only if the credit check succeeds. To simplify the example, we do not use these obvious optimizations.

The code for the Composite operator com.ibm.designpatterns.pipelineflow.Pipeline is shown in Example 3-8.

Example 3-8 Composite operator for Pipeline

```
composite Pipeline(output stream<OrderRecordT> Out; input stream<OrderRecordT> In) {
  graph
    // Given a part number, retrieve and add the price to the
    // order record
    stream<OrderRecordT> Order1 = Functor(In) {
      output
        Order1 : price = getPrice(In.partNum);
      config
        threadedPort : queue(In, Sys.Wait, 1);
    }
    // Given the price and quantity, check the customer's credit. Set orderOK to
    // false, if the customer credit is not sufficient for this order.
    stream<OrderRecordT> Order2 = Functor(Order1) {
      output
       Order2 : orderOK = creditOK(Order1.custNum, (float32)Order1.quantity *
         Order1.price);
      config
        threadedPort : queue(Order1, Sys.Wait, 1);
    }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000118

```
    // Given part number and quantity, verify that the inventory is sufficient to
    //satisfy the order.
    stream<OrderRecordT> Out = Functor(Order2) {
      output
        Out : orderOK = ((float32)Order2.quantity <= getInventory(Order2.partNum));
      config
        threadedPort : queue(Order2, Sys.Wait, 1);
    }
}
```

Figure 3-9 shows a graph of the Pipeline design pattern.

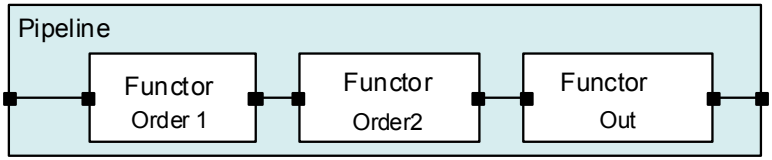


Figure 3-9 Pipeline design pattern

Walkthrough

The following steps provide a walkthrough for Example 3-8 on page 98.

1. Tuple-1, containing a customer order, is received by the first Functor in the pipeline on input stream In. After retrieving the price for the input part number, this Functor forwards the tuple on output stream RawOrderRecords.
2. The second Functor receives Tuple-1 on its threaded input port and runs a credit check to validate that the customer can afford the part.
3. Because the second Functor uses a threaded input port, the first Functor in the pipeline can process the Tuple-2 without waiting for the second Functor to complete. While the second Functor runs the credit check for the customer reference by Tuple-1, the first Functor obtains the price for the part number specified in Tuple-2.
4. When the second Functor completes the credit check for Tuple-1, it forwards the tuple on stream Order2 to the third and last Functor in the pipeline. While the third Functor verifies the inventory for Tuple-1 while running on a thread used to dequeue that tuple, the first and second operators in the pipeline may be concurrently processing the next tuples on the inbound stream (Tuple-3 and Tuple-2, respectively).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000119

Example

The following main composite invokes the operator that implements the pipeline. The pipeline is used to gather data to complete a customer order. The input consists of tuples that contain a part number and a quantity. The Pipeline operator enriches the input data with pricing information and sets the orderOK Boolean to true if the order can be processed (for example, the customer has the required credit).

A graph of the main composite is shown in Example 3-9. The Composite operator that produces the OrderRecords stream implements the Pipeline design pattern. This stream is exported by the main composite. It is assumed that another SPL application may consume this output and further process the customer order.

Example 3-9 Composite Pipeline main

```
// Represents a customer number.
type CustNumberT = enum {c9999, c9998, c9997, c9996};
// Represents the part that the customer is ordering
type PartNumberT = enum {p001, p002, p003, p004, p005};
// Represents an order for a part submitted by a customer.
type CustomerOrderT = CustNumberT custNum, PartNumberT partNum, int32 quantity;
// Represents an order processed by the fullment system.
// The orderOK boolean indicates whether the order has been verified
// (e.g. credit check and inventory are OK).
type OrderRecordT = CustomerOrderT,tuple<float32 price, boolean orderOK>;

composite PipelineMain {
  graph
    // The Beacon operator emulates a stream of customer orders
    // Each order is initialized with a part number and quantity.
    stream<CustomerOrderT> Orders = Beacon() {
      param period : 0.5;
      output Orders : custNum = (CustNumberT) ((int32)(random() * 3.0)),
                      partNum = (PartNumberT) ((int32)(random() * 4.0)),
                      quantity = (int32) (random() * 100.0);
      }
      // Transform a customer order to a OrderRecord and assign default
      // values for the fields required to process the order, such as the
      // current price and the orderOK boolean.
      stream<OrderRecordT> RawOrderRecords = Functor(Orders) {
        output RawOrderRecords : price = -1.0w, orderOK = false;
      }
      // The Pipeline gathers additional the information required to
      // process the orders.
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000120

```
      stream<OrderRecordT> OrderRecords = Pipeline(RawOrderRecords) {
      }
      // Export the order records for additional processing.
      stream<OrderRecordT> eOrderRecords = Export(OrderRecords) {
     }
}
```

Figure 3-10 shows a graph of the Pipeline design pattern.

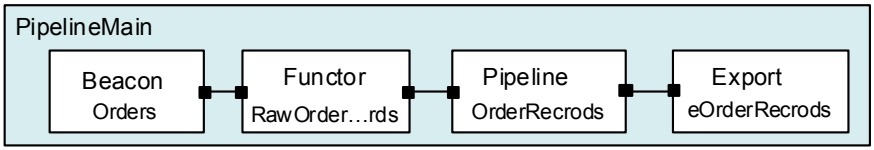


Figure 3-10 PipelineMain

Refer to 3.2.3, “Data Parallel” on page 90, which describes a different methodology for parallelizing execution.

3.2.5 Outlier detection

This design pattern illustrates an operator that monitors a data stream and detects “deviant” data. Real world scenarios that may employ outlier detection include monitors on health metrics where data outliers may imply a critical condition, or stock transactions where high or low prices may inform buy or sell decisions. In this pattern, the operator analyzes the input data to establish a normal range for the stream of data. Those tuples containing the values outside the normal range are considered outliers and are forwarded for further processing.

When to use

Use this pattern whenever the data values conform to a normal distribution, and a standard deviation provides an appropriate method for identifying outliers. This pattern is also useful in scenarios where you need to reduce the data volume. When used to filter tuples consumed by downstream operators, consider positioning this pattern upstream in the application.

How to use

Embed this pattern in Streams applications to generate alerts given exceptional data, or to identify outliers or statistically significant results.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000121

Input ports

AllData: The complete take of all tuples.

Output ports

Outliers: The tuples with relatively high or low data values.

Parameters

$numStandardDeviations: expression<float64>, which is the number of standard deviations used to identify outliers.

Implementation

Given an input stream consisting of tuples with non-negative data values, this operator forwards only those tuples that are considered outliers. Outliers are identified by computing the distance of a data value from a mean. The mean is aggregated over a window of tuples, and the distance is defined by a number of standard deviations. In the OutlierIdentifier operator, the mean and standard deviation are computed by the Aggregate operator, which uses a window of tuples. Using the mean and standard deviation provided by the Aggregrate, the Join operator forwards a tuple received on the AllData stream only if the data value is a specified number of standard deviations distant from the mean. The isOutlier function, invoked as part of the join condition, computes this distance and returns true if the data value is an outlier. This definition of the function and the code for Composite operator OutlierIdentifier is shown in Example 3-10.

Example 3-10 Composite operator OutlierIdentifier

```
composite OutlierIdentifier(output stream<OutlierStreamT> Outliers; input
   stream<DataStreamT> AllData){
       param expression<float64> $numStandardDeviations;
       type RunningStatsT = SourceIdT sourceId, float64 average, float64 standardDev;

       graph
         stream<RunningStatsT> RunningStats = Aggregate(AllData) {
            window
                AllData : partitioned, tumbling, count(100);
            param
                partitionBy: AllData.sourceId;
            output
                RunningStats :
                    sourceId = Any(AllData.sourceId),
                    average = Average(AllData.value),
                    standardDev = SampleStdDev(AllData.value);
      }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000122

```
    stream <OutlierStreamT> Outliers as O = Join (AllData as I1; RunningStats as
       I2) {
          window
              I1: sliding, count(0);
              I2: partitioned, sliding, count(1);
          param
              match : I1.sourceId == I2.sourceId
                      && isOutlier(I1.value, I2.average, I2.standardDev,
                         $numStandardDeviations);
              partitionByRHS : I2.sourceId;
          output
              O: sourceId = I1.sourceId,
                 value = I1.value,
                 average = I2.average,
                 standardDev = I2.standardDev;
    }
}
```

The Composite operator OutlierIdentifier is shown in Figure 3-11:

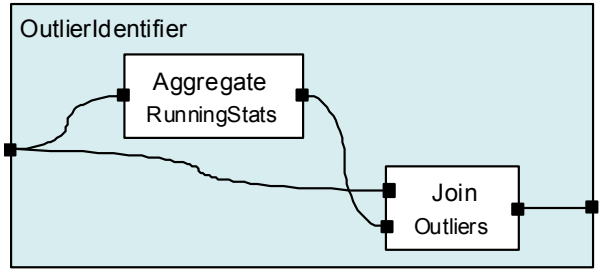


Figure 3-11 OutlierIdentifier

Walkthrough

The following steps provide a walkthrough for Example 3-10 on page 102:

1. Invoke the OutlierIdentifier Composite operator with the number of standard deviations used to identify outlying data values.
2. Forward each tuple received on the AllData stream to the Aggregrate and Join operators. The Aggregate operator derives the statistics used to identify outliers, and Join operator uses these statistics, if available, to evaluate each tuple on the AllData stream and forwards only those tuples that are statistical outliers. The following steps provide more detail about the operation of the Aggregrate and the Join.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000123

3. The Aggregate operator collects each tuple in a sub-window partitioned by the sourceId attribute. When the hundredth tuple is collected for a particular sourceId, the Aggregate computes the average and standard deviation for tuples in that window and forwards a tuple containing these statistics on the RunningStats stream to the Join operator.
4. The Join discards the tuples received on AllData stream, until a tuple containing aggregate statistics with a corresponding sourceId is received from the Aggregate operator on its right port, I2. This may be considered a initialization phase, or “warm-up” period, for the OutlierIdentifier.
5. When the Join operator receives the tuple containing the aggregate statistics on port I2, it caches the tuple in a sub-window partitioned by the sourceId.
6. The next tuple with the same sourceId received by the Join operator on the AllData stream is matched with corresponding tuple received on port I2. If the sourceIds match, and the isOutliers function returns true, the operator's join condition is satisfied, and a tuple representing an outlying data value is output by the Composite operator on the Outliers stream. The tuple cached in the sub-window for port I2 is not evicted until another tuple is received for that sourceId from the Aggregate operator.

Example

The following code provides an example of how to invoke the OutlierIdentifier Composite operator. The Beacon and Functor are used together to generate data tuples with a value field initialized to a random floating point number between 0 and 100, and a sourceId set to one of five enumerated values. To serve this example, the Functor is used to spread the data values and accentuate the top and bottom of the range, and thereby generate more outliers. The output from the Functor, the SpreadDatastream, is fed to the OutlierIdentifier. The resulting outliers, which consist of tuples where the data values exceeds two standard deviations, are captured in the `Outliers.Result.txt` file by one of the FileSinks. The other FileSink records the full take of data fed to the OutlierIdentifier in the `Outliers.AllData.txt` file.

The isOutlier function invoked by the OutlierIdentifier is shown in Example 3-11. This function returns true if a data value exceeds a distance from an average, and false otherwise.

Example 3-11 Outlier function

```
type SourceIdT = enum {A, B, C, D, E};
type DataStreamT = SourceIdT sourceId, float64 value;
type OutlierStreamT = DataStreamT, tuple<float64 average, float64 standardDev>

// Returns true if the input value is an outlier, and false otherwise.
// value - the input value
// avg - the average of the sample
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000124

```
// sigma - the standard deviation of the sample
// outlierFactor - the number of standard deviations
boolean isOutlier(float64 value, float64 avg, float64 sigma, float64 outlierFactor) {
  return (value > avg + outlierFactor * sigma) || (value < avg - outlierFactor *      sigma);
}

composite OutlierIdentifierMain {
       graph
         stream<DataStreamT> InputData = Beacon() {
              param period : 0.2;
              output InputData : value = (random()*10.0),
                                 sourceId = (SourceIdT)((int32)(random() * 5.0));
         }

         stream<DataStreamT> SpreadData = Functor(InputData) {
              logic state : {
                 mutable float64 extendedValue;
              }
              onTuple InputData : {
                       mutable float64 randomValue = random();
                       if (randomValue > 0.89) {
                              extendedValue = value * random() * 10.0;
                       } else if (randomValue < 0.19) {
                              extendedValue = value * random() * 0.1;
                       } else {
                              extendedValue = value;
                       }
              }
              output SpreadData : value = extendedValue;
     }

         stream<OutlierStreamT> Outliers = OutlierIdentifier(SpreadData) {
             param numStandardDeviations : 2;
         }

         () as sinkPrefixSink = FileSink(SpreadData)   {
            param
                file : "Outliers.AllData.txt";
      }

      () as sinkPrefixSink1 = FileSink(Outliers)   {
            param
                file : "Outliers.Result.txt";
      }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000125

Figure 3-12 shows a graph of OrderIdentifierMain.

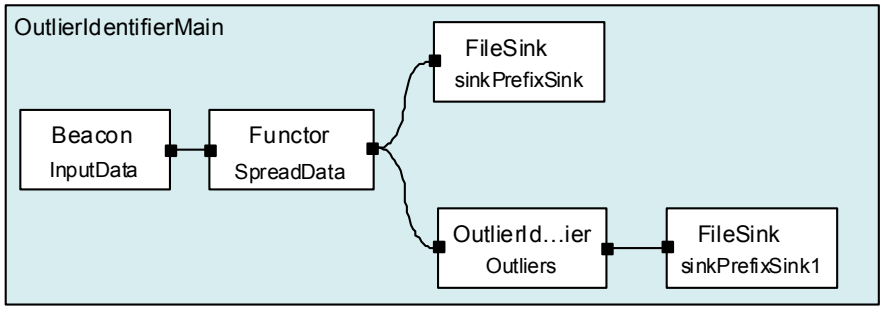


Figure 3-12 OrderIdentifierMain

The first few tuples saved in the `Outliers.AllData.txt` file is shown in the following list. The A - E values in the right column represents the sourceId. The left column contains the generated data value.

- D,0.192206682094148
- D,76.0289535039347
- B,0.107009603935798
- B,5.91479657683522
- B,9.07441698480397
- A,7.40057148071483
- E,9.08254098503395
- D,6.70154246501625
- D,30.2277856968631
- B,5.51431748084724

A sample output of outliers produced by OutlierIdentifers is shown in the following list. From left to right, the columns contain the sourceId, the data value that is an outlier, the average, and the standard deviation.

- A,23.7816749420277,5.35299225659099,8.21130862480932
- A,46.9148279364518,5.35299225659099,8.21130862480932
- B,38.7011607369096,6.29576116903712,9.75789555449789
- B,88.3713359529445,6.29576116903712,9.75789555449789
- B,34.0259439556577,6.29576116903712,9.75789555449789
- D,84.1985481073777,7.30886770639842,11.87863003809
- D,48.6204212472285,7.30886770639842,11.87863003809
- B,67.9634801996605,6.29576116903712,9.75789555449789
- A,34.9190637760897,5.35299225659099,8.21130862480932
- C,49.4983640114433,7.98486332462365,14.0166991654126

Refer to 3.2.1, “Reducing data using a simple filter” on page 82 for a simpler alternative for selecting or filtering streaming data.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000126

3.2.6 Enriching streams from a database

This design pattern demonstrates how you can enrich a stream with information drawn from a database. An ODBC Enrich operator from the Database Toolkit is used to populate stream data with attributes returned by a database query. This pattern shows how the Enrich operator is used to join with a database table, and thereby add reference information to the stream. This section also references technical guides and example applications that provide the detailed guidance required to configure the connection between the SPL operators and a database.

When to use

Consider using this pattern when you need to enrich the streaming data with relatively static reference information that is stored in a database. In general, to reduce load and latency, extensive reference information should be offloaded to a database, and introduced into your application only when needed. At that point in the data flow, invoke the ODBC Enrich operator to join the stream data with reference information returned from the database. To perform the enrichment, the stream schema must have an attribute, for example, an employee ID, that is shared by both the stream and table schema.

How to use

This example assumes several dependencies related to accessing a database from an SPL application. In particular:

- The Streams Database Toolkit (com.ibm.streams.db), which is packaged with the Streams product. The ODBCEnrich operator is contained in this toolkit.
- A database, with one or more tables containing reference information to be used to enrich the streaming data.
- An ODBC driver, such as UnixODBC. This driver is used by the operators in the Database Toolkit.
- A connections document that specifies access information used by the operators in the SPL Database Toolkit. This information is usually specified in the `etc/connection.xml` file located in your toolkit.

For more information about the Database Toolkit and configuration procedures, refer to:

- IBM InfoSphere Streams Version 2.0.0.2 Database Toolkit,, which can be found by referring to “Online resources” on page 421.
- The odbc_adapters_for_solid_db_at_work sample code is available on the Streams Exchange website, found at the following address:

  https://www.ibm.com/developerworks/mydeveloperworks/files/app?lang=en#/collection/ef9f8aa9-240f-4854-aae3-ef3b065791da

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000127

Input ports
None

Output ports
None

Parameters
$orderStatusStream: A string representing the name of the stream containing order status data. An order status stream is imported by this composite.

Implementation
The example implementation of this pattern enriches a stream containing order status for catalog items with the associated customer, product, and pricing information from a database. The enriched order status records are produced by this example and saved in a file. More specifically, this example works with the following sources of information:

- The OrderShipmentStatus stream, which is input by the Import operator. A tuple is received on this stream whenever the status of the order changes, as examples, order created, order shipped, and order received. This stream consists of OrderId, ShipmentStatus, and ShipmentDate attributes, where the OrderId is assumed to uniquely identify an order.
- A database table of OrderRecords that is accessed by the ODBCEnrich operator. A record is added to this table whenever a new order is created. The schema for this information is defined by the OrderRecord in the composite's `type` clause.

The ODBCEnrich operator enriches the OrderShipmentStatus with the corresponding OrderRecord, and outputs the combined tuples to the EnrichedOrderRecord stream. For each OrderShipmentStatus tuple, the ODBCEnrich operator queries the database specified in the `connection.xml` file for all records containing the same OrderId. The OrderRecord in the result set from the query is then appended to the input tuple to produce the enriched output tuple. The access information and query configuration are defined in the `connection.xml` document referenced by the ODBCEnrich operator. As you would expect, the query is specified as a SELECT statement, where columns correspond to the attributes in the OrderRecord and the WHERE clause filters on the OrderId. The relevant extract from the `connection.xml` is shown in the following example:

```
<query query="select CustomerName, CustomerId, ProductName, ProductId,
       Price from   Order where OrderId = ?" />
<parameters>
  <parameter name="orderId" type="int32" />
</parameters>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000128

The code for the Composite operator my.sample.Main is shown in Example 3-12.

Example 3-12 Code for my.sample.Main

```
composite Main {
  param
    // The OrderStatusStream is used to select streams containing order
    // status data exported by another application.
    // Setting this property at application submission time enables
    // you to configure the stream connection when
    // when the application is launched.
       expression<rstring> $orderStatusStream : getSubmissionTimeValue("OrderStatusStream");
    type
     OrderRecord = int32 OrderId, rstring CustomerName, int32 CustomerId, rstring
        ProductName,
                  int32 ProductId, rstring Price;
     graph
      // Receive tuple that describes change in an order's status - for example order       //
shipped
      stream<int32 OrderId,int32 ShipmentStatus, rstring ShippingDate>
         OrderShipmentStatus = Import() {
       param
         subscription : orderStream==$orderStatusStream;
       }

       // Using the OrderId, enrich the shipment status with customer and product
      // information from DB.
      stream<OrderRecord, tuple<int32 ShipmentStatus, rstring ShippingDate>>
         EnrichedOrderRecord = ODBCEnrich(OrderShipmentStatus) {
          param
            connectionDocument : "./etc/connections.xml" ;
            connection : "SenTestConnection" ;
            access : "readFromOrderTableForEnrich" ;
            orderId : OrderId ;
   }


       // Save enriched tuples in file
       () as SinkOp2 = FileSink(EnrichedOrderRecord) {
          param
            file : "odbc_enrich.result" ;
            format : csv ;
            flush : 1u ;
            writePunctuations : true ;
       }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000129

my.sample.Main is shown in Figure 3-13.

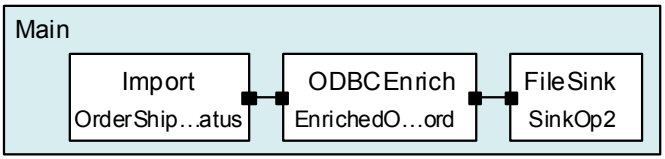


Figure 3-13 my.sample.Main

Walkthrough

The following steps provide a walkthrough for Example 3-12 on page 109:

1. When launching an application that includes this composite, provide a value for the OrderStatusStream submission time parameter. This parameter sets the subscription property used by the Import operator. The Import operator selects streams that are exported with the same name-value property.
2. Whenever a customer's order has been shipped, a tuple is received on the OrderShipmentStatus stream provided by the Import operator.
3. When the ODBCEnrich operator receives the order status tuple, the operator executes the SELECT statement shown above for all records with an OrderId matching the identically named field in the input tuple. The result-set, which contains the CustomerName, CustomerId, ProductName, ProductId, and Price, is combined with the attributes in the input tuple. This extended tuple is output to stream EnrichOrderRecord.
4. The tuples on the EnrichOrderRecord are saved by the FileSink operator to the `odbc_enrich.result` file.

Example

You have the following input tuple on stream OrderShipmentStatus:

```
34567, 1, Sat Feb 19 21:31:12 2011
```

You also have this order record in the OrderResults table in the database:

```
34567, 'John Price', 453234, 'iPod Nano', 53234223, '231.56
```

Given these two items, the ODBCEnrich operator enriches the order status tuple with the OrderResults record to produce:

```
34567, 'John Price', 453234, 'iPod Nano', 53234223, '231.56, 1, Sat Feb
19 21:31:12 2011
```

Refer to 3.2.3, “Data Parallel” on page 90, which describes an approach for running Streams operators in parallel, which may be useful if your application needs to run multiple instances of the enrichment pattern.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000130

3.2.7 Tuple Pacer

This design pattern is used to pace the flow of tuples within an application. This pattern balances the tuple rates between data producing and consuming operators, which may not be adjacent on a stream.

When to use

Consider using this pattern when you need to explicitly control the tuple processing flow between an upstream data producing operator and downstream data consumers. Note that InfoSphere Streams already provides a built-in queuing mechanism for input ports that calibrate the flow rate between producing and consuming operators on a stream. Streams uses a queue whenever an input port is connected to a stream that crosses PEs, or when the operators are fused in a single PE and a threaded port is configured for the consuming operator. For most applications, this built-in pacing mechanism provides the desired behavior, and the custom solution suggested by this pattern is not required.

However, in cases where it is necessary to regulate the tuple flow across multiple hops, or when custom pacing logic is required, the pattern suggested here might be useful. Suitable examples include scenarios where the application needs to manage the aggregrate flow across multiple operators, or when the pacing logic needs to incorporate parameters other than a queue length (as examples, CPU load, buffer consumption, and so on).

How to use

You may use this pattern in several ways:

- Encapsulate existing graphs between the Gate operator and Custom operator that generates the control tuple.
- Reuse the TuplePacer Composite operator in your own application by interposing this operator in a stream that needs to be paced.
- Update the Custom operators in the TuplePacer (see Example 3-13 on page 112) to invoke your application's business logic. For this example, the business logic is assumed to be contained in functions doWork_0 and doWork_1. These functions are invoked in the logic clause of the Custom operators. The implementation of these functions is not shown.

Input ports

InputStream: An input data stream.

Output ports

PacedOutputStream: A stream where the data is paced.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000131

Parameters

$maxQueueLength: expression<int32>, which is the size of the port queue used for inbound tuples.

Implementation

This pattern is built around a Gate operator, which forwards tuples, and a complementary downstream Custom operator, which emits a “control” tuple that is treated as an acknowledgment. In the diagram, the rightmost Custom operator produces the control tuple. The Custom operator in the middle is performing the analytic work on the inbound tuples. The Gate operator uses two input streams: data and control. This operator paces the flow of tuples on the input data stream by requiring a control tuple to acknowledge that previously sent tuples have been processed; an indicator that the downstream operators are ready to receive the next tuples. The Custom operator generates a control tuple containing the count of the number of inbound tuples that have been processed and are being acknowledged. Any number of operators that process the data may be interposed between the Gate and Custom operators.

The code for the Composite operator com.ibm.designpatterns.pacer.TuplePacer is shown in Example 3-13.

Example 3-13 Composite operator for TuplePacer

```
composite TuplePacer(output PacedOutputStream; input InputStream) {
       param expression<int32> $maxQueueLength;
       graph
         stream<StreamType> GatedData_0 = Gate(InputStream; Control) {
              // No more than 10 tuples may be sent prior to
              // receiving an acknowledgement
              param maxUnackedTupleCount : 10u;
                    // Indicates the number of tuples being acknowleged.
                    numTuplesToAck : Control.count;

         }

         stream<StreamType> GatedData_1 = Custom(GatedData_0) {
              logic onTuple GatedData_0:
              {
                // Process input tuple data. Implementation of the doWork_0 function
                // is not shown.
                doWork_0(GatedData_0);
                // Forward processed data
                submit(GatedData_0, GatedData_1);
              }
         }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000132

```
        (stream<StreamType> PacedOutputStream; stream<uint32 count> Control)
           = Custom(GatedData_1) {
             logic onTuple GatedData_1:
             {
               // Process input tuple data. Implementation of the doWork_1 function
               // is not shown.
               doWork_1(GatedData_1);
               // Forward processed data
               submit(GatedData_1, PacedOutputStream);
               // Acknowledge one tuple
               submit({count = 1u}, Control);
             }
             // Input port queue won't have more than 10 (the value of
             // maxUnackedTupleCount)tuples even though the max queue size specified
             // is larger than the $maxQueueLength
             config threadedPort : queue(GatedData_1, Sys.Wait, $maxQueueLength);
        }
}
```

The Composite operator TupldPacer is shown in Figure 3-14.

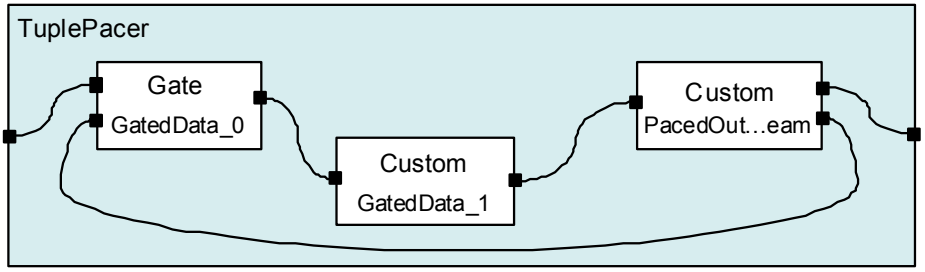


Figure 3-14 TuplePacer

Walkthrough

The following steps describe a flow through the TuplePacer (Example 3-13 on page 112) when the data consuming operators are keeping up with the rate of inbound data tuples:

1. The Gate operator receives the first data tuple on the InputStream stream.
2. Because the Gate operator is configured to continue forwarding tuples until more than 10 tuples are not acknowledged, the first tuple is forwarded on output stream GatedData_0. The count of unacknowledged tuples maintained by the Gate operator is incremented, and now equals 1.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000133

3. The first tuple is received by the Custom operator that outputs stream GatedData_1. This operator invokes function doWork_0 to process the tuple, and forwards the results on GatedData_1.
4. The data tuple is next received by the Custom operator that implements the control logic for producing acknowledgements. This operator forwards the data tuples on the PacedOutputStream. A control tuple that acknowledges that a data tuple has completed processing is sent on the output stream named Control. For each data tuple received, this Custom operator invokes function doWork_1 to process the data tuple. After the tuple has been processed, a control tuple is sent with a count set to 1 to indicate that a single data tuple is being acknowledged.
5. The Gate operator receives the acknowledgement on the input Control stream. Given this acknowledgement, the Gate operator decrements the count of unacknowledged tuples by 1 (the value of the count attribute). Because the value of the counter is 1, as set in step 1 on page 113, the counter of unacknowledged tuples is reset to 0. The next inbound tuple will be immediately forwarded.

In the case where the processing time of the data consuming operators exceeds the inter-arrival rate of the inbound data, tuples will start to build up on the queue associated with the threaded port of the GatedData_1custom operator. When this happens, the flow rate is calibrated as follows:

1. The Gated operator continues to forward data tuples on the GatedData_0 stream until the 10 tuples are sent and no acknowledgement has been received. At this point, the Gate suspends forwarding the inbound tuples. The inbound tuples will back up on the queue of the Gate operator's threaded port.
2. Whenever the GatedData_1 customer operator completes processing an inbound tuple, it will generate an acknowledgment for that tuple on the Control stream.
3. When the Gate operator receives the acknowledgment, it will decrement the unacknowledged tuple count from 10 to 9. Because the unacknowledged tuple count is less than 10, the Gate operator will dequeue and forward a tuple from the InputStream and increment the unacknowledged tuple count to 10. Any tuples now received on the InputStream will be queued until the next acknowledgement is received.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000134

Example

Example 3-14 illustrates how the TuplePacer operator may be invoked. When installing this pattern in your own application, the imported stream may be replaced by the output of a sub-graph that you want to pace. Similarly, the Export operator may be replaced by the SPL code that consumed the paced stream.

Example 3-14 Invoking the TuplePacer operator

```
composite TuplePacerMain {
   graph
     stream<StreamType> InputStream = Import() {
          param subscription : InboundTuples;
     }
     // No more than 1000 tuples may be queued awaiting for processing.
     stream<StreamType> PacedStream = TuplePacer(InputStream){
          param maxQueueLength : 1000;
     }
     stream<StreamType> OutputStream = Export(PacedStream) {
          param streamId : PacedTuples;
     }
}
```

Figure 3-15 shows TuplePacerMain.

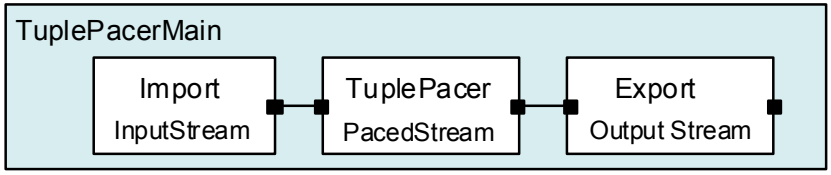


Figure 3-15 TuplePacerMain

Refer to the following items:

- Section 3.2.11, “Adapting to observations of runtime metrics” on page 134. This design pattern addresses a related scenario where the application adjust its workload dynamically.
- The spl.utility.Gate operator in the InfoSphere Streams SPL Standard Toolkit Reference. For the Toolkit location, refer to “Online resources” on page 421.
- The spl.utility.Throttle operator in the InfoSphere Streams SPL Standard Toolkit Reference. For the Toolkit location, refer to “Online resources” on page 421.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000135

3.2.8 Processing multiplexed streams

This design pattern is about processing multiplexed streams. It is often the case that a stream contains multiple logical sub-streams in it, where each logical stream requires processing and a state independent of the others. A typical example is computing the volume weighted average price (VWAP) for each stock ticker in a financial trading application. In this example, the tuples corresponding to a particular stock ticker constitutes a logical stream and the VWAP is computed over this logical stream independent of the others.

When to use

This pattern is useful when the application contains streams that carry multiplexed sub-streams in them and there is a requirement to process the sub-streams independently of each other. A stream is often multiplexed by identifying one or more attributes as the partitioning key, which is often referred to as “partition-by-attributes” in SPL.

How to use

To use this pattern, first a set of attributes on a stream are identified as the partition-by-attributes. Then the stream is processed by taking it through one or more partitioned-stateful operators that are capable of processing multiplexed streams, or through stateless operators. In the SPL Standard Toolkit, partitioned-stateful operators consistently provide a partitionBy attribute that can be used to configure them to perform processing on multiplexed streams such that the processing is performed and state is maintained, independently for each sub-stream. For partitioned-stateful Custom operators in SPL and primitive operators developed by third parties, use a partitionBy parameter. A common way of implementing this setup is to organize the state kept by the operator inside a map, and use the partition-by-attribute of the incoming tuple to locate the subset of the state that corresponds to the sub-stream to which the input tuple belongs, and operate on the state.

Implementation

Here we provide an application example, RevenueBySector, where we apply this pattern using an SPL Standard Toolkit operator, as well as a Custom SPL operator that we develop. The goal of the application is to read a list of data items from a file, where the data items contain information about revenues of companies. Each company belongs to an industry. We identify the industry as the partition-by-attribute. The goal is to compute the total revenue per industry and write the result to a file, where we want one result file per-industry, containing that industry's total revenue.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000136

The MultiFileWriter is a Composite operator that is implemented in SPL using a Custom operator. It creates a file for each unique session attribute value and writes all tuples with the same session ID to the corresponding file. Internally, it uses an SPL map to maintain the partition by attribute to file handle mapping. For each tuple it receives, it uses the session attribute to locate the appropriate file handle. If the file handle is not there, it creates it (the session is seen for the first time). On receiving the final punctuation, it flushes the files. The code for the MultiFileWriter follows the listing of the RevenueBySector composite.

The code for the Composite operator com.ibm.designpatterns.demultiplexstream.RevenueBySector is shown in Example 3-15.

Example 3-15 Composite operator for RevenueBySector

```
composite RevenueBySector {
  type
    Revenue = tuple<rstring sector, rstring company, float64 revenue>;
    SectorRevenue = tuple<rstring sector, float64 revenue>;
  graph
    stream<Revenue> Revenues = FileSource() {
      param
        file : "input";
        format : csv;
    }
    stream<SectorRevenue> TotalRevenuesBySector = Aggregate(Revenues) {
      window
        Revenues: partitioned, tumbling, punct();
      param
        partitionBy: sector;
      output
        TotalRevenuesBySector: revenue = Sum(revenue);
    }
    () as Sink = MultiFileWriter(TotalRevenuesBySector) {
      param
        partitionBy: sector;
        fileNamePrefix: "output_";
    }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000137

The Composite operator RevenueBySector is shown in Figure 3-16.

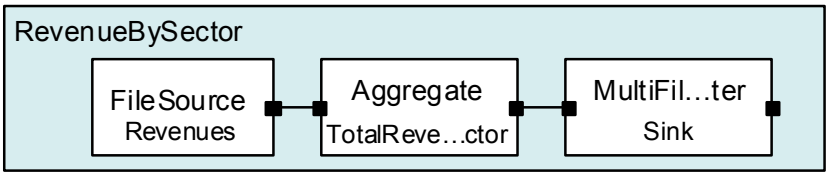


Figure 3-16 RevenueBySector

The code and graph diagram for the MultiFileWriter Composite operator is shown in Example 3-16.

Example 3-16 Composite MultiFileWriter

```
composite MultiFileWriter(input S)
{
  param
    expression $partitionBy;
    expression<rstring> $fileNamePrefix;
  graph
    () as Sink = Custom(S) {
      logic
        state :
          mutable map<rstring, uint64> files = {};
        onTuple S: {
          rstring key = (rstring) $partitionBy;
          rstring file = $fileNamePrefix + key;
          if (!(key in files)) {
            mutable int32 err = 0ul;
            uint64 fd = fopen(file, "w", err);
            if (err!=0)
              log(Sys.error, "Cannot open file '"+file+"' for writing");
            else
              files[key] = fd;
          }
          if(key in files) {
            mutable int32 err = 0ul;
            fwriteTxt(S, files[key], err);
            if (err!=0)
              log(Sys.error, "Cannot write to file '"+file+"'");
            fwriteString("\n", files[key], err);
          }
        }
        onPunct S: {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000138

```
            mutable int32 err = 0;
            if (currentPunct()==Sys.FinalMarker)
              for (rstring key in files)
                fclose(files[key], err);
        }
    }
}
```

The Composite operator MultiFileWriter is shown in Figure 3-17.

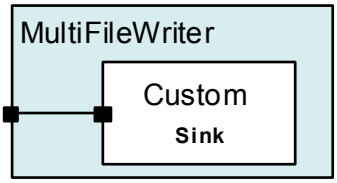


Figure 3-17 MultiFileWriter

Walkthrough

The following steps provide a walkthrough for Example 3-15 on page 117 and Example 3-16 on page 118:

1. Revenue is a tuple type that describes the revenue of a company, including the company's sector.
2. SectorRevenue is a tuple type that describes the revenue of a sector.
3. The FileSource operator, shown in Example 3-15 on page 117, is used to create a stream out of the input file `input`.
4. The results of the FileSource operator is fed into an Aggregate operator, which is configured with a partitionBy parameter where the partition by attribute is identified as the sector. The aggregation has a punctuation based window, which means that the entire input stream will be aggregated (the entire file). In a more practical variation of this application, the window can be count or time based.
5. The results from the Aggregate operator, which will include one tuple per session, are fed into the MultiFileWriter operator. Again, the partitionBy parameter is specified and the session attribute is used as its value.
6. For each sector, the MultiFileWriter, shown in Example 3-16 on page 118, creates a separate output file to store the total revenue for the sector. The contents of the tuple containing the aggregated results for the sector are written to the output file.
7. The output files are closed when a final punctuation is received.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000139

Example

Given the following input:

```
technology, IBM, 100
finance, "American Express", 50
aerospace, Boeing, 150
technology, HP, 100
aerospace, "Lockheed Martin", 150
technology, MS, 100
finance, "Goldman Sachs", 50
```

The program generates the following output:

```
output_aerospace
================
{sector="aerospace",revenue=300}

output_finance
==============
{sector="finance",revenue=100}

output_technology
=================
{sector="technology",revenue=300}
```

Refer to 3.2.3, “Data Parallel” on page 90 for more information. This design pattern may be used to process the logical sub streams in parallel.

3.2.9 Updating import stream subscriptions

This design pattern shows how to dynamically update the stream subscription specified as part of an Import operator. The Import operator has a subscription parameter that specifies an expression to be evaluated against the stream properties of exported streams. Based on this evaluation performed at run time, new stream connections are established. The goal of this pattern is to show how this subscription expression can be updated at run time.

When to use

A common use case is to update the subscription expression based on the changing needs of the downstream processing. As an example, consider a video monitoring application that watches who enters and leaves a building. The subset of video source streams that are of interest to the application can change based on the analysis performed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000140

For example, if the monitoring application detects high activity on the camera feed that provides a view of the front door, it might decide to start processing feeds from additional cameras that provide close-up views of the door.

How to use

For a realistic implementation of this pattern, the ImportController Composite operator, described in the following text, would ingest a data stream from one of the cameras monitoring the front door. The logic in this operator would be updated to generate subscription properties based on the level of activity reported by the camera. Additionally, the Importer and Exporter operators would likely be implemented in separate SPL applications. The stream connections from the export to the import operators would be established when the applications are deployed to the run time.

Implementation

Import operators are pseudo-operators in SPL. The import subscriptions are associated with the input ports of the downstream operators that consume the stream produced by the Import operator. Those operators can update the import subscription associated with their input port at run time. At the time of the writing of this book, these APIs are only available to Primitive operators, but future versions of Streams will make them available to Custom operators as well.

We showcase the implementation of this pattern using a sample application that uses dynamic subscription updates to switch between receiving one of two exported streams. As part of this application, we will make use of a Primitive operator called DynamicImporter. This operator is connected to the output of the Import operator and forwards the imported stream. At the same time, it accepts subscription expressions as tuples on a second (control) port and uses the runtime APIs to update the subscription expression associated with the first input port.

In addition to the Importer that updates the stream subscription, this section includes the code and graph diagrams for the Exporter, SourceGen, and ImportController operators. The Exporter and SourceGen operators collaborate to generate an exported stream that is consumed by the Import operator in the Importer. The ImportController serves this example by periodically producing a control tuple containing the new subscription expression. The top of the code example, shown in Example 3-17, is a Main that invokes both the Exporter and Importer.

Example 3-17 Composite using the Exporter and Importer

```
// A main composite that invokes both the stream exporter and importer.
composite Main {
  graph
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000141

```
    () as E = Exporter() {}
    () as I = Importer() {}
}

// This operator exports two streams
composite Exporter {
  graph
    // We export a stream with an export property of name="A"
    stream<int32 id, rstring name> DataA = SourceGen() {
      param
        name : "A";
    }
    () as ExporterA = Export(DataA) {
      param
        properties : { name = "A" };
    }
    // We export a stream with an export property of name="B"
    stream<int32 id, rstring name> DataB = SourceGen() {
      param
        name : "B";
    }
    () as ExporterB = Export(DataB) {
      param
        properties : { name = "B" };
    }
}

// This operator is used as a workload generator
composite SourceGen(output Out) {
  param
    expression<rstring> $name;
  graph
    stream<int32 id> Src = Beacon() {
      logic
        state: mutable int32 cnt=0;
      param
        period: 0.25;
      output
        Src: id = cnt++;
    }
    stream<int32 id, rstring name> Out = Functor(Src) {
      output
        Out: name = $name+"_"+(rstring)id;
    }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000142

```
// This operator imports an alternating stream
composite Importer {
  graph
    stream<int32 id, rstring name> Imported = Import() {
      param
        subscription : name=="B";
    }
    stream<Imported> DynImported = DynamicImporter(Imported; Control)
       {}
    stream<rstring subscription> Control = ImportController() {}
    () as Sink = FileSink(DynImported) {
      param
        file: "results";
        flush: 1u;
    }
}

// This operator periodically produces a new subscription expression
composite ImportController(output Out) {
  graph
    stream<int8 dummy> Beat = Beacon() {
      param
        iterations: 1u;
    }
    stream<rstring subscription> Out = Custom(Beat) {
      logic
        onTuple Beat: {
          while(!isShutdown()) {
            block(1.0);
            submit({subscription="name==\"A\""}, Out);
            block(1.0);
            submit({subscription="name==\"B\""}, Out);
          }
        }
    }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000143

Figure 3-18 shows Main composite for importing and exporting.

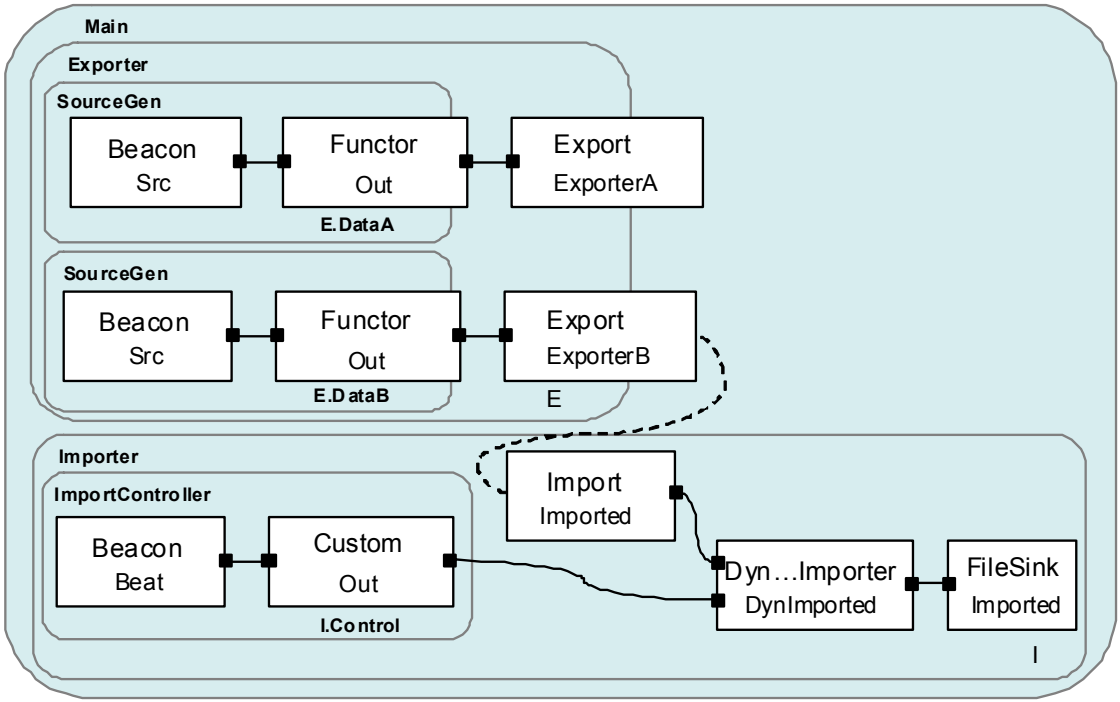


Figure 3-18 The expanded Main composite for importing and exporting

Where:

- The pseudo Import and Export operators are shown in the graph. The connection between the Export and Import operators enables the stream data to flow from the Exporter to Importer Composite operators.
- Note that stream data flows from the composite Export operator to the Import operator (dashed line). Also note that they are pseudo operators and as such do not terminate with an operator port as do the other streams operators.
- Composite E is exporting streams DataA and DataB, with property values name = “A” and name = “B”, respectively.
- Composite I dynamically toggles between importing streams DataA and DataB. In the example code above, the Import operator in composite I is initialized to import DataB.
- The name of the DynamicImporter has been abbreviated to Dyn...Importer.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000144

Walkthrough

The Main Composite invokes the Importer and the Exporter Composites. These are two independent flows. In a real world application of this pattern, it is more likely that the Importer and the Exporter are part of different applications.

The following steps provide a walkthrough for Figure 3-18 on page 124:

1. The Exporter Composite operator creates and exports two streams. The first stream is exported with a name property of value “A” and the second stream is exported with a name property of value “B”. Both of the streams are generated using the SourceGen Composite operator.
2. The SourceGen Composite operator is a simple workload generator that uses Beacon and Functor operators to generate periodic tuples with increasing numeric identifiers and a string attribute that has a prefix followed by the same numeric ID. The prefix is a parameter of the SourceGen operator.
3. The Importer Composite operator imports the stream with the name property value of “A” using an Import operator. The Import operator is connected to a DynamicImporter Primitive operator. This Primitive operator controls the import subscription of the Import operator. It has a control input port that receives tuples that contain subscription expressions as string attributes. Whenever a new subscription expression is received, this operator will update the Import operator's subscription expression. The control input port of the DynamicImporter operator is fed by the output of a ImporterController operator, which we describe next. Finally, the results from the DynamicImporter are sent to a FileSink operator, which writes the results to a file.
4. The ImportControllerComposite operator uses a Custom operator to periodically generate a tuple that contains an alternating subscription expression string, one of anaemia” or nameable”.

The C++ code that implements the DynamicImporter operator is shown in Example 3-18.

Example 3-18 DynamicImporter operator

```
void MY_OPERATOR::process(Tuple const & tuple, uint32_t port)
{
    if(port==1) { // control tuple, update the subscription expression
        for (ConstTupleIterator it=tuple.getBeginIterator();
             it!=tuple.getEndIterator(); ++it) {
            ConstValueHandle handle = (*it).getValue();
            if(handle.getMetaType()==SPL::Meta::Type::RSTRING) {
                rstring const & exp = handle;
                try {
                    getInputPortAt(0).setSubscriptionExpression(
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000145

```
                        *SubscriptionExpression::fromString(exp));
                    break;
                } catch(...) {}
            }
        }
    } else { // data tuple, just forward
        submit(tuple, 0);
    }
}
```

Example

As it can be seen from the DynamicImporter code shown in Example 3-18 on page 125, the data tuples are forwarded from the first input port to the output port, and the control tuples from the second input port are used to update the subscription expression associated with the first input port.

A fragment of results from running this application is shown in the following list. Note that the stream subscription toggles every three or four tuples, which is expected, as Composite E generates four tuples per second on each output steam, and the ImportController emits a control message that switches the stream subscription once per second.

```
613,"A_613"
614,"A_614"
615,"A_615"
616,"A_616"
617,"B_617"
618,"B_618"
619,"B_619"
621,"A_621"
622,"A_622"
623,"A_623"
624,"B_624"
625,"B_625"
626,"B_626"
627,"B_627"
```

Refer to 3.2.10, “Updating export stream properties” on page 127 for more information. This pattern shows how to write a Primitive operator that updates a stream's export properties at run time.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000146

3.2.10 Updating export stream properties

This design pattern shows how to dynamically update the stream properties specified as part of an Export operator. The Export operator has a properties parameter that specifies a list of properties to be evaluated against the stream subscriptions of imported streams. Based on this evaluation performed at run time, new stream connections are established. The goal of this pattern is to show how these properties can be updated at run time.

When to use

A common use case is to update the export properties based on the changing characteristics of the exported stream. As an example, consider one or more applications performing analysis on different news feeds. Based on the analysis performed by an application, the resulting news stream can be associated with a priority property. This property can change depending on the current contents of the associated feed. An importing application can specify a stream subscription that only receives news feeds with high priority. The exporting applications, by updating their priority property at run time, enable the downstream applications to import only the relevant news feeds.

How to use

The example pattern shown here employs several simplifications that would likely be changed for commercial implementations. In particular:

- Both the Exporter and Importer operators are invoked by the same main Composite. A more useful implementation would be to invoke each of these Composite operators from separate main Composites. As a result, these applications would be able to be submitted to the Streams run time separately. A Stream connection between the Export and Import operators contained in these Composites would be established dynamically, provided that the subscription expression matches the export properties.
- The ExportController Composite operator would be replaced or updated with an operator that changed the export priority based on an inspection of the news feed data. The implementation used in this example periodically updates the priority, which is appropriate for demonstration only.

Implementation

Export operators are pseudo-operators in SPL. The export properties are associated with the output ports of the upstream operators that produce the stream consumed by the Export operator. Those operators can update the export properties associated with their output port at run time. At the time of the writing of this book, these APIs are only available to Primitive operators, but future versions of Streams will make them available to Custom operators as well.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000147

We showcase the implementation of this pattern using a sample application that uses dynamic stream property updates to switch between sending data to one of two downstream importers. As part of this application, we make use of a Primitive operator called DynamicExporter. This operator receives the stream to be exported and forwards it to the Export operator. At the same time, it accepts new stream properties as tuples on a second (control) port and uses the runtime APIs to update the stream properties associated with its output port.

We provide the code for the application in Example 3-19.

Example 3-19 DynamicExporter

```
composite Main {
  graph
    () as E = Exporter() {}
    () as I = Importer() {}
}

composite Exporter {
  graph
    stream<int32 id, rstring time> Exported = SourceGen() {}
    stream<Exported> DynExported = DynamicExporter(Exported; Control) {}
    stream<rstring priority, int64 level> Control = ExportController() {}
    () as Exporter = Export(DynExported) {
        param
            properties : {priority="low", level=4l};
    }
}

composite SourceGen(output Out) {
  graph
    stream<int32 id, rstring time> Out = Beacon() {
      logic
        state : mutable int32 cnt=0;
        param
          period : 0.25;
        output
          Out: id = cnt++, time=ctime(getTimestamp());
    }
}

composite ExportController(output Out) {
  graph
    stream<int8 dummy> Beat = Beacon() {
      param
        iterations: 1u;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000148

```
    }
    stream<rstring priority, int64 level> Out = Custom(Beat) {
      logic
        onTuple Beat: {
          while(!isShutdown()) {
            block(1.0);
            submit({priority="high", level=3l}, Out);
            block(1.0);
            submit({priority="low", level=4l}, Out);
          }
        }
    }
}

composite Importer {
  graph
    stream<int32 id, rstring time> ImportedA = Import() {
      param
        subscription : priority=="high" && level==3l;
    }
    () as SinkA = FileSink(ImportedA) {
      param
        file: "results.A";
        flush: 1u;
    }
    stream<int32 id, rstring time> ImportedB = Import() {
      param
        subscription : priority=="low" && level==4l;
    }
    () as SinkB = FileSink(ImportedB) {
      param
        file: "results.B";
        flush: 1u;
    }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000149

We provide a diagram of the application in Figure 3-19.

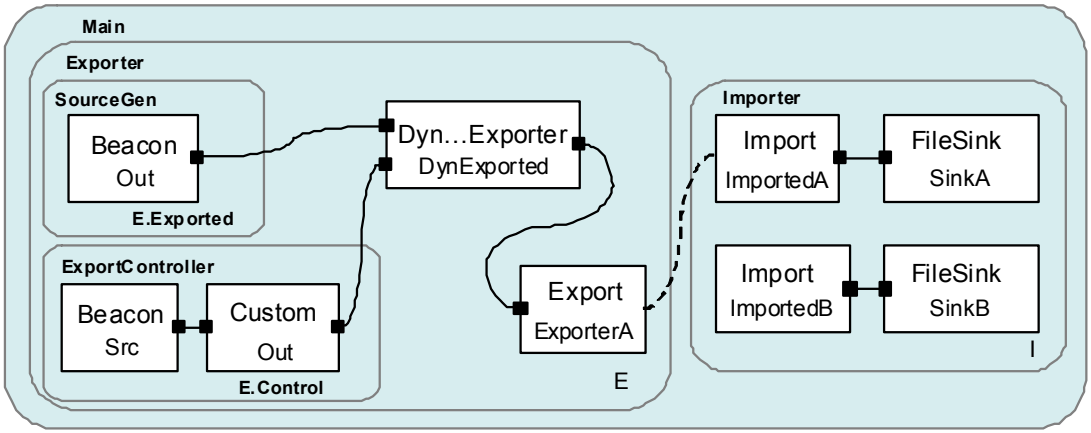


Figure 3-19 DynamicExporter

Where:

- DynamicExported is labelled as Dyn...Exporter, which is an abbreviation.
- Composite E is exporting stream DynExported, with property values priority=”low”, level=4l.
- The connection between the Export operator in the Exporter and one of the Import operators in the Importer represents the exported stream property being updated.
- Note that stream data flows from the composite Export operator to the Import operator (dashed line). Also note that they are pseudo operators and as such do not terminate with an operator port as do the other streams operators.
- Composite I dynamically toggles between importing the stream DynExported into one of the two FileSink operators. In the example code above, the Export operator in Composite E is initialized to send data to the first FileSink.

The C++ code for the process function of the DynamicExporter operator is shown in Example 3-20.

Example 3-20 Dynamic Exporter application C++ code

```
void MY_OPERATOR::process(Tuple const & tuple, uint32_t port)
{
    if(port==1){
        OperatorOutputPort & oport = getOutputPortAt(0);
        StreamPropertyCollection newCol;
        for (ConstTupleIterator it=tuple.getBeginIterator();
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000150

```
             it!=tuple.getEndIterator(); ++it) {
            ConstValueHandle handle = (*it).getValue();
            std::string name = (*it).getName();
            StreamProperty prop;
            bool success = setStreamProperty(name, handle, prop);
            if (success)
                newCol.push_back(prop);
        }
        try {
            StreamPropertyCollection oldCol;
            oport.getStreamProperties(oldCol);
            std::vector<std::string> names;
            for (StreamPropertyCollection::const_iterator
                   it=oldCol.begin(); it != oldCol.end(); ++it)
                names.push_back(it->getName());
            oport.removeStreamProperties(names);
            oport.addStreamProperties(newCol);
        } catch(...) {}
    } else {
        submit(tuple, 0);
    }
}

bool MY_OPERATOR::setStreamProperty(std::string const & name,
    ConstValueHandle handle, StreamProperty & prop)
{
    Meta::Type type = handle.getMetaType();
    switch (type) {
        case Meta::Type::RSTRING: {
            rstring const & value = handle;
            prop = StreamProperty(name, value);
            return true;
        }
        case Meta::Type::INT64: {
            int64 const & value = handle;
            prop = StreamProperty(name, value);
            return true;
        }
        case Meta::Type::FLOAT64: {
            float64 const & value = handle;
            prop = StreamProperty(name, value);
            return true;
        }
        case Meta::Type::LIST: {
            List const & value = handle;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000151

```
            Meta::Type elemType = value.getElementMetaType();
            switch (elemType) {
                case Meta::Type::RSTRING: {
                    list<rstring> const & lstValue = static_cast<list<rstring>
                       >(value);
                    prop = StreamProperty(name, lstValue);
                    return true;
                }
                case Meta::Type::INT64: {
                  list<int64> const & lstValue = static_cast<list<int64> >(value);
                    prop = StreamProperty(name, lstValue);
                    return true;
                }
                case Meta::Type::FLOAT64: {
                    list<float64> const & lstValue = static_cast<list<float64>
                       >(value);
                    prop = StreamProperty(name, lstValue);
                    return true;
                }
                default: return false;
            }
        }
        default: return false;
    }
    return false;
}
```

Walkthrough

The following steps provide a walkthrough for Example 3-20 on page 130:

1. The Main Composite invokes the Importer and the Exporter Composites. These are two independent flows. In a real world application of this pattern, it is more likely that the Importer and the Exporter flows are part of different applications.
2. The Exporter Composite operator creates and exports a stream. This stream is generated using the SourceGen Composite operator. The generated stream is fed into the DynamicExporter Primitive operator. This Primitive operator controls the export properties of the Export operator that consumes its output. It forwards the stream that it receives on its input port to the Export operator and uses the tuples it receives on its second input port to control the export properties. Whenever a new tuple is received on the control port, this operator will use the attribute names and values form the tuple to update the Export operator's export properties.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000152

The Export operator has an initial set of export properties specified, which will be replaced by the DynamicExporter Primitive operator at run time.

3. The SourceGen Composite operator is a simple workload generator that uses a Beacon operator and a Functor operator to generate periodic tuples with increasing numeric identifiers and a string attribute that carries the current time.
4. The ExportController Composite operator uses a Custom operator to periodically generate a tuple that contains an alternating set of export properties, one of {priority=”high”, level=3l} or {priority=”low”, level=4l}.
5. The Importer Composite operator imports two streams, one using a subscription expression that matches priority==”high” && level==3l and another that matches priority=="low" && level==4l. As the export properties switch from one to the next, one of these imports stops receiving tuples where the other one will start receiving them.

Example

As can be seen from the code, the data tuples are forwarded from the first input port to the output port, and the control tuples from the second input port are used to update the export properties associated with the output port.

A fragment of results from running this application is given in Table 3-1. Note that the export properties toggle around every four tuples. This is expected, as Composite E generates four tuples per second on each output steam, and the ExportController emits a control message that switches the stream subscription once per second.

Table 3-1 Application results

results.A	**results.B**
25,"Fri Aug 5 11:15:44 2011"	29,"Fri Aug 5 11:15:45 2011"
26,"Fri Aug 5 11:15:44 2011"	30,"Fri Aug 5 11:15:45 2011"
27,"Fri Aug 5 11:15:45 2011"	31,"Fri Aug 5 11:15:46 2011"
28,"Fri Aug 5 11:15:45 2011"	32,"Fri Aug 5 11:15:46 2011"
33,"Fri Aug 5 11:15:46 2011"	37,"Fri Aug 5 11:15:47 2011"
34,"Fri Aug 5 11:15:46 2011"	38,"Fri Aug 5 11:15:47 2011"
35,"Fri Aug 5 11:15:47 2011"	39,"Fri Aug 5 11:15:48 2011"
36,"Fri Aug 5 11:15:47 2011"	40,"Fri Aug 5 11:15:48 2011"

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000153

Refer to 3.2.9, “Updating import stream subscriptions” on page 120 more information. This pattern shows how to write a Primitive operator that updates a stream's import properties at run time.

3.2.11 Adapting to observations of runtime metrics

This design pattern concerns adapting to changes based on observations of system or user-defined metrics. We first briefly look at what metrics are, and then discuss how they can be used for adapting to changes.

The metrics support of Streams provides two types of metrics:

- System-supplied metrics
- User-defined metrics

System-supplied metrics are provided by the run time without any additional help from the developers. As an example, the number of tuples submitted by an operator port is a metric that the system keeps track of and makes available.

User-defined metrics are defined by the operator developers and their values are updated by the operator implementations. As an example, the number of invalid input lines could be a metric a Source operator can maintain, which the system then makes available.

The metrics are made available to both SPL application and external tools, such as scripts and programs. For applications, operator-level APIs are provided to access metrics associated with operators and PEs. Such APIs provide access to local metrics only (each operator can only access metrics that pertain to itself). These metrics can be used by the operator to adapt its behavior. As an example, by looking at a metric that specifies the number of tuples queued up on an operator's input port, the operator can adapt the processing it does to reduce the load.

For the scripts and programs, the set of all metrics associated with the applications and the Streams middleware components are made available, subject to security constraints. External tools, such as application management scripts, can poll the system for these metrics and take action depending on the metric values. The streamtool **`capturestate command`**, which is documented the IBM InfoSphere Streams Installation and Administration Guide (located at the address `http://publib.boulder.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.product.doc/doc/IBMInfoSphereStreams-InstAdmin.pdf`), returns an XML document containing a snapshot of the performance metrics.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000154

As an example, in an anomaly detection application, a user-defined metric provided by the application can be used by an application management script to determine that a suspicious condition has been detected, and as an action, the script could decide to launch a new application piece that does deeper analysis.

When to use

This pattern is useful when the knowledge obtained from metrics can be used by the application to adapt its behavior. Examples include, but are not limited to:

- Reactive-analytics: Information derived from metrics can be used to launch new applications, stop or replace jobs, or adapt to changing goals derived from the current processing.
- Load shedding: Performance metrics can be inspected to change the nature of the processing performed to adapt to the current load.

How to use

Add calls to the SPL functions that return operator metrics in the onTuple clause as shown in Example 3-21. In addition to the getInputPortMetricValue function, the getOutputPortMetricValue and getCustomMetricValue functions are available to retrieve built-in metrics associated with output ports, and custom user-defined customer metrics, respectively.

Implementation

In the implementation shown below, the Beacon and Throttle operators are used to feed a stream of tuples at a consistent rate to the Custom operator. The purpose of this Custom operator is to process the inbound tuple using either an expensive (doWorkFull) or cheap (doWorkCheap) algorithm, depending on the number of tuples waiting to processed. For each tuple, the Custom operator invokes the built-in getInputPortMetricValue function to obtain the number of tuples that are currently queued on its input port. If the more than 500 (represented as a constant “500l” of type long) tuples are queued, the cheap algorithm, which produces approximate results, is invoked. Otherwise, the input port is assumed to not be too congested, and the expensive, and presumably more precise, algorithm is used.

After 100,000 tuples are processed, the Beacon operator quiesces and produces a final punctuation. When this punctuation is received, the Custom operator prints out a the total number of times it invoked the cheap and expensive algorithms. The code for this application is shown in Example 3-21.

Example 3-21 Operator metrics

```
types
  Any = tuple<int32 data>;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000155

```
composite Main {
  graph
    stream<Any> Source = Beacon() {
      param
        iterations: 100u * 1000u;
    }

    stream<Any> Throttled = Throttle(Source) {
      param
        rate : 100000.0;
        precise : true;
    }

    () as Worked = Custom(Throttled as I) {
      logic
        state: {
          mutable uint64 cheap = 0, full = 0;
        }
    onTuple I: {
      mutable int64 nQueued = 0;
      getInputPortMetricValue(0u, Sys.nTuplesQueued, nQueued);
      if(nQueued>500l) {
        cheap++;
        doWorkCheap(I);
      } else {
        full++;
        doWorkFull(I);
      }
      }
    onPunct I : {
      if(currentPunct()==Sys.FinalMarker)
       printString("cheap="+(rstring)cheap+" "+
                   "full="+(rstring)full);
     }
   config
     threadedPort: queue(I, Sys.Wait, 1000);
    }
}

stateful void doWorkCheap(tuple<Any> work)
{
  // cheap but low accuracy
}

stateful void doWorkFull(tuple<Any> work)
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000156

```
{
  // costly but high accuracy
  block(0.0001);
}
```

We provide a diagram of the application in Figure 3-20.

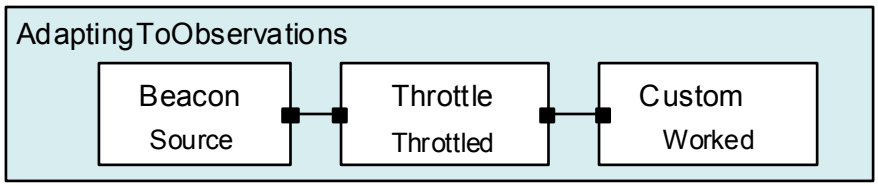


Figure 3-20 Operator metrics

Walkthrough

The following steps provide a walkthrough for Example 3-21 on page 135:

1. The first operator is a Beacon that produces a stream of tuples.
2. It is followed by a Throttle operator that throttles the Stream to 100,000 tuples per second.
3. The throttled stream is fed to a Custom operator that performs some custom logic.
4. The Custom operator has a threaded input port, and thus a queue associated with it.
5. The Custom operator inspects the queue size and if it detects that the queue is more than half full, it switches to processing tuples using the “cheap” algorithm that is low accuracy. Whenever the queue size goes below 500, it goes back to using the “full” algorithm that has high accuracy.
6. When all the data is processed, the Custom operator prints the number of times the “cheap” and the “full” versions of the algorithm were used.

Example

The result from running this example application (in stand-alone mode) looks like the following:

```
cheap=94959 full=5041
```

Additional reading

Refer to the *Streams Installation and Administration Guide* for the use of streamtool to access system and application metrics from the command line. This document comes with the Streams product.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000157

Refer to *SPL Standard Toolkit Types and Functions* for the use of SPL APIs to access metrics. For the Toolkit location, refer to “Online resources” on page 421.

Refer to 3.2.7, “Tuple Pacer” on page 111 for more information. This pattern shows how to adapt to changes by pacing the flow of tuples to match the processing speed of a downstream operator.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000158

4

InfoSphere Streams deployment

In this chapter, we describe InfoSphere Streams runtime deployment, which includes planning, installation, and configuration of the InfoSphere Streams run time on Linux clusters and configuration of Streams instances. In addition, we discuss Streams application deployment capabilities.

The following topics are discussed:

- InfoSphere Streams runtime deployment
  - Streams runtime architecture and services
  - Streams instances, and deployment topologies
  - Planning a Streams runtime deployment
  - Pre- and post-installation and configuration of the Streams environment
- InfoSphere Streams instance creation and configuration
  - Streams shared instance planning and configuration
  - Streams private development instance configuration
- InfoSphere Streams application deployment capabilities
  - Dynamic application composition
  - Operator placement
- InfoSphere Streams failover and recovery
  - Processing element recovery
  - Streams runtime services recovery

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000159

4.1 Architecture, instances, and topologies

In this section, we discuss and describe the Streams runtime architecture, instances, and deployment topologies.

4.1.1 Runtime architecture

Much like database management systems and J2EE application servers, InfoSphere Streams has a runtime engine that Streams administrators define, configure, and run. The term Streams instance is used to refer to the software configuration executing on one or more hosts (servers) that provides an environment for Streams applications to be started, stopped, managed, and monitored.

A Streams instance consists of a number of service processes that interact to manage and execute the Streams applications. Separate Streams instances can be created and configured independently from each other even though they may share some of the same physical hardware.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000160

Figure 4-1 shows a Streams instance with its constituent services that will be described in the following sections.

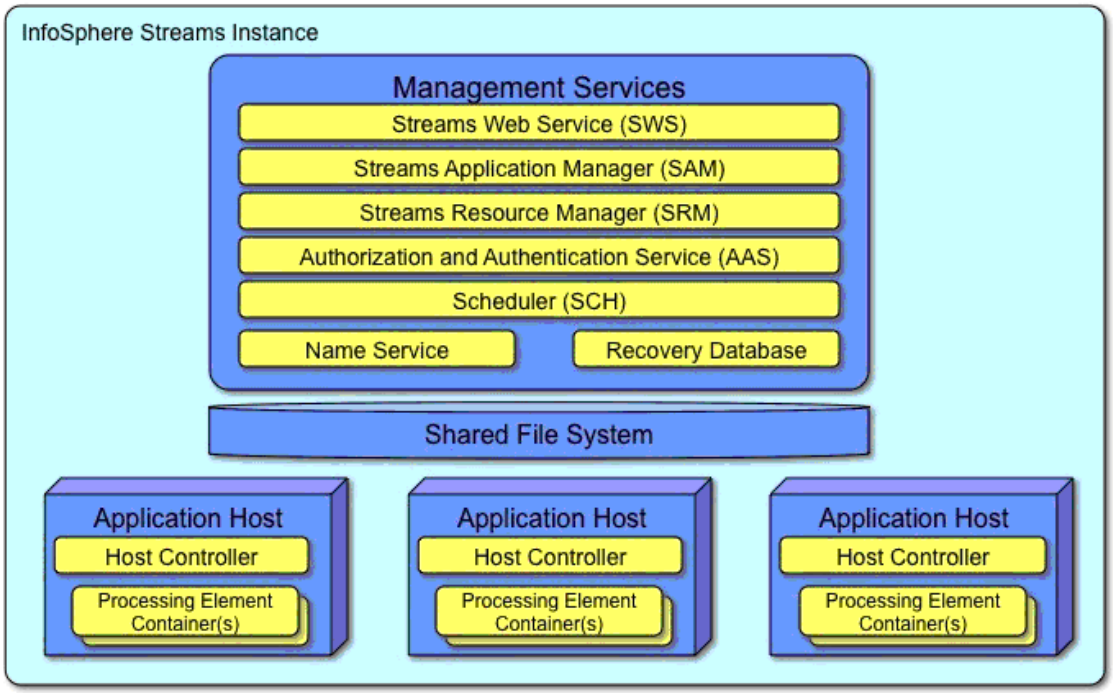


Figure 4-1 Services associated with the Streams instance

Management services

There is a single set of management services for each Streams instance. If the instance is instantiated on a single host, then all services will be run on that single host. If the instance is spread across multiple hosts, the management services can be placed on different hosts or co-located together on a single host. The management services include the following:

- Streams Application Manager (SAM)

  The Streams Application Manager is a management service that administers the applications in the Streams runtime system. The SAM service handles job management tasks, including user requests, such as job submission and cancellation. This service interacts with the Scheduler to compute the placement of processing elements (PEs) that are associated with an application. It also interacts with the Host Controller(s) to deploy and cancel the execution of PEs.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000161

- Streams Resource Manager (SRM)

  The Streams Resource Manager is a management service that initializes the Streams instance and monitors all instance services. SRM collects runtime metrics on the instance hosts and Streams components. This service also collects performance metrics that are necessary for scheduling and system administration by interacting with the Host Controller.

- Scheduler (SCH)

  The Scheduler is a management service that computes placement decisions for applications that are deployed in the Streams runtime system. The SCH service interacts primarily with the SAM service to handle job deployment requests, and with the SRM service to obtain the set of hosts that can be used for deployments. The SCH service also interacts with the SRM service to collect runtime metrics that are necessary for computing PE placement recommendations.

- Name service (NSR)

  The name service is a management service that stores service references for all the instance components. This service is used by all instance components to locate the distributed services in the instance so that Streams components can communicate with each other.

- Authentication and Authorization Service (AAS)

  The Authentication and Authorization Service (AAS) is a management service that authenticates and authorizes operations for the instance.

- Streams Web Service (SWS)

  The Streams Web Service (SWS) is an optional management service that provides web-based access to instance services. To use the Streams Console, SWS must be running on a host in the instance.

- Recovery database

  Streams instance recovery is an optional configuration setting. If recovery is configured, the Streams recovery database records the state of instance services. When instance management services fail, Streams restarts them and restores their state by retrieving the appropriate state data from the recovery database.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000162

Application host services

Application host services run on each host in a Streams instance that will be available to execute Streams applications. Those services include:

- Host Controller (HC)

  The Host Controller is an application service that runs on every application host in an instance. This service carries out all job management requests made by the SAM service, which includes starting, stopping, and monitoring PEs. This service also collects all performance metrics from PEs and reports the results to the SRM service.

- Processing Element Container (PEC)

  Streams Processing Language programs consist of operators, which are grouped at compile time into partitions. The execution container for a partition is a processing element (PE). Each partition runs in one PE. PEs are loaded by the Processing Element Container (PEC), which is started and monitored by the Host Controller for the host on which the PE is running.

Shared file system

A multihost Streams instances configuration requires a shared file system to share instance configuration information and SPL application binaries across the hosts.

Technologies used to implement a shared file system must be Portable Operating System Interface for UNIX (POSIX) compliant, and include, but are not limited to the following:

- Network File System (NFS)
- IBM General Parallel File System (GPFS™)

4.1.2 Streams instances

The Streams runtime environment is configured as a Streams instance. Each instance operates as an autonomous unit and can be configured with multiple options. In this section, we look at the most commonly used options for instances. For a complete set of options, refer to the *Streams Installation and Administration Guide* and the **`streamtool man command`**. You can find the guide at the following address:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000163

Each Streams instance is autonomous and isolated from other instances. You could create multiple instances on the same Streams installation, but this is not a best practice. Even though instances can share the same physical resources, there is no logical interference with other instances. (The use of multiple instances on the same set of physical resources can cause physical resource contention because the services for each instance are competing for the same CPU cycles and networking resources.) Streams SPL applications are run on an individual instance, and no other instances are aware of the jobs.

Streams instance IDs

A unique instance ID identifies each Streams instance within a Streams environment. The format of a fully qualified Streams instance ID is *instance-name@userid*, where instance-name provides a meaningful name for an instance to identify one from another in a multi-instance environment, and userid is the user name of the person who created the instance (instance owner). Note the use of the "@" character to separate the instance-name from the userid.

The instance-name must satisfy the following requirements:

- The name must be unique among the instances defined by you, the instance owner.
- The name must contain alphanumeric characters and start with an alphabetic character. Each of the alphanumeric characters must be one of the following ASCII characters: 0 - 9, A - Z, and a - z.
- The name can contain a dash (-), an underscore (_), or a period (.).

The following are examples of instance IDs that would be valid and unique in a multi-instance environment:

- streams@streamsadmin
- lab-inst01@streamsadmin
- stream_dev@mary
- stream_dev@john

Use the following guidelines when you use Streams instance IDs to specify Streams instances:

- The instance owner can use the full name of the Streams instance ID, or the just the instance name. The system will assume the userid is the user name of the current user.
- Users accessing an instance created by a different user must use the full name of the Streams instance ID (instance-name@userid), specifying the userid of the instance owner.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000164

Streams instance configurations

To make decisions about how many instances you will need, and how they will be configured, it is important to understand a couple of basic ways to categorize instances from a planning perspective. Streams instance configurations are characterized by configuration type and usage pattern or purpose. The configuration type is specified when you create a Streams instance, but the usage purpose will be chosen and governed by you.

Instance configuration types

Here are the two primary instance configuration types:

- Private instances

  Private instances are configured to only provide access to the instance owner. The instance owner is another term for the user that creates the instance. Private instances are easier to create and configure than shared instances, but they are not capable of supporting multiple users, and are not recommended for production use.

- Shared instances

  Shared instances allow a group of users to have access to a single instance. Access Control Lists (ACLs) are used to secure the instance and specify which users can administer the instance, and which users can submit and cancel jobs from the instance. Other than development instances, most instances should be created as shared instances.

Instance usage purpose

As you plan your Streams environment, you should plan to separate development activities, integration and test activities, and production operations into separate instances. Here are descriptions of those activities:

- Development instances

  Development instances are usually created as private (single user) instances, which use a single or small number of hosts. The primary purpose of these instances allows a developer to develop and test Streams applications, generic primitive operators, native functions, and so on.

  It is common for multiple development instances to share the same physical hardware.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000165

- Integration and Test instances

  Integration and Test instances are usually created as shared instances, which mimic a production environment. If separate Test and Integration instances are created, the Integration instance can use a smaller number of hosts than the production instance, but where possible, testing instances should provide an environment that is configured as much like the production instance as possible.

  The primary purpose of Integration and Test instances are to provide an environment where multiple users can run their applications in a single environment and verify adequate resource usage and availability, sharing of imported/exported streams, and cohabitation of applications.

  It is a best practice that Integration and Test instances be run on dedicated physical hardware.

- Production instances

  Production instances are primarily created as shared instances on a dedicated set of physical resources. These instances should be monitored for resource utilization to identify both CPU and I/O saturation.

Streams instance files

In addition to the runtime processes that make up a Streams instance, there are file system based components as well. Each Streams instance interacts with configuration, user authentication, and log files, which provide the persistent aspects of the instance.

Streams instance configuration files

The configuration information for each instance is stored in the `~/.streams/instances` directory of the instance owner ("~" is the Home directory in Linux). As configuration changes are made to the instance, these files are updated to persist the configuration.

Although instances are created using the tools from a Streams installation, instances themselves are separate from any Streams installation. If Streams is uninstalled, the instance configuration files remain untouched.

Each user's `~/.streams` directory should be backed up, and it is especially important to back up instance owners' `~/.streams` directories.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000166

Streams instance user authentication files

The public and private keys that Streams uses to authenticate Streams instance users are stored in either the default location of `~/.streams/instances/instance-name@userid/config/keys` or a directory configured by the instance owner. For more information, see “Configuring RSA authentication for Streams users” on page 169.

The authentication files should also be backed up on a regular basis.

Streams instance log files

When a Streams instance is active, both the runtime components and any running applications can generate log file content. Log files are located in `/tmp/streams.`*`instance-name@userid`*. This directory exists on each host that is part of the instance, but the contents of the directory will be different for each host, depending on what management services are executing on the host. Within this directory are two sub-directories: `logs` and `jobs`. The `logs` directory contains the logs for any management services running on the host. The `jobs` directory contains a separate directory for each job (for example, Streams application) that is currently executing on the host. Each of the jobs sub-directories contain the log files for the processing elements that is currently executing on that host.

When you stop a Streams application job, its corresponding log files are removed unless you use the collectlogs parameter the **`streamtool canceljob command`**.

All instance logs are removed when you stop an instance.

4.1.3 Deployment topologies

In preparation for planning a Streams runtime deployment, an understanding of different deployment topologies is necessary. In this section, we look at three of the most common topologies.

Single Host Topology

The simplest topology is of course, a single host computer running a single Streams instance. This is a configuration that is suitable for learning Streams, developing applications, and possibly running lower-volume production applications.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000167

Figure 4-2 shows this topology.

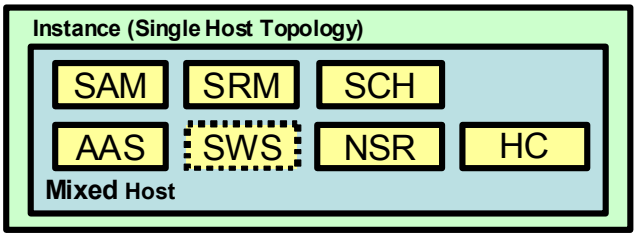


Figure 4-2 Single Host Topology

In a Single Host Topology, the host executes both the management services (SAM, SRM, SCH, AAS, NSR, and, optionally, SWS) and the application host services (HC and PEs when running an application). This type of host is referred to as a mixed host.

Configuration of a Single Host Topology is simplified because there are fewer *cluster* related setup and configuration steps required when preparing the operating system and environment.

This topology is well suited for single node VMWare Player/Workstation/Fusion installations.

Multi-Host Reserved Topology

The Multi-Host Reserved Topology is the most common and recommended topology for integration, test, and production environments. The *reserved* aspect of this topology refers to the usage of a single host for execution of the instance management services. This type of host is referred to as a Management Host.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000168

Figure 4-3 shows the Multi-Host Reserved Topology.

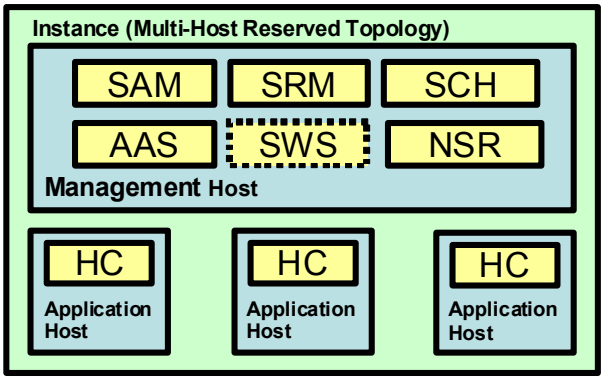


Figure 4-3 Multi-Host Reserved Topology

The additional hosts in this topology are configured to run the application host services, and are referred to as *application hosts*.

The advantage of this topology is the isolation between the management hosts and the applications hosts. Application hosts have dedicated resources for Streams application execution. In addition, a poorly written Streams application is not as likely to impact the stability of the entire instance. The management services are co-located on a single machine, and because all of the management services are required for the instance to be operational, it is considered a best practice to have a single point of failure, rather than multiple single points of failure.

The management services can be configured with a recovery database that allows them to be restarted on another host if they crash.

In environments with two to three hosts, the Host Controller (HC) service should be added to the management host to maximize resource utilization for applications that require it. This task can be accomplished by running **`streamtool addservice`**.

Multi-Host Unreserved Topology

The Multi-Host Unreserved Topology spreads the management and application host services across the nodes. In this configuration, most nodes are considered mixed hosts, although on large clusters there will be application hosts because there is a fixed number of management services.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000169

Figure 4-4 shows a sample Multi-Host Unreserved Topology.

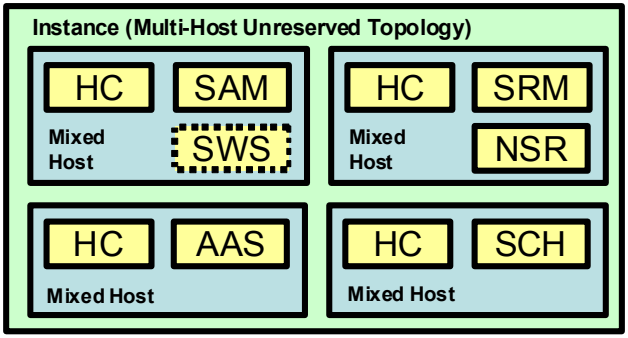


Figure 4-4 Multi-Host Unreserved Topology

This topology is not recommended for most configurations. It is found most commonly in environments where management service recovery is configured and the management services have been recovered onto different hosts.

Additional topology options

The Single Host and two multi-host topologies are examples of single instance topologies. If resources are limited, multiple instances can be configured to share resources. Figure 4-5 shows two instances running on a single host.

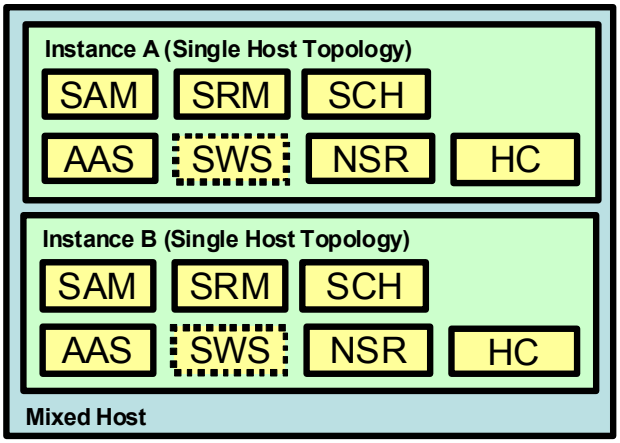


Figure 4-5 Single Host Multiple Instance Topology

Multiple instance topologies are typical in development environments. For more complex development environments, multi-host instances can be configured which share physical resources.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000170

4.2 Streams runtime deployment planning

In this section, we discuss the most common installation options you should consider when planning your InfoSphere Streams deployment. For a detailed list of all installation and configuration options, refer to the *Streams Installation and Administration Guide*.

The most important deployment options to identify and consider are:

- Streams Software Installation
  - Operating system version, configuration, and required packages
  - Streams software owner
  - Software installation directory (Streams and Eclipse)
  - Hosts to be configured for InfoSphere Streams usage
- Streams Instance Configuration
  - Streams administrator user (for shared instances)
  - Streams administrator home directory (Instance shared space)
  - Instance ID
  - Instance Administrative Group
  - Instance Users Group
  - Hosts assigned to the instance
  - Security Authentication Type
  - Instance Security Key Directory

4.2.1 Streams environment

There is a distinction between a Streams environment and Streams instances. A Streams environment is a set of one or more hosts that are capable of being part of an instance, but at any one time, the hosts may or may not be part of an instance. For example, you could install Streams on 10 hosts, and ensure all of the prerequisite and post-installation steps have been performed. At this point, you have a 10-host Streams environment.

The hosts in a Streams environment can be dedicated to a single instance, but in a development environment, they are often divided between a shared instance and hosts available to developers for private instances.

4.2.2 Sizing the environment

Part of designing a Streams system is sizing the hardware to determine the type and quantity of computers needed to support the envisaged applications with the necessary throughput and speed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000171

Sizing the hardware for an undeveloped application is a difficult activity. You need to determine the amount of hardware required to execute the application, and which steps of the application are the most processor, network, and memory intensive. To do these task, you need to consider the following criteria:

- Volumes of data flowing in from the data sources, which includes average data rates and peak data rates.
- Estimated data reduction at various stages of processing. Wherever possible, you should reduce the data volumes by filtering, enabling you to only process data items that are relevant to the application.
- Complexity of the Streams operators. It is useful to assess, when possible, the processing requirement of the operators and processing flows within Streams. These requirements can be evaluated by prototyping and measuring the workload and resource usage of the prototype or by comparing it with Streams applications that have similar complexity.
- Model the total workload. You should scale the measured or estimated processing requirements by the projected workload volumes to give a projection of the processing requirement, that is, how many compute hosts are required to support the envisaged workload.

There is a trade-off as to whether you size a system to cope with average load, peak load, or somewhere in between:

- Sizing to the peak volumes means that you can achieve your throughput and latency targets more frequently, but the machines will be underutilized for periods of time.
- Sizing to the average throughput rate will result in less hardware than a peak volumes sizing, which results in an overall lower cost system. However, the latency of end-to-end tuple processing is likely to increase during times of above-average traffic, which will, of course, be undesirable in low latency focused systems.

4.2.3 Deployment and installation checklists

Before performing a Streams deployment, there are two types of checklists that should be created. In Table 4-1 on page 153, we provide an example of a Streams environment checklist. This checklist provides the information necessary when installing and configuring the Streams environment, but does not specify the options for any instances that will be defined in the environment. Table 4-2 on page 155 gives an example of a Streams instance configuration checklist. You should create one of these checklists for each shared instance that will be created in your Streams environment.

This section describes the options and values provided in these checklists.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000172

Note: There are many additional configuration parameters for Streams environments and instances beyond the options and values shown in this chapter. For more information, refer to the *Streams Installation and Administration Guide*.

Streams environment checklist

This section describes the options and values that you should organize before performing a Streams environment installation.

Table 4-1 is an example of a Streams environment checklist.

Table 4-1 Sample Streams environment checklist

Option	Sample value
Streams version	2.0.0.0
Streams Software Owner	streamsadmin
Streams Software Group	streams
Streams Admin Home Directory (Must be shared by all nodes.)	/home/streamsadmin
Streams installation location ($STREAMS_INSTALL)	/opt/streams/InfoSphereStreams
Eclipse installation location	/opt/streams/eclipse
Streams Installation Hosts	redbook1 - redbook14
Streams Available Hosts	redbook8 - redbook14

The options in this table are explained in the following sections.

Streams version

The options discussed in this book are based on Streams Version 2.0. Streams versioning is composed of {major}.{minor}.{release}.{patch}. Major interfaces and configuration options as described in here should not change with release or patch changes.

Streams Software Owner

The Streams Software Owner is the user account that will own the Streams software after installation. You should install the software as the root user. You supply the Streams Software Owner account as part of the install process (interactive, console, or silent).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000173

Streams Software Group

The Streams Software Group is the group that is assigned to the Streams software after installation. You will supply the group as part of the install process.

Streams Admin Home Directory

Your Linux administrator will most likely configure the Streams Admin Home Directory. It is important to know this location as it will contain a `~/.streams` (pronounced *dot streams*) directory. The `~/.streams` directory contains the Streams instance configuration files. If the same user account is used for the Streams admin and Streams software owner, then the `~/.streams` directory will also contain the Streams global configuration files. This directory should be backed up regularly.

Streams installation location

When installing Streams, you can set the location of the installed Streams files. During the install, you are prompted to specify the location. If you are performing the installation as the root user (suggested approach), the default location is `/opt/ibm/InfoSphereStreams`. If you do not have root authority, the default location is `<your home directory>/InfoSphereStreams`.

A best practice is to override the default location and select a directory that allows you to co-locate the Eclipse installation for Streams Studio, InfoSphere Streams, and any additional toolkits you install. See Figure 4-6 for a sample hierarchy. Note the use of a symbolic link to hide the version number. This action facilitates installing future versions that can be switched between by changing the symbolic link. All references should be made through the link (for example, `/opt/streams/eclipse`).

```
/opt/streams_2.0.0.0/
 InfoSphereStreams/   - Streams Installation Location
 eclipse/             - Eclipse Installation Location
 scripts/             - Your own scripts in this location
 toolkits/            - Additional Toolkits Location
/opt/streams -> /opt/streams_2.0.0.0 - Symbolic link
```

Figure 4-6 Sample Streams deployment directory structure

Eclipse installation location

The Streams IDE requires the Eclipse IDE framework. You have the option to use an existing Eclipse installation, or a new one.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000174

If you are using an existing Eclipse installation, this value should contain the file system path to that installation. You will not need this path during the installation of Streams, but it is still a best practice to capture this information for setting up users $PATH environment variables so that they run the correct version of Eclipse.

If you installing a new Eclipse framework for use with Streams, you should select a directory that allows you to co-locate Eclipse with Streams (Figure 4-6 on page 154 for a sample).

Streams Installation Hosts

The Streams Installation Hosts are the set of all of hosts that are going to be part of your Streams environment. Each of these hosts have InfoSphere Streams installed and configured to be available as part of shared and private instances.

Streams Available Hosts

The Streams Available Hosts are the hosts that you will make available to Streams users for developing applications and creating their own (that is, private) instances. Hosts that you will dedicate exclusively to a shared instance should not be contained in this list, as they will be specified when you define the shared instance.

Streams instance configuration checklist

This section describes the options and values that you should organize in preparation for the creation of a shared Streams Instance. Table 4-2 shows a sample instance configuration checklist.

Table 4-2 Sample instance configuration checklist

Option	Sample value
Streams Admin User (Usually the same as the software owner)	streamsadmin
Streams Admin Home Directory (Must be shared by all nodes,)	home/streamsadmin
Streams Instance ID	streams@streamsadmin
Streams Admin Group	streamsadmin
Streams User Group	streams
Shared Instance Primary (Management) Host/Server	redbook1
Shared Instance Secondary (Application) Hosts/Servers	redbook2 - redbook7

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000175

Option	Sample value
Instance Shared Space	home/streamsadmin/.streams
User Shared Space	shared/data
Streams Name Service URL	DN: - Distributed Name Server (default)
Security Authentication Type	PAM (default)
Pam Enable Key	True (default)
Security Public Key Directory	home/streamsadmin/streams_keys

Streams Admin User

The Streams Admin User (also known as the instance owner) is the user account that will create and own your Streams instance. This user account should be the same as the Streams Software Owner. This user account can create multiple instances in a multi-instance topology.

You will not be prompted to specify this user account during the installation process, but you will use this account to create your instance.

Streams Admin Home Directory

Your Linux administrator will most likely configure the Streams Admin Home Directory. It is important to know this location as it will contain a `~/.streams` (pronounced *dot streams*) directory. The `~/.streams` directory contains the Streams instance configuration files. If the same user account is used for the Streams Admin and Streams Software Owner, then the `~/.streams` directory will also contain the Streams global configuration files. This directory should be backed up regularly.

Streams instance ID

The Streams instance ID is discussed in “Streams instance IDs” on page 144. This deployment option refers to a shared instance that will be created by the Streams Admin for use by multiple users.

Your deployment may have multiple instances. Some will be single-user developer instances that are somewhat transient, but likely you will have one or more shared instances.

Instances are not created as part of the installation. You will create them after installing the software.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000176

Streams Admin Group

This option is the name of the administration group to use during instance creation to initialize the access control list for shared instances. This group must exist in the authentication database before creating the shared instance.

Users in the administration group will have permissions to manage all configuration parameters and streams jobs in the instance. As an example, users in the admin group will be able to cancel jobs submitted by any user.

Streams User Group

This option is the name of the users group to use during instance creation to initialize the access control list for shared instances. This group must exist in the authentication database before creating the shared instance.

Users in the user group will have permissions to start, stop, and manage their own jobs in the Streams instance.

Primary (Management) Host / Server

The Primary host represents the server that will host the management services in a Multi-host Reserved Topology. If you are installing a single host instance, this will be the only host that your instance uses. In the case of a Multi-host Unreserved Topology, identifying a single node where you will execute commands from is a best practice.

Secondary (Application) Hosts / Servers

The Secondary hosts are the additional hosts that will be configured to be part of your Streams environment. Some of these hosts may be reserved for private developer instances and will not be part of your shared instance. In these cases, it is still important to identify these hosts because they do require Streams to be installed.

Instance Shared Space

The Instance Shared Space is the location where Streams Instance configuration files are located. In the current release of Streams, this location is located within the `~/.streams` directory of the Streams instance owner.

User Shared Space

The User Shared Space is file system storage that is shared across all hosts in a Streams instance. This space is usually in addition to the users home directories that are also are shared across the hosts.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000177

The primary purpose of this space is for application and toolkit development. Permissions are often opened up to include a development group for collaboration and sharing.

NFS is not the best choice for this space because of latency issues observed with NFS in a distributed computing environment. The IBM General Parallel File System (GPFS) is a low-latency, high performance, and highly distributed file system that works well for this purpose. It can use either extra local disks on the cluster hosts and it can also use SAN attached storage. GPFS is licensed and distributed separately from InfoSphere Streams.

Streams Name Service URL

This option is the URL for the name service the instance will use. This option provides a reference to either a distributed or file system based name service. The default is a distributed name service.

Specify 'DN:' to use the distributed name service. This is the best value if you are not configuring your instance for management service recovery.

Specify a value of 'FS:' to use the file system based name service. A default file path will be used unless a path is specified following the 'FS:'. The directory path must not contain white space characters. If you are configuring management service recovery, then you must specify the file system based name service.

Security Authentication Type

The run time's authentication processing integrates with the customer's existing user authentication infrastructure. Integration with Pluggable Authentication Modules (PAM) for Linux and Lightweight Directory Access Protocol (LDAP) back ends are supported. All user repository management occurs outside of InfoSphere Streams, directly using the back-end infrastructure's toolset.

The instance's authentication configuration is established when the instance is created. By default, the PAM back end is used. The PAM authentication defaults to a PAM service named *login*. To customize the name of the default PAM service for your installation, refer to the *Streams Installation and Administration Guide*.

The integration with PAM also enables use of a public/private key mechanism (created by the **`streamtool genkey command`**) to enable the construction of a secure environment where the user is not prompted to provide their password. To avoid password prompting for shared instances, the public key for users authorized to an instance should be copied to the directory identified by the SecurityPublicKeyDirectory instance configuration property. To disable the use of keys, specify `'--property PamEnableKey=false'` with **`streamtool mkinstance`**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000178

PAM is the suggested authentication mechanism.

A security template that uses LDAP can be used if the installation owner has enabled LDAP. The --template ldap option can be used to specify LDAP authentication if it has been enabled. To configure the use of LDAP for your installation, refer to the *Streams Installation and Administration Guide*.

Security Public Key Directory

The Security Public Key Directory property is the PAM-specific property for Streams instances. Enabling RSA authentication for Streams users involves managing the user's public keys and storing them in the directory path that is specified in this property.

Attention: It is extremely important that access to the RSA public keys directory be restricted and that you ensure the secure management of the public keys. For more information, refer to the *Streams Installation and Administration Guide*.

The default locations for the RSA public keys are as follows:

- For a private instance, the default directory is:

  `~instance_owner/.streams/key`

- For a shared instance, the default directory is:

  `~instance_owner/.streams/instances/instance_name/config/keys`

In environments that support multiple shared instances with a single instance owner (for example, Instance Admin), it is common to create a single keys directory in the instance admins home directory and set the instance configuration parameter (SecurityPublicKeyDirectory) to point to this location when creating the instance. This action allows multiple instances to use the same set of keys and centralize the management of these keys. Access control can still be maintained at the instance level to limit users abilities on each instance.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000179

4.3 Pre- and post-installation of the Streams environment

In this section, we discuss the requirements and preparations that must be performed before installing InfoSphere Streams.

4.3.1 Installation and configuration of the Linux environment

In this section, we define Installation and configuration of the Linux environment.

InfoSphere Streams cluster

One of the most overloaded terms in recent years is the term *cluster* when referring to a set of computers configured to work cooperatively to accomplish tasks. There are several categories of clusters including, such as high availability (HA) clusters, load-balancing clusters, and compute clusters. InfoSphere Streams does not require any additional cluster hardware or software to be used in a clustered configuration.

InfoSphere Streams provides the functionality of a compute cluster for streaming-data applications. In addition, Streams provides high availability and load-balancing capabilities for Streams applications.

Hardware requirements

InfoSphere Streams V2.0 requires Intel/AMD x86 (32- or 64-bit) hardware platforms. No special hardware components are required.

For multi-node (cluster) configurations, the hosts must be connected to a local area network.

Linux operating system requirements

All IBM InfoSphere Streams hosts must be running a supported operating system and satisfy all Streams operating system-related requirements.

The supported Red Hat Enterprise Linux (RHEL) operating system versions for Streams Version 2.0 are:

- RHEL 5.3, 32-bit and 64-bit
- RHEL 5.4, 32-bit and 64-bit
- RHEL 5.5, 32-bit and 64-bit
- RHEL 5.6, 32-bit and 64-bit

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000180

Streams V2.0 does not support other Linux operating systems such as CentOS or Fedora. If you are upgrading Streams from a previous release, you might need to upgrade your Red Hat Enterprise Linux operating system version.

If you are running Streams across multiple hosts, you must satisfy the following operating system requirements:

- All hosts in an instance must run the same operating system version and architecture. For example, all hosts in an instance can run Red Hat Enterprise Linux 5.5, 64-bit.
- Different instances can use hosts that do not run the same operating system version. However, the hosts must run the same operating system architecture. For example, one instance can use hosts that run Red Hat Enterprise Linux 5.4, 64-bit, and another instance in the same Streams installation can use hosts that run Red Hat Enterprise Linux 5.5, 64-bit.

Streams provides tools that help you to verify this requirement on single and multiple hosts:

- Before install, run the dependency checker script to verify requirements on individual hosts.
- After install, use the **`streamtool checkhost command`** to verify requirements on single or multiple hosts.

Red Hat installation and configuration

The standard Red Hat installation meets the requirements of InfoSphere Streams, but there are several installation options available as you perform the operating system installation. In addition, there are several post installation environment configurations that must be configured, such as DNS and Shared File System. In Table 4-3, we provide guidance for these options.

Table 4-3 Red Hat installation and configuration options

Topic	Installation options
Red Hat Version	RHEL 5.3, 32-bit and 64-bit RHEL 5.4, 32-bit and 64-bit RHEL 5.5, 32-bit and 64-bit RHEL 5.6, 32-bit and 64-bit
RAM	Minimum: 1 GB (possible but not ideal). The amount of RAM required by Streams will be dependent on the Streams applications that will be developed and deployed on the Streams hosts. Recommended: 8 GB +

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000181

Topic	Installation options
SELinux	DISABLED[a] You should disable SELinux for your first Streams installation.
Linux Firewall	DISABLED Firewall protection is recommended at the perimeter of the Streams cluster. Firewalls between cluster nodes will introduce latency.
Character Encoding	UTF-8
File System Space	`/opt` minimum: 2 GB `/opt` recommended: 8 GB +
Networking	Each host must have full TCP networking capabilities (binding to non-reserved TCP sockets) that are available to programs running in user mode. When you create a Streams instance or add hosts to an instance, you cannot use localhost or any other host name or address that resolves to a loopback address (127.*.*.*). The name of the host returned by **`hostname`** must match an entry in the Linux name resolver, for example, DNS, LDAP, Yellow Pages, or `/etc/hosts`.
Common User Management System (for multi-host topologies)	The Streams software owner, instance administrators, and users must have consistent UIDs across all hosts in the cluster. A common user management system (for example, LDAP or NIS) is a best practice.
X Window System	An X Window System interface can be used so that Streams nodes can have the Streams Studio IDE enabled. The Streams Studio IDE must run on a host configured to run InfoSphere Streams.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000182

Topic	Installation options
Consistent & Reliable Host Resolution	Use a host name and resolution technology which ensures: ► Forward and reverse lookup consistent across the cluster of nodes. ► Each host must be resolvable to itself and the other nodes through the same FQDN or short names, but not a mix. ► Complications are introduced when the system host name is not completely aligned with local host files, NIS, LDAP, or DNS.
NTP	Ensure all nodes synchronize to the same time source.
Default Shell	A bash shell should be configured as the default shell for all Streams users.
Shared File System Space (for multi-host topologies)	Use a technology such as IBM GPFS or NFS.
VNC Server	VNC Server is the recommended approach for remote display from Streams hosts to a user workstation. VNC Server is available as part of the Red Hat installation.

a. Streams does support SELinux, but it should be disabled unless you are going to enable and configure Streams for SELinux.

User and group configuration

Before performing the Streams installation, you must create the Streams Software Owner and Streams Software Group that you identified in the Installation and Configuration Checklist in Table 4-1 on page 153. Your Linux administrator will be able to create these users and groups. You will be prompted for them if you perform the installation as the root user.

4.3.2 InfoSphere Streams installation

After you have configured your Red Hat Linux environment correctly, you are ready to install InfoSphere Streams and its prerequisite software packages on each of the hosts that will part of your Streams environment.

Perform the Streams installation on every host that will part of your Streams environment.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000183

Although it is possible to install Streams on a shared disk that is available to every host, this action should be avoided because any requirement to fetch a remote page of the binary during run time will be fighting with the network traffic of the Streaming application, which could lead to stalls in the data processing, potentially leading to a backup.

Refer to Appendix A, “InfoSphere Streams installation and configuration” on page 369 for detailed Streams installation procedures.

4.3.3 Post-installation configuration

After InfoSphere Streams has been installed on all of the hosts in your Streams environment, there are several post-installation tasks that are recommended or required before using Streams.

The recommendations in this section are based on real-world experiences deploying Streams clusters for development, integration, and production environments.

Verifying execution permission on /sbin/ifconfig

Recent security concerns have prompted Linux system administrators to harden and lock down most Linux installations. In some cases, access to the **`/sbin`** commands have been revoked from all users except for the ones with root authority.

Streams requires that instance administrators have execute permission on **`/sbin/ifconfig`**. If you cannot execute this command, have your Linux administrator grant all of your instance administrators execute permission. This task can be accomplished by running the following command:

```
# setfacl -mask -m u:streamsadmin:rx /sbin/ifconfig
```

Configuring the Streams environment variables

InfoSphere Streams provides the **`streamsprofile.sh command`** for configuring user’s environment variables. In addition to the Streams environment variables, most users will need to have additional environment setup performed, including:

- Adding the Streams Studio IDE (Eclipse) command to their execution path ($PATH)
- Configuring the JAVA_HOME environment variable
- Adding Java commands to their execution path ($PATH)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000184

To avoid a situation where each user adds these items to their own startup scripts (for example, .bashrc), a user with root authority should create a single script placed in `/etc/profile.d` that will set up the Streams environment for any user logging into one of the Streams hosts. This script needs to be configured on every host in the Streams environment. The script shown in Example 4-1 is a sample based on the suggested deployment layout. Note that the default streams instance (streams@streamsadmin) is configured (see bold italic text) when the **`streamsprofile.sh command`** is sourced.

Example 4-1 Streams environment variables

```
#!/bin/bash
# /etc/profile.d/streams.sh
# Setup Streams environment for users
#
streams_path="/opt/streams"
streams_dir="InfoSphereStreams"
eclipse_dir="eclipse"
toolkits_dir="toolkits"

pathmunge () {
   if ! echo $PATH | /bin/egrep -q "(^|:)$1($|:)" ; then
      if [ "$2" = "after" ] ; then
         PATH=$PATH:$1
      else
         PATH=$1:$PATH
      fi
   fi
}

### InfoSphere Streams
# Setup default instance for users (remove -i option for streams
default)
source ${streams_path}/${streams_dir}/bin/streamsprofile.sh -i
streams@streamsadmin

### Toolkits, including non standard toolkits included with Streams
export
STREAMS_SPLPATH=${streams_path}/${streams_dir}/toolkits:${streams_p
ath}/${toolkits_dir}

### Streams Studio Eclipse
export ECLIPSE_HOME=${streams_path}/${eclipse_dir}
pathmunge ${ECLIPSE_HOME}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000185

```
### IBM Java
export JAVA_HOME=/opt/ibm/java-x86_64-60
pathmunge ${JAVA_HOME}/bin
pathmunge ${JAVA_HOME}/jre/bin

### Setup umask for Streams
umask 0002
```

Defining the Streams available hosts

As described earlier, there is a distinction between a Streams environment and Streams instances. Within a Streams environment, there are usually hosts that are exclusively used for shared instances and are not considered available for use in development or temporary instances.

In this section, you will define the set of hosts that are available for use by Streams users to create additional instances (for example, development instances). To use instances from the set of available hosts, the instance will be defined using the numHosts option. The number of hosts requested will be provided from the set of available hosts defined in this section.

The *Streams Installation and Administration Guide* defines a series of steps that the **`streamtool mkinstance command`** will use to determine the set of hosts that are available in the Streams environment as follows:

1. **`~/.streams/bin/streams_get_available_hosts`**
2. **`~/.streams/config/hostfile`**
3. **`~install_owner/.streams/bin/streams_get_available_hosts`**
4. **`~install_owner/.streams/config/hostfile`**

We suggest the following approach for creating the list of available hosts in your Streams environment:

- Do not list hosts that will be exclusively part of a shared instance as available. These hosts will be specified by name when creating the shared instance.
- Create a list of hosts in the Streams software owners `~/.streams` directory (`~install_owner/.streams/config/hostfile`) that lists the hosts that are not dedicated to a shared instance. Developers can use these hosts to create their own private instances.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000186

Example 4-2 contains a sample available hosts file based on the example used in this book. Lines that begin with a pound sign (#) are comments. The hosts are listed on separate lines along with the host tags that represent how they can be used.

Example 4-2 Sample available hosts file

```
# /home/streamsadmin/.streams/config/hostfile
# Our Streams environment
# redbook1 - redbook7 are exclusively used for
# streams instance streams@streamsadmin
# redbook8 - redbook14 are available
redbook8, execution, build
redbook9, execution, build
redbook10, execution, build
redbook11, execution, build
redbook12, execution, build
redbook13, execution, build
redbook14, execution, build
```

There are two predefined host tags:

- Build

  A build host is one that has tools such as compilers, linkers, and libraries that are necessary to build an SPL application. The SPL compiler uses one or more of these hosts when performing remote parallel builds.

- Execution

  An execution host can be used to run SPL applications. These are the hosts that can be included in a Streams runtime instance.

In addition to the two predefined host tags, you can apply additional host tags to the available hosts. These additional host tags can be used during application deployment design to help control operator placement on specific hosts. For more information about user defined tags and host placement, see 4.5.2, “Operator host placement” on page 184.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000187

You can test and verify your method for defining the available hosts by running **streamtool lsavailablehosts -v**. The -v option will include the location of the source of available hosts. Example 4-3 illustrates this situation in our test environment:

Example 4-3 Test environment

```
$ streamtool lsavailablehosts -v
The source location of the host information is
'/home/streamsadmin/.streams/config/hostfile'.
Host                    Tags
redbook8   build,execution
redbook9   build,execution
redbook10  build,execution
redbook11  build,execution
redbook12  build,execution
redbook13  build,execution
redbook14  build,execution
```

Configuring a shared file system space

Multi-host Streams instances depend on a shared storage space area that is accessible from each of the hosts that are part of the instance. You should use a shared file system with extremely low latency, such as IBM General Parallel File System (GPFS); NFS is sufficient for small installations (for example, two to four hosts).

In addition to the shared home directory file space, you should have a dedicated area for software development that has shared access to all members of the development team. Even if you are using a software configuration management system (as examples, Clear Case and Subversion), it is useful to have a shared area for test data, test applications, test results, and any other shared files required by your project.

4.3.4 Streams user configuration

In this section, we discuss the tasks that need to be performed to enable Streams users to interact with the environment. Streams users are defined as anyone who interacts directly with the Streams environment or a Streams instance. These activities include:

- Creating and configuring private or shared instances
- Managing and monitoring Streams instances
- Compiling and running Streams applications

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000188

Configuring Secure Shell for Streams users

Streams requires the use of Secure Shell (SSH) without a passphrase for communications between hosts in both single and multi-host installations. Even in a single host environment, you need to configure the instance owners account so that it can perform an SSH connection back to the same host.

You can either set up SSH without a passphrase, or use a passphrase caching programs such as ssh-agent.

Example 4-4 shows the process to configure ssh without a passphrase on Linux systems.

Example 4-4 Configure SSH

```
$ cd ~/
$ ssh-keygen -t dsa
Generating public/private dsa key pair.
Enter file in which to save the key
(/home/streamsadmin/.ssh/id_dsa): <ENTER> TO ACCEPT DEFAULT
Created directory '/home/streamsadmin/.ssh'.
Enter passphrase (empty for no passphrase): <ENTER>
Enter same passphrase again: <ENTER>
Your identification has been saved in
/home/streamsadmin/.ssh/id_dsa.pub.
The key fingerprint is: 6e:72:23:80:55:ec:73:d1:7f:10:09:16:a7:71:61:0d
streamsadmin@redbook1

$ cd ~/.ssh
$ cat id_dsa.pub >> authorized_keys
$ chmod 600 *
$ chmod go-w ~
```

Configuring RSA authentication for Streams users

Using RSA authentication eliminates the need for users to enter a password when running Streams commands that require authentication. By default, instances are created with RSA authentication enabled. We suggest that all users have a Streams RSA key generated as part of user setup.

Each user needs to generate their own public and private keys using the following command:

```
$ streamtool genkey
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000189

The **streamtool genkey command** generates the following two files and places them in the ~/.streams/key/ directory:

- Private key: username_priv.pem
- Public key: username.pem

The generated keys for a user are not instance-specific, and can be used for any Streams instance that is configured to enable RSA authentication.

Attention: Take extreme care to guard your private key, as you would a password. The file must be readable only by your user ID.

To configure your private instances, no additional configuration steps are required.

To configure a shared instance, work with the instance administrator to copy your public key to the path specified in the instance's SecurityPublicKeyDirectory property.

4.4 Streams instance creation and configuration

In this section, we discuss the steps used to create, start, and verify a shared Streams instance. In addition, we discuss how developers can create their own private development instances using the set of hosts in their environment that are not reserved for shared instances. The examples in this section are based on the sample deployment and installation checklists described in Table 4-1 on page 153 and Table 4-2 on page 155.

4.4.1 Streams shared instance configuration

A Streams shared instance can be created using any of the topologies discussed in 4.1.3, “Deployment topologies” on page 147. In this example, we define a shared instance using a multihost reserved topology.

The commands in this section should be performed as the instance administrator (often the same account as the Streams Software Owner account). In our example, this will be the *streamsadmin* account.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000190

The management host, application hosts, and boot logs

In a multi-host reserved topology, one host is identified as the management host and will run all of the management services of the instance. The commands to configure and interact with a Streams instance can be executed from any of the hosts in the Streams environment (including those that are not even part of the instance).

You should issue commands and perform the configuration on the management host because it will have the most information about the status of starting the instance. A boot log is created on each host in the instance and located in `/tmp/<instance name>@<instance owner>.boot.log` (for example, `/tmp/streams@streamsadmin.boot.log`). The boot log on each host contains the boot (or startup) log for the services that are running on that host. The boot log on the management host, therefore, will have the boot information for each of the management services and it will be most accessible to you if you are issuing the instance configuration and start commands on that host.

In a multi-host reserved topology, there are one or more application hosts. When you start an instance, the application hosts will have boot logs in `/tmp` with the same name as the boot log on the management host. The boot logs on application hosts will only show the log for starting the host controller service.

Specifying shared instance hosts

There are two common ways to define the specific hosts that will be part of your shared Streams instance using the **`streamtool mkinstance command`**:

1. Use the --hfile option to specify a file that contains the host names of the hosts to be included in the instance. The first host listed will be the management host.
2. Use the --hosts option to specify a comma-separated list of host names. The first host listed will be the management host.

Note: There are many additional options described in the *Streams Installation and Administration Guide* for host selection; however, we are presenting the most common for shared instances, which use a reserved topology.

We suggest using the first option, which simplifies the **`streamtool mkinstance command`**, while providing a single place to make changes if you need to remove and recreate the instance. We created the host names file shown in Example 4-5 for our shared instance:

Example 4-5 Sample mkinstance host names file (for use with the -hfile option)

```
# /home/streamsadmin/streams@streamsadmin_hosts
# List of hosts for shared instance: streams@streamsadmin
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000191

```
# First host listed will be management host
redbook1
redbook2
redbook3
redbook4
redbook5
redbook6
redbook7
```

Instance host tags

In addition to simply specifying the hosts that are used in the instance, you have the option to apply host tags to any or all of the hosts in the instance. The tags are user-defined labels that can be used to identify capabilities, configurations, or any other differentiation feature of the hosts. These tags can be used during application design and deployment to control how operators are distributed and placed on hosts. For more information about user-defined tags and host placement options, see 4.5.2, “Operator host placement” on page 184. Example 4-6 shows a host file with tags applied to some of the hosts.

Example 4-6 Sample mkinstance host names file (to use with -hfile) with host tags

```
# /home/streamsadmin/streams@streamsadmin_hosts
# List of hosts for shared instance: streams@streamsadmin
# First host listed will be management host
redbook1
redbook2 --tags ingest,fpga
redbook3 --tags ingest,fpgs
redbook4 --tags dbclient
redbook5
redbook6 --tags sanstorage
redbook7 --tags sanstorage
```

Creating and populating the security public key directory

The run time's authentication processing integrates with the customer's existing user authentication infrastructure. Integration with PAM and LDAP back ends are supported. The product, in our current example, uses PAM authentication in this example because it is easiest to configure and maintain. All user repository management occurs outside of InfoSphere Streams, directly using the back-end infrastructure's toolset.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000192

The instance's authentication configuration is established when the instance is created. By default, the PAM back end is used. The PAM authentication default is configured to use a PAM service named *login*. To customize the name of the default PAM service for your installation, refer to the *IBM InfoSphere Streams Installation and Administration Guide*. It is located at the following URL:

http://publib.boulder.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.product.doc/doc/IBMInfoSphereStreams-InstAdmin.pdf

Another PAM integration default is configured to provide for the use of a public / private key mechanism to enable the construction of a secure environment where the user is not prompted to provide their password. To avoid password prompting for shared instances, the public key for users, authorized to an instance, should be copied to the directory identified by the SecurityPublicKeyDirectory instance configuration property.

The following commands are used to create a public key directory in the stream administrators home directory:

```
$ mkdir /home/streamsadmin/streams@streamsadmin_keys
$ chmod 700 /home/streamsadmin/streams@streamsadmin_keys
```

Previously in this chapter, we showed you how to create a public key and private key for each Streams user using the **streamtool genkey command**. At this point, you need to copy the private key for each user that will be authorized to use the shared instance into the public key directory you just created. The streams instance administrator key must also be copied into the key directory. The following is an example of the commands required:

```
$ cp /home/streamsadmin/.streams/key/streamsadmin.pem
/home/streamsadmin/streams\@streamsadmin_keys
```

Note: Copy the key for every use that will be authorized to use this shared instance into the directory. Keys can be added at any time. They can even be added after the instance has been created and started.

Creating and starting the shared instance

After defining the instance hosts file and creating the security public key directory, it is time to create the shared instance.

The **streamtool mkinstance command** is used to create Streams instances. For complete information about all of the options, refer to the *Streams Installation and Administration Guide* and the output from the **streamtool man mkinstance command**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000193

The following command creates the shared instance for our example:

```
$ streamtool mkinstance -i streams@streamsadmin --hfile
/home/streamsadmin/streams@streamsadmin_hosts --template shared
--property
SecurityPublicKeyDirectory=/home/streamsadmin/streams@streamsadmin_keys
--property AdminGroup=streamsadmin --property UsersGroup=streams
CDISC0040I Creating Streams instance 'streams@streamsadmin'...
CDISC0001I The Streams instance 'streams@streamsadmin' was created.
```

Note: Save this command in a script file for reuse and documentation.

After the instance has been created, you can start the instance using the following command:

```
$ streamtool startinstance -i streams@streamsadmin
CDISC0059I Starting up Streams instance 'streams@streamsadmin'...
CDISC0078I Total instance hosts: 7
CDISC0056I Starting a private Distributed Name Server (1 partitions, 1
replications) on host 'redbook1'...
CDISC0057I The Streams instance's name service URL is set to
DN:redbook1.mydomain.net:34875.
CDISC0061I Starting the runtime on 1 management hosts in parallel...
CDISC0060I Starting the runtime on 6 application hosts in parallel...
CDISC0003I The Streams instance 'streams@streamsadmin' was started.
```

Verifying the instance

After your instance has been started, there are several **streamtool** commands that can be used to manage and configure your instance:

```
streamtool (get/set/rm)property
streamtool (ls/add/rm)host
streamtool (add/rm)service
```

For a complete list of **streamtool** commands, run **streamtool man**.

The **streamtool getresourcestate command** is the best way to get a quick snapshot of the health of the services of an instance. In Example 4-7, we show the results of this command immediately after starting the instance successfully:

Example 4-7 Command results

```
$ streamtool getresourcestate -i streams@streamsadmin
Instance: streams@streamsadmin Started: yes State: RUNNING Hosts: 7
(1/1 Management, 6/6 Application)
Host            Status   Schedulable  Services
Tags
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000194

```
redbook1        RUNNING  -          RUNNING:aas,nsr,sam,sch,srm,sws
redbook1         RUNNING  yes       RUNNING:hc
redbook2         RUNNING  yes         RUNNING:hc
redbook3         RUNNING  yes         RUNNING:hc
redbook4         RUNNING  yes         RUNNING:hc
redbook4         RUNNING  yes         RUNNING:hc
redbook4         RUNNING  yes         RUNNING:hc
redbook5         RUNNING  yes         RUNNING:hc
redbook6         RUNNING  yes         RUNNING:hc
```

4.4.2 Streams private developer instance configuration

In addition to interacting with a shared instance, your developers will want to perform development using their own private instance. This approach provides isolation from other developers; however, the overlap of development instances on hosts can lead to resource constraints. Therefore, private developer instances should be used for functional development and testing of Streams applications; however, performance testing should be performed on a shared instance with exclusively dedicated hosts.

Development instances are simplified versions of shared instances for use by a single user. They can use one or more hosts from the set of available hosts configured in your Streams environment.

The developer instance configuration template simplifies instance creation and provides many default values and properties. The most important aspects of the developer template are as follows:

- It creates a private instance using the creators private key to authenticate access.
- The Streams Web Service (SWS) HTTPS port is randomly chosen from those ports available when the instance is started.

Important: At this time, it is possible for a user to manually specify hosts that are being used by shared instances. Users should be encouraged to use the --numhosts option for creating instances.

The commands in this section can be performed by any Streams user account configured as described in 4.3.4, “Streams user configuration” on page 168.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000195

STREAMS_DEFAULT_IID environment variable

The **streamtool** commands allow the user to specify the instance to interact with by using the -i or --instance-id flags. If a user does not use this option, **streamtool** will use the value of the STREAMS_DEFAULT_IID environment variable to determine the instance to target with the **streamtool** command. In an environment where a shared instance is set up as the default instance ID, users should override the STREAMS_DEFAULT_IID if they are going to be working with a private developers instance rather than the shared instance. This action will reduce confusion if the instance option is accidentally omitted when using **streamtool** commands.

Creating a single host private developer instance

The **streamtool mkinstance** default for host selection is --numhosts 1. This option will select the first host from the available hosts configured for the Streams environment.

The following commands create and start a single host private developer instance on one of the available hosts configured in the Streams environment:

```
$ streamtool mkinstance -i streams@devuser --template developer
CDISC0040I Creating Streams instance 'streams@devuser...
CDISC0001I The Streams instance 'streams@devuser' was created.
$ streamtool startinstance -i streams@devuser
CDISC0059I Starting up Streams instance 'streams@devuser...
CDISC0078I Total instance hosts: 1
CDISC0056I Starting a private Distributed Name Server (1 partitions, 1
replications) on host 'redbook8.mydomain.net'...
CDISC0057I The Streams instance's name service URL is set to
DN:redbook8.mydomain.net:48643.
CDISC0061I Starting the runtime on 1 management hosts in parallel...
CDISC0003I The Streams instance 'streams@devuser was started.
```

If you need to specify the exact host on which to create the instance, you can use the --hosts parameter to specify the host by name. The following commands create and start a single host private developer's instance on host named redbook14:

```
$ streamtool mkinstance -i streams@devuser --template developer --hosts
redbook14
CDISC0040I Creating Streams instance 'streams@devuser...
CDISC0001I The Streams instance 'streams@devuser was created.
$ streamtool startinstance -i streams@devuser
CDISC0059I Starting up Streams instance 'streams@devuser'...
CDISC0078I Total instance hosts: 1
CDISC0056I Starting a private Distributed Name Server (1 partitions, 1
replications) on host 'redbook14'...
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000196

```
CDISC0057I The Streams instance's name service URL is set to
DN:redbook14.mydomain.net:41656.
CDISC0061I Starting the runtime on 1 management hosts in parallel...
CDISC0003I The Streams instance 'streams@devuser was started.
```

Creating a multi-host private developer instance

In addition to single host instances, developers can create multi-host private instances as well. The --numhosts option allows the user to specify the number of hosts to be included in the instance. If this number is greater than 1, the instance will default to using a single reserved node for the management host, and the additional hosts will be application hosts. To maximize the number of application hosts, you can use the --unreserved option, which will place the host controller (hc) service on all nodes.

The following example demonstrates the creation of a multi-host private developer instance on four available hosts and configures them to all be used for running Streams applications. In addition, it shows the **streamtool lshosts command** that displays the host layout of the new instance:

```
$ streamtool mkinstance -i streams@devuser --template developer
--numhosts 4 --unreserved
CDISC0040I Creating Streams instance 'streams@devuser...
CDISC0001I The Streams instance 'streams@devuser was created.
$ streamtool lshosts -l -i streamsInstance: streams@devuser
Host                  Services                    Tags
redbook8              hc,aas,nsr,sam,sch,srm,sws
redbook9              hc
redbook10             hc
redbook11             hc
```

Note: If you specify a value for the --numhosts option that is greater than the number of available hosts, you will receive an error from the **streamtool mkinstance command**.

Creating a private developer instance using Streams Studio

In addition to the command-line interface, you can create instances from within Streams Studio. It is a best practice that shared instances be created using the command-line interface and that the scripts to create them are preserved. Developer instances, however, are good candidates for creation from within Streams Studio.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000197

In Figure 4-7, we select **Make Instance…** from the Instances folder within the Streams Explorer pane.

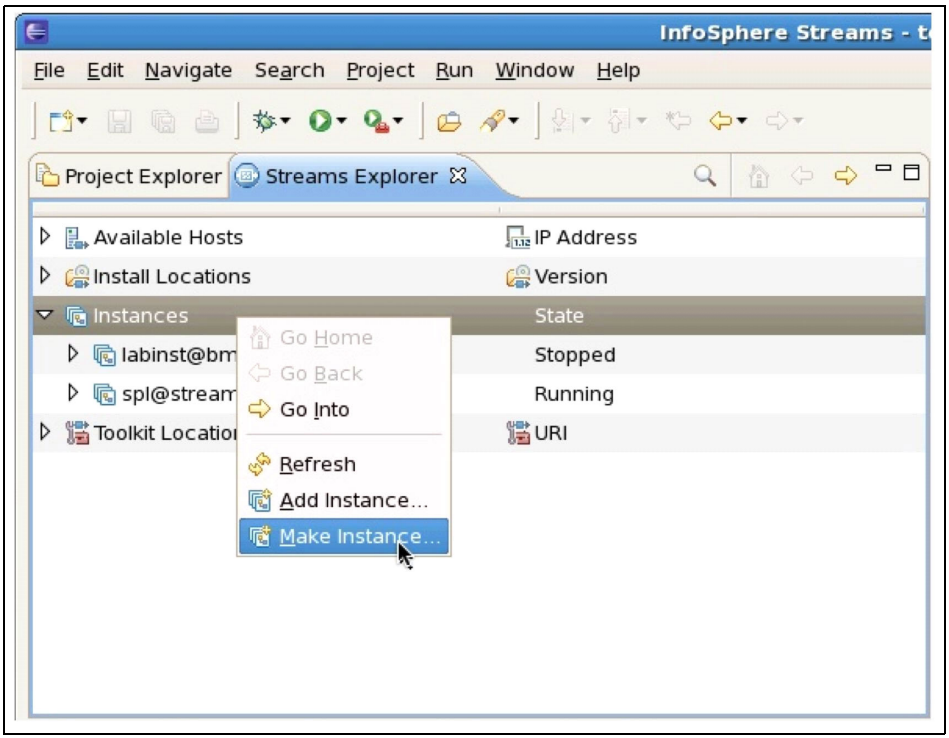


Figure 4-7 Streams Studio Make Instance menu

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000198

Figure 4-8 shows the Make Instance window. It is important to select the **Use template:** option and the **Browse** button to select the developer template (developer.template.properties).

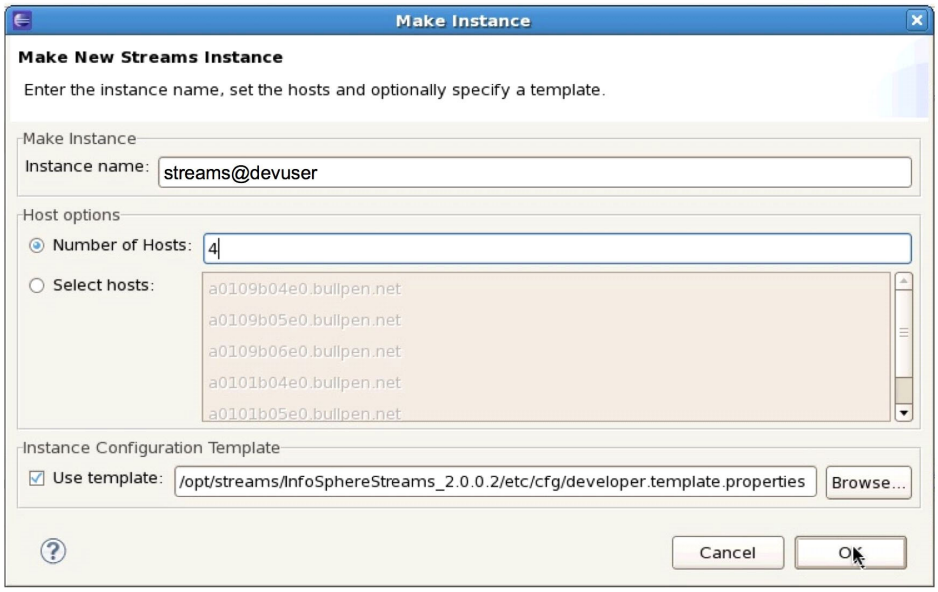


Figure 4-8 Streams Studio Make Instance window

After entering the instance name, number of hosts, and selecting the template, click the **OK** button. To start the instance, right-click the new instance within the Streams Explorer pane and select **Start Instance**.

At this point, you can use the instance from both Streams Studio and the command line.

4.5 Application deployment capabilities

In the first half of this chapter, we discussed Streams runtime deployment. We now focus on Streams application design and deployment. Having determined the topology, size, and layout of the runtime environment required for the projected applications (including their required volumes), you can determine how the processing should be divided across multiple hosts.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000199

Application deployment addresses the issue of how the components of the application are distributed and executed on the physical hosts. There are a number of ways that the application may be segmented and grouped to optimize throughput, latency and scalability:

- At development time
  - An application may be segmented into multiple applications. Then the data flowing through different streams within Streams may be exported from one application and imported into another application.
  - You may choose to deploy multiple copies of an operator, each working on a subset of the total stream. This allows parallel processing when the operator is processor intensive and cannot satisfy the throughput requirements on a single processor or host.
  - You may choose to segment processor-intensive analysis into multiple stages and then process the stream one stage after the other. This action allows pipelined processing and segmenting / distributing the processing load of the operators.
  - You may choose to control operator placement on the hosts of your instance. You can control operators that should be collocated, other operators that should not be collocated, and what set of hosts an operator may be allowed to run.
  - You may choose to manually combine multiple operators into a single Processing Element, which is called *fusing*. When operators are fused into PEs, they communicate through function calls and tuples are passed from one operator to the next using memory references, as opposed to using the transport (which may involve a transmission of the communication to take place across a network). Fusing can significantly reduce the cost of communication and improve both latency and throughput.
- At compile time

  Profile-driven fusion optimization is an approach in which the compiler will figure out how to best fuse operators into one or more PEs while respecting the user-specified constraints. This process is called *fusion optimization* and requires a profiling step where the application is first run in a profiling mode to characterize the resource usage with respect to CPU consumption and data traffic between each of the operators that make up the application.
- At run time
  - The Streams Application Manager (SAM) accepts requests to run applications. The SAM invokes the Scheduler service to determine which hosts the various PEs should run. The Scheduler then retrieves host information and performance metrics from the Streams Resource Manager, which enables the best placement of PEs to be determined dynamically at run time.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000200

– The PEs are deployed to the hosts as determined by the scheduler. The Host Controller on the hosts creates and manages a Processing Element Controller (PEC) to execute each PE.

The rest of this section discusses several of the application design and deployment capabilities that are available in InfoSphere Streams.

4.5.1 Dynamic application composition

The Streams run time allows you to run multiple applications in a Streams instance and export a stream from one application and import the stream into other applications. In Figure 4-9, we illustrate where the imagefeeder application (on the left) exports a stream of .jpeg images to the imagecluster, greyscaler, and imagewriter applications. In addition, this example illustrates streams data flowing from the imagecluster and greyscaler applications to the imagewriter application.

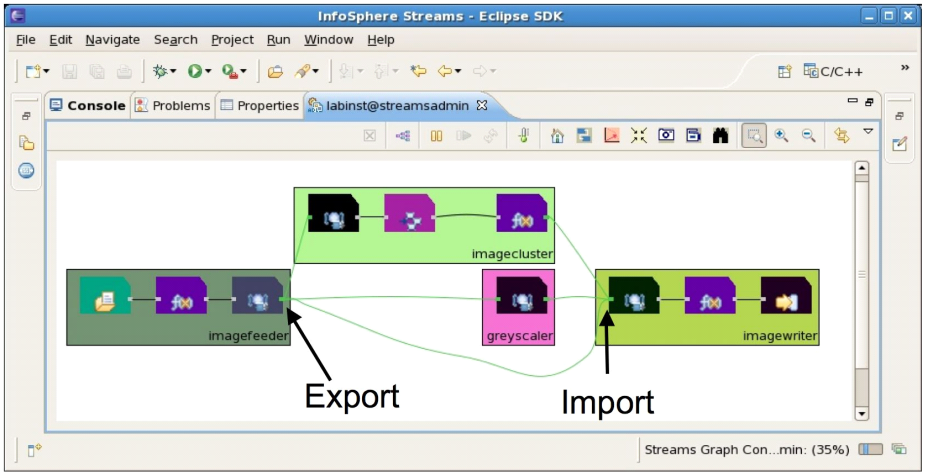


Figure 4-9 Application composition through Export and Import operators

The exporting and importing of streams between applications allows Streams administrators to use applications together to form larger integrated applications. On advantage of this approach is the ability to start and stop the individual applications independently without affecting the others. For example, you could choose to segment the Streams applications into three different categories, such as:

- Applications to read, validate and filter individual source streams.
- Applications to analyze a number of the source streams.
- Applications to format and deliver output data to sinks.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000201

This segmentation would allow Streams administrators to stop and start the source applications when the availability of Streams hosts varies. For example, you can run multiple analysis applications and develop and deploy new analysis algorithms without impacting the existing ones. You can also independently start and stop the sink applications to support new downstream applications to relieve load on the system when downstream applications are not active.

When exporting a stream from an application, you use the Export operator. When importing a stream into an application, you use the Import operator and then define which streams are being imported.

Exported streams and imported streams can be matched / connected in two different ways:

- Property-based application composition: This method uses a combination of stream properties that are set on the Export operator along with subscriptions specified on the Import operators. The middleware connects the imported and exported streams if the subscription predicate matches the exported stream properties and the stream schemas are compatible. If the Import predicate matches multiple streams exported by jobs running in the middleware, they are all connected. If there are no matching streams, nothing arrives at the Import operator. Properties and even subscriptions can be added, updated, or removed at run time.

  Example 4-8 shows property-based application composition:

Example 4-8 Property based composition

```
// imagefeeder application (partial listing)
composite imagefeeder {
  graph
    …
    stream<IplImage image,
           rstring filename,
           rstring directory> Img = ImageSource(Files) {…}

   () as ImagesExport = Export(Img) {
         param
           properties : { dataType = "IplImage",
                          writeImage = "true"};
    }
}
-------------------------------------------------------------
// imagewriter application (partial listing)
composite imagewriter {
  graph
    …
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000202

```
    stream<IplImage image,
            rstring filename,
            rstring directory> ImagesIn = Import() {
          param
            subscription : dataType == "IplImage" &&
                           writeImage == "true";
    }
    …
}
```

- Application name and stream ID composition: This method requires the importing application to specify the application name and a stream ID of the Export stream. The exporting application uses the streamId property of the Export operator to identify the stream. The importing application uses the applicationName and streamID parameters of the Import operator to identify the specific stream to be imported. If the exporting application is defined within a namespace, the importing application must fully qualify the application name using the namespace and the application name. This approach to job integration is less flexible than property-based composition.

  Example 4-9 shows application name and stream ID composition:

Example 4-9 Application name and stream ID composition

```
// SensorIngest application (partial listing)
namespace projectA.ingest;
composite SensorIngest {
   graph
    …
    stream<uint32 sensorId,
          timestamp sensorTime,
          blob sensorData> SensorData = Functor(DataIn){…}

    () as SensorExport = Export(SensorData) {
      param
          streamId: "SensorIngestData";
    }

}

-------------------------------------------------------------
// SensorProcess Application (partial listing)
namespace projectA.processing;
composite SensorProcess {
   graph
    stream<uint32 sensorId,
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000203

```
            timestamp sensorTime,
            blob sensorData> SensorData = Import(){
        param
          streamId : "SensorIngestData" ;
          applicationName : "projectA.ingest::SensorIngest";
      }
…

}
```

Note: If you export a stream from one application and it is not imported elsewhere, then the tuples in this streams are lost. They are not automatically persisted or buffered.

When you start an importing application, the contents of the stream are tapped from that point in time onwards. Likewise, if an importing application is stopped for a period of time and restarted, then the contents of the stream will be lost during the time the application is stopped.

4.5.2 Operator host placement

The Streams compiler and Scheduler are able to optimize your application by spreading the various operators (executed as processing elements known as PEs) across different hosts. By default, the Scheduler attempts to schedule hosts based on their current load.

In addition to automatic host placement, there is also the capability for you to manually allocate operators to specific hosts. You can also control which operators are placed on the same host, different hosts, and even specify which specific operators should be isolated and run on a dedicated host.

This section discusses the steps and artifacts involved in controlling operator host placement. For complete details, go to the following address:

http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp

Search for the section named SPL Config Reference.

Host tags

In “Defining the Streams available hosts” on page 166, we discussed host tags briefly, which included the two predefined host tags: build and execution. In addition to the predefined tags, the system allows you the ability to create your own tags and assign them to the hosts in your instance.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000204

The reason for defining host tags is to classify groups of hosts as having particular characteristics that a developer operator might need without specifying a specific host and restricting portability. Host tags are not required, but are a good idea if you are going to take manual control over operator placement.

In addition to assigning tags to hosts at instance creation, you can create host tags and assign them to hosts after the instance has been created, even while the instance is running. After an instance has been created, you use the **streamtool mkhosttag command** to define host tags. After defining tags, they can be assigned to hosts using the **streamtool chhost command**. Tags may be assigned to multiple hosts and a host may be assigned multiple tags.

Example 4-10 provides an example of defining, assigning, and displaying the host tags on a running instance.

Example 4-10 Host tags

```
$ export STREAMS_DEFAULT_IID=streams@streamsadmin
$ streamtool mkhosttag --description "Ingest Nodes" ingest
CDISC0102I Host tag 'ingest' has been added to instance
'streams@streamsadmin'.
$ streamtool mkhosttag --description "32GB RAM" himem
CDISC0102I Host tag 'himem' has been added to instance
'streamtool@streamsadmin'.
$ streamtool chhost --add --tags ingest,himem redbook2
CDISC0094I Adding host tags 'ingest,himem' to host 'redbook2' in
instance 'streams@streamsadmin'.
$ streamtool chhost --add --tags himem redbook3
CDISC0094I Adding host tags 'himem' to host 'redbook3' in instance
'streams@streamsadmin'.
$ streamtool lshosts -l
Instance: streams@streamsadmin
Host      Services                Tags
redbook1  aas,nsr,sam,sch,srm,sws  build,execution
redbook2  hc                      build,execution,himem,ingest
redbook3  hc                      build,execution,himem
redbook4  hc                      build,execution
redbook5  hc                      build,execution
redbook6  hc                      build,execution
redbook7  hc                      build,execution
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000205

Host pools

Host pools are a collection of hosts defined in a Streams application that can be used to support operator host placement. There is always a default pool available, which by default consists of the complete set of hosts that are available in the instance and have not been allocated to exclusive pools. The size of the default pool can be limited through the use of the defaultPoolSize config directive or the -d Streams compiler option. These are defined in the config clause on the main composite for the application.

Host pools can be defined as shared or exclusive. A pool is shared if it can contain hosts that also appear in other pools. Conversely, a pool is exclusive if it must not contain any hosts that appear in other pools.

The hosts that are included in each host pool can be selected at run time in one of the two following ways:

- Tagged pool: Hosts included in this type of pool are allocated at run time from a set of hosts that been assigned a specific host tag or set of hosts tags.
- Implicit Pool: Hosts are allocated at run time from the set of hosts available in the instance the application is being deployed to that are not allocated to an Exclusive pool.

Although not recommended, a host pool may have the set of hosts assigned to it specified at compile time. In this case, the pool is defined by specifying the set of hosts by name or IP address at compile time. This is not recommended because it hinders deployment activities by tying the application to specific hosts and requiring a recompile if host assignments need to be modified in the future.

Host pools can be defined by providing a size, in which case, only the specified number of hosts will be selected at run time to be part of the pool. For example, if an instance has four hosts with a host tag of ingest, but the application defines a tagged pool based on the ingest tag with a size of 2, it will only include two of the four tagged hosts in the host pool. If a size is not specified in this example, then all hosts with the tag of ingest would be included in the pool.

Example 4-11 illustrates several host pool examples.

Example 4-11 Host pool examples

```
composite Main {
  graph
    // operator invocations here
  config
    hostPool :
      // Tagged pool, shared, no size specified
      ingestPool=createPool({tags=["ingest"]}, Sys.Shared),
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000206

```
    // Tagged pool, exclusive, no size specified
    dbPool=createPool({tags["dbclient"], Sys.Exclusive),
    // Implicit pool, shared, sized
    computePool=createPool({size=2u}, Sys.Shared)
    // Explicit compile-time pool - NOT RECOMMENDED
    badPool = ["redbook2","10.5.5.10"];
```

Host placement configuration directives

SPL provides a config clause that allows developers to control the behavior and deployment of Streams operators. The placement operator config provides several subconfig options for controlling operator host placement.

There are two groups of host placement subconfigs:

- Absolute host location: This host subconfig specifies the absolute location (pool, IP address, or name) of the host on which the operator instance should run.
- Relative host constraint: These subconfigs constrain whether operator instances must run (hostColocation) or must not run (hostExlocation) on the same host, or whether they run in a partition that has a host of their own (hostIsolation).

placement : host

After one or more host pools have been defined, then you can specify if an operator instance should use a host from one of the host pools. If a runtime selected host pool is specified in the host config, then the actual host will be chosen at run time. This approach ensures maximum application portability. In addition to specifying a host pool, the host config can be used to specify an explicit host name or IP address, but this approach severely limits portability.

Example 4-12 illustrates examples of host config specifications.

Example 4-12 Host config specifications

```
composite Main {
  graph
    stream<int32 id> S1 = Beacon() {
      // System picks a host from the pool at runtime
      config placement : host (ingestPool);
    }
    stream<int32 id> S2 = Beacon() {
      // System picks host at a fixed offset from a pool
      config placement : host (ingestPool[1]);
    }
    stream<int32 id> S3 = Beacon() {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000207

```
      // System picks host by name !!Not Recommended!!
      config placement : host ("redbook1.mydomin.com");
    }
  config
    // Tagged pool, shared, no size specified
    hostPool :
      ingestPool=createPool({tags=["ingest"]}, Sys.Shared);
```

placement : hostColocation

The hostColocation placement config allows you to specify a colocation tag. All other operator invocations that use the same collocation tag will be run on the same host.

placement : hostExlocation

The hostExlocation placement config allows you to specify an exlocation tag. Any other operator invocations that use the same exlocation tag must be run on different hosts.

placement : hostIsolation

The hostIsolation placement config allows you to specify that an operator must be run in an operator partition that has a host of its own. Other operators can be run in the same partition (refer to 4.5.3, “Operator partitioning” on page 189), but no other partition can be run on that host.

Note: There is potential for someone to erroneously specify inconsistent or conflicting config statements, making it impossible to resolve them completely. For example, exlocation can conflict with colocation. In most cases, the compiler can detect such conflicts, but in some cases it cannot. When the compiler detects a conflict, it issues an error message. If the compiler cannot detect conflicts, they will be identified when the application is submitted to run on an instance, resulting in a `streamtool submitjob` error.

In Example 4-13, we demonstrate the use of relative operator placement. In this example, all three operators will be placed on hosts from the ingestPool. There are, however, constraints that will require at least two hosts to be available in the pool. The operators that produce streams S1 and S2 will be collocated, and the operator that produces stream S3 cannot be located on the same host as S1.

Example 4-13 Relative operator placement

```
composite Main {
  graph
    stream<int32 id> S1 = Beacon() {
      // System picks a host from the pool at runtime
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000208

```
      // and sets a colocation tag
      config placement : host (ingestPool),
                         hostColocation("withS1"),
                         hostExlocation("notWithS3");
    }
    stream<int32 id> S2 = Beacon() {
      // place this operator on the same host as the S1
      config placement : hostColocation("withS1");
    }
    stream<int32 id> S3 = Beacon() {
      // System picks host by name !!Not Recommended!!
      config placement : hostExlocation("notWithS3");
    }
 config
    // Tagged pool, shared, no size specified
    hostPool :
      ingestPool=createPool({tags=["ingest"]}, Sys.Shared);
```

4.5.3 Operator partitioning

You can fuse multiple operator instances in an SPL application into units called *partitions*. Note that the notion of a partition of operator instances in the flow graph is distinct from the notion of a partition in an SPL window clause; they are unrelated concepts. The execution container for a partition is a processing element (PE). At run time, there is a one-to-one correspondence between partitions and PEs, because each partition runs in exactly one PE. Fusion is important for performance, because PE-internal communication is faster than cross-PE communication. Conversely, multiple PEs can use the hardware resources of multiple hosts. By default, the optimizer does not perform fusion and places each operator instance in its own PE. Furthermore, users can control fusion with explicit partition placement constraints in the code, as shown in Example 4-14.

Example 4-14 Operator partitioning example

```
int32 foo(rstring s, int32 i) { /*do something expensive*/ return i; }
//1
composite Main {
graph
  stream<int32 i> In = Beacon(){}

  stream<int32 i> A1 = Functor(In) {
    output A1        : i = foo("A", i);
    config placement : partitionColocation("A");
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000209

```
    //fuse A1 into "A"
  }
  () as A2 = FileSink(A1) {
    param  file      : "A.txt";
    config placement : partitionColocation("A");
    //fuse A2 into "A"
  }
  stream<int32 i> B1 = Functor(In) {
    output B1        : i = foo("B", i);
    config placement : partitionColocation("B");
    //fuse B1 into "B"
  }
  () as B2 = FileSink(B1) {
    param  file      : "B.txt";
    config placement : partitionColocation("B");
    //fuse B2 into "B"
  }
}
```

4.5.4 Parallelizing operators

When you deploy your application on the runtime hardware, you may find that you have a single application operator, which is a performance bottleneck. This means that the operator cannot deliver the necessary throughput or perform all the required tasks within the required response time to satisfy the overall system performance requirements.

If you leave this operator unchanged, it will throttle the rate of throughput and require you to spread the workload over multiple hosts.

Parallelism of operators

One option to spread the workload is to segment the stream into a number of sub-streams, each of which can be processed in parallel on different host computers.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000210

Figure 4-10 shows this pattern.

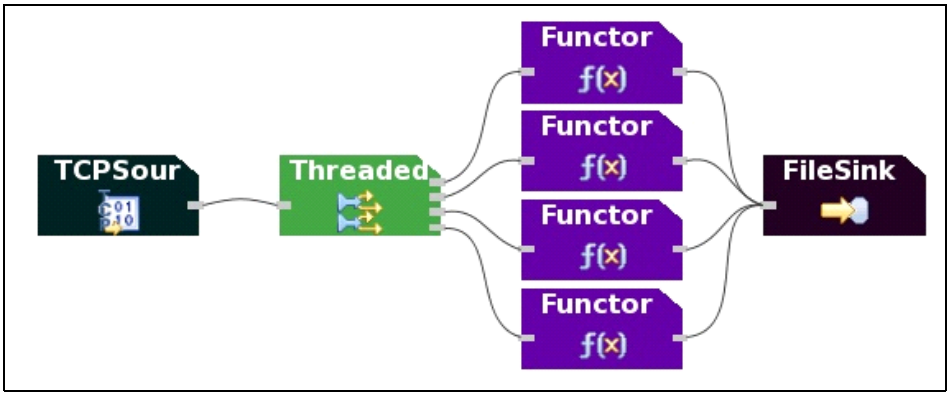


Figure 4-10 Parallel operator example

The problem of state

Stream computing works on the general principle that applications are stateless and that data passing through the system is not stored. However, a number of the Streams operators are not totally stateless, meaning they can be configured to keep state between subsequent tuples. When this is the case, splitting a stream for parallel processing can change the results of the operators.

The following list outlines the state management possibilities for the Streams operators:

- Tuple history: Expressions in parameters or output assignments of operator invocations can refer directly to tuples received in the past. History access maintains a list of tuples that can be subscripted by using the input port name.
- Operator windows: A window is a logical container for tuples recently received by an input port of an operator.
- Operator custom logic: The logic clause consists of local operator state that persists across operator firings and statements that execute when an input port receives a tuple or a punctuation and thus fires.
- Generic and Non-generic Primitive operators: User-written operators written in C++ and Java may have any number state mechanisms built into them. These are controlled by the operator developer and should be well documented.

You may want to segment a stream to distribute its workload, but be aware that this action can change the behavior of the stateful operators as they will now only receive a subset of the original stream.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000211

For example, consider a Join operator that joins data from multiple, high-volume streams. If these streams are segmented to distribute the processing load, the data items that would have matched in a single operator will be spread out over different join operators and the number of matches could be reduced.

Figure 4-11 shows an illustration of segmenting a Join operator:

- In the single join case, the operator processes three tuples (1,2,3) from both streams. Three matches will be output from the Join operator.
- In the split join case, the streams have been segmented and processed by three separate join operators. Only one of the tuple pairs will match in this case.
- In the partitioned join case, we have ensured that the streams are segmented to send related tuples to the same Join operator. All three matches will be identified and output in this case.

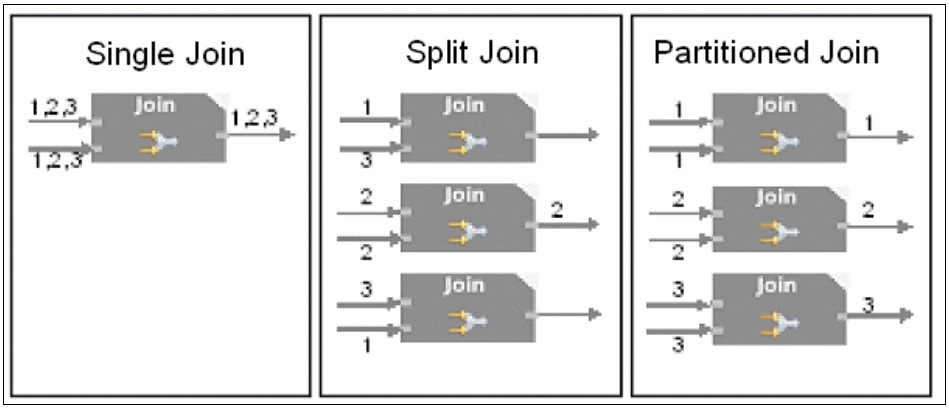


Figure 4-11 Parallel Join operator side effects

The use of partitioned windows can help when parallelizing operators. If you split the data using the windows partition key attribute, data will be kept together, preserving the integrity of partitioned window functions. For example:

- Consider the example of a stock trading application where you calculate the average price of a stock for each company in your portfolio. You can split the stock price stream by the stock symbol, as tuples will be kept together and your aggregate will be unaffected.
- In contrast, if you want to keep an average of all stock quotes together, segmenting the streams by the stock symbol will change the aggregate calculation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000212

Pipelining as an alternative

An alternative to splitting a stream and processing in parallel is to use the pipeline pattern. With this option, processing is divided into multiple stages, which can be performed one after another.

Note that pipelining has larger communication requirements than parallelism. There is a processing impact and a time penalty in communicating between processing elements, especially when the PEs are on different hosts.

In the case of pipelining, every tuple is sent to every operator, so if you have five separate operators pipelined together, and each operator is executing on separate hosts, you have to send the tuples across all five hosts.

In contrast, if you segment a stream into five parallel operators, then each tuple only needs to be sent between hosts twice (once to the Parallel operator and then back again), but you incur a processing impact when segmenting the streams.

Parallel pipelining

The ultimate solution for many applications facing a performance bottleneck is to combine parallel processing with pipeline processing, resulting in parallel pipelines, as shown in Figure 4-12.

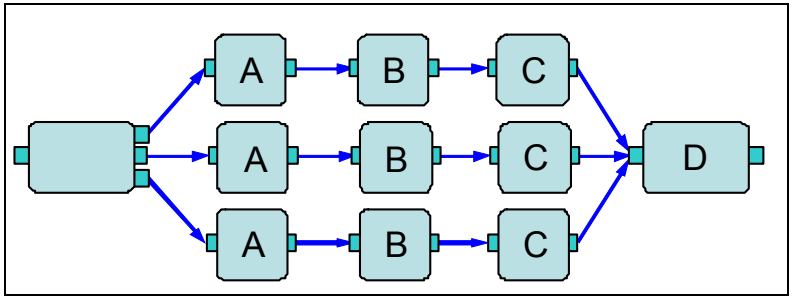


Figure 4-12 Parallel pipelining

The benefits of combining these two patters include:

- Tuples that require more processing (due to the size or complexity of the data) may slow down one pipeline. However, the other pipelines will still be processing tuples. This allows the faster tuples to pass the slower tuple as opposed to causing other tuples to clog up within the system.
- Each instantiation of the pipeline can be fused or at a minimum co-located on the same host, reducing network impact for each tuple. The set of instantiations of the pipeline can be distributed across a set of hardware, thus taking advantage of available resources.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000213

4.6 Failover, availability, and recovery

InfoSphere Streams provides a number of facilities to allow recovery from failed hardware and software components. Support for the following failure recovery situations is provided:

- The restarting and relocating of processing elements
- The recovery of failed application hosts (hosts running only the host controller and PEs)
- The recovery of management hosts (a host running any of the management services: SRM, SAM, SCH, AAS or NSR)

Recovery of application hosts and processing elements (PEs) do not require any additional instance configuration, but the extent to which processing elements can be recovered does depend on how they are configured within an SPL application.

Recovery of management hosts requires a recovery database (DB2 is required) and two changes to the instance configuration, that is, the recovery mode must be set to true, and you must use a file-system-based name service (NSR).

These recovery situations are discussed in the following sections.

4.6.1 Restarting and relocating processing elements

Recovery and relocation of stream processing applications has a few factors that can limit and even prevent recoverability and relocation, especially in a multi-host topology. For example, if operator A adds a counter to tuples that it processes, and operator B, which is down stream, depends on those counters continuing to increase, a restart of operator A that does not continue where it left off could adversely impact the logic and integrity of your application results. In another case, if operator C reads and writes information to a local (non-shared) file system, and Streams were to restart that operator on a different host, the file system may not exist or may not be in a configuration compatible with the operator. Therefore, this situation could cause further failure or invalid behavior.

The Streams runtime engine does not have insight into the internal workings of each operator, and it does not understand the dependencies between operators that have been built into your application (beyond the operator-to-stream connectivity). For this reason, the Streams run time takes a conservative view of processing-element recovery and relocation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000214

By default, Streams assumes that an operator (and thus the processing element that contains it) cannot be restarted or relocated. It is up to you, the application developer, to provide configuration directives that tell Streams which operators can be restarted and which can be relocated.

To ensure that Streams can automatically restart restartable and relocatable PEs on a different host after a host failure, do not specify specific host names in the placement configuration section of your operators.

Restartable operator configuration

An operator instance can be restarted after a user-requested **`streamtool stopPE command`** completes or after a partition crashes due to a runtime error if it is marked as restartable in its config clause. In Example 4-15, we provide an example of a TCPSource operator that is being used in the role of a TCP server. We can mark this operator restartable because it does not contain a state, but we may not be able to make it relocatable because TCP clients may not be able to find it if it is restarted on a different host.

Example 4-15 Restartable example

```
stream<unit32 empid, float64 rate, float64 hours> payroll =
   TCPSource(){
   param
      role : server ;
      port : "55555" ;
      format : csv ;
   config
      restartable : true;
      relocatable : false;   // default
}
```

Relocatable operator configuration

An operator instance can be relocated on a different host, either after a failure or for better load balancing, based on relocation recommendations from the scheduler. The relocatable config implies the restartable config, but you may want to specify both for clarity.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000215

In Example 4-16, we provide an example of a TCPSource that is being used in the role of a TCP client. This operator can be both restarted and relocated without causing our application any problems. If, however, the network configuration of the environment has limited connectivity (through a firewall) to the TCP server and a limited number of hosts, you may not be able to relocate operators within this environment.

Example 4-16 Relocatable example

```
stream<uint32 empid, float64 rate, float64 hours> payroll =
   TCPSource(){
   param
      role : client ;
      address : "HRServer1.mycompany.com" ;
      port : "55555" ;
      format : csv ;
   config
      restartable : true ; // implied by relocatable : true
      relocatable : true;
}
```

Note: If an operator A is restartable and operator B is not, then A and B are partition-exlocated (meaning they cannot be in the same partition), unless the compiler option -O, --setRelaxFusionRelocatabilityRestartability is used.

Manual and automatic restart

Automatic restarting of processing elements are governed by the following three rules:

- Processing elements marked as restartable (but not relocatable) will be automatically restarted when they fail or crash if the application host they were running on is available.
- Processing elements marked as restarted and relocatable will be automatically restarted on the same host or a different host when they fail or crash.
- Processing elements not marked as restartable can only be restarted by canceling and resubmitting the application as a new job. If these types of PEs fail, the entire application must be cancelled and restarted.

Note: When a processing element or a Streams Job is restarted, entries in the data streams may be missed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000216

In addition to automatic restart due to a failure, you can stop and restart processing elements manually. The automatic restart rules are still enforced (for example, you cannot restart a processing element manually that is not configured to be restartable). Manual processing element control can be performed using both the **streamtool** command and the Streams administration console. The following sub-commands are used through the **streamtool** command:

- **streamtool stoppe** is used to stop a specified processing element.
- **streamtool restartpe** is used to start a specified processing element on the same host or a different host.
- **streamtool lspes** is used to list the processing elements in the instance and is used to view their health and host assignments.

4.6.2 Recovering application hosts

Application hosts are hosts that are configured to run the Streams Host Controller (HC) service. The information in this section applies to Mixed hosts as well, because they also run a host controller. As discussed previously, the host controller service controls the execution of processing elements on a Streams host. If the host controller fails or stops responding, the SRM flags that the host cannot be scheduled. After a short period of time, the SAM changes the state of all PEs running on the failed host to Unknown. For example, if the state is *Running* when the HC fails, the state becomes *Unknown*.

The status of hosts in a running instance can be viewed using the **streamtool getresourcestate command**. The Schedulable column may contain the following values:

- A value of yes indicates that the host is available to run PEs.
- A value of “no” (and the *reason text* that is shown here) indicates that the HC service on the host is not functional and that any PEs on that host are in the Unknown state. The *reason text* indicates why the host is not available to run PEs.
- A dash (-) indicates that there is no HC service running or planned on the host.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000217

Figure 4-13 shows the results of the **streamtool getresourcestate command** after a host controller has failed on the specific host named *redbook6*.

```
$ streamtool getresourcestate
Instance: streams@streamsadmin Started: yes State: PARTIALLY_FAILED
Hosts: 7 (1/1 Management, 5/6 Application)
Host            Status    Schedulable  Services
redbook1        RUNNING   -            RUNNING:aas,nsr,sam,sch,srm,sws
redbook2        RUNNING   yes          RUNNING:hc
redbook3        RUNNING   yes          RUNNING:hc
redbook4        RUNNING   yes          RUNNING:hc
redbook5        RUNNING   yes          RUNNING:hc
redbook6        FAILED    no(failed)   FAILED:hc
redbook7        RUNNING   yes          RUNNING:hc
```

Figure 4-13 Failed application host

PEs in the *Unknown* state cannot be restarted automatically or with the **streamtool restartpe command**. However, they can be stopped with the **streamtool stoppe command**, but do not move to the Stopped state until the problem with the host controller has been resolved.

To repair a host that has a value of no in the Schedulable column, the administrator can perform the following actions:

- Restart the HC (**streamtool restartservice**): This command will attempt to restart the host controller service on the specified host. If the host controller does start, any PEs running on that host will move from the Unknown state to the Running state if they had not also failed.
- Quiesce the host (**streamtool quiecehost**): This command can only be performed if the host is not running a Streams management service. Quiesce performs a controlled shutdown of the PEs and the HC on a host. PEs will be relocated if possible. The host remains in the instance's configuration.
- Remove the host (**streamtool rmhost**): This command can only be performed if the host is not running a Streams management service. The command here performs a controlled shutdown of the PEs and the HC on a host. PEs will be relocated if possible. The host is removed from the instance's configuration.
- Remove the HC from the host (**streamtool rmservice**): This command will remove the host control (HC) from a host, which will prevent the scheduler from running PEs on that host. The PEs on that host will be relocated if possible. The Schedulable column shown as a result of the **streamtool getresourcestate command** will contain a dash (-).

In the last three cases, SAM will attempt to move restartable and relocatable PEs to other hosts. If SAM cannot move the PEs to other hosts, the PEs will remain in the Stopped state.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000218

4.6.3 Recovering management hosts

The default behavior of a Streams instance is to not allow the recovery of management services. In this default mode of behavior, if one of the instance's management services or hosts fails, you must restart your instance to restore it to a fully operational state.

One exception to this default behavior is the Streams Web Service (SWS). This service can be stopped or restarted without affecting the Streams instance or requiring special instance configurations.

To enable the recovery of any failed management services, Streams can work with an IBM DB2 database server to store the runtime data in a recovery database. For example, if a host is running the Streams Application Manager (SAM) server and the host crashes, the Streams instance administrator can restart the SAM server on another cluster host and the Streams run time will restore its state from the values stored within the recovery database.

Streams recovery database

The Streams recovery database is a DB2 database that records the state of an instance's services. If instance services fail, Streams restarts them and restores their state by retrieving the appropriate state data from the recovery database.

In addition, the Streams recovery database records information that enables the Streams instance administrator to manually restart the following management services, move these management services to another host, or both:

- The Streams Resource Manager (SRM)
- The Streams Application Manager (SAM)
- The Authentication and Authorization Service (AAS)

The information that is stored in the recovery database includes the following states for an instance and its applications:

- Job state
- Processing elements (PE) placement
- Stream connection state information
- Instance resource state (instance state, instance daemon placement, and state)
- Authentication and authorization state information

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000219

A steady state is when all jobs are submitted and running and there are no changes to the state of the instance. The recovery database is not active when an instance is in the steady state. In addition, the recovery database is unaffected by the activity of Streams applications. This action prevents the configuration of failover and recovery from impacting the performance of Streams applications.

For details about configuring the Streams Recovery Database, see the *Streams Installation and Administration Guide*.

RecoveryMode instance property

In order to configure your instance to make use of the Streams Recovery Database, you must set the Streams instance property *RecoveryMode* to *on*. The following command will accomplish this task:

```
$ streamtool setproperty RecoveryMode=on
```

Note: If the instance is running when you set the RecoveryMode to on, you must restart the instance for the change to take effect.

File system based name service

By default, the NameServiceUrl property for a Streams instance is set to *DN:*, meaning a distributed name service is used. A distributed name service does not support a Streams recovery database. In contrast, it is a good idea to use a file system based name service when the recovery of management services is critical. To use a file system based name service, set the NameServiceUrl property to FS:.

Recovery process

If your Streams instance is configured with the RecoveryMode property set to on and you have configured a recovery database on one of your cluster's hosts, individual Streams management components can be recovered if they fail.

If more than one of these management services fail, restart them in the following order to ensure that any prerequisite service is already restarted:

1. SRM
2. AAS
3. SCH
4. SAM
5. SWS

You can view the status of all services using the **`streamtool getresourcestate command`**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000220

To restart a service on the configured node, use the **streamtool restartservice command**. For example, to restart the SAM service, enter the following command:

```
$ streamtool restartservice sam
```

To move a service from one node to another node within the instance, use the **streamtool addservice command**. For example, to move the SAM service to the host named *hostname*, enter the following command:

```
$ streamtool addservice --host hostname sam
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000221

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000222

5

Streams Processing Language

In the previous chapters, we have described Streams positioning, concepts, process architecture, and examples of the types of problems that can be solved by Streams.

In this chapter, we describe the language that is used to develop Streams applications and provide code examples related to real-world scenarios. More advanced features of the language, such as user-defined operators, are covered in Chapter 6, “Advanced Streams programming” on page 277.

To learn the intricate and extensive details about the Streams Processing Language, refer to the detailed set of documentation files shipped with the InfoSphere Streams product.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000223

5.1 Language elements

In this section, we describe the language elements that make up a Streams application. As a means to describe the elements of the language, we introduce a simplified example from within a mobile telecommunications company.

In this example, a mobile phone operating company is concerned about how their wireless infrastructure is performing. Customers have been complaining of calls dropping or the customer's lack of ability to make calls reliably. Consequently, the company is concerned that issues with cell broadcast masts may be impacting service revenues and customer satisfaction. Therefore, the company has built a Streams application to monitor calls and perform call re-routing around defective masts.

In Figure 5-1, we show an example Streams data flow for a mobile call re-routing application. Note that it consists of a number of operators, sources, sinks, and named flows (streams).

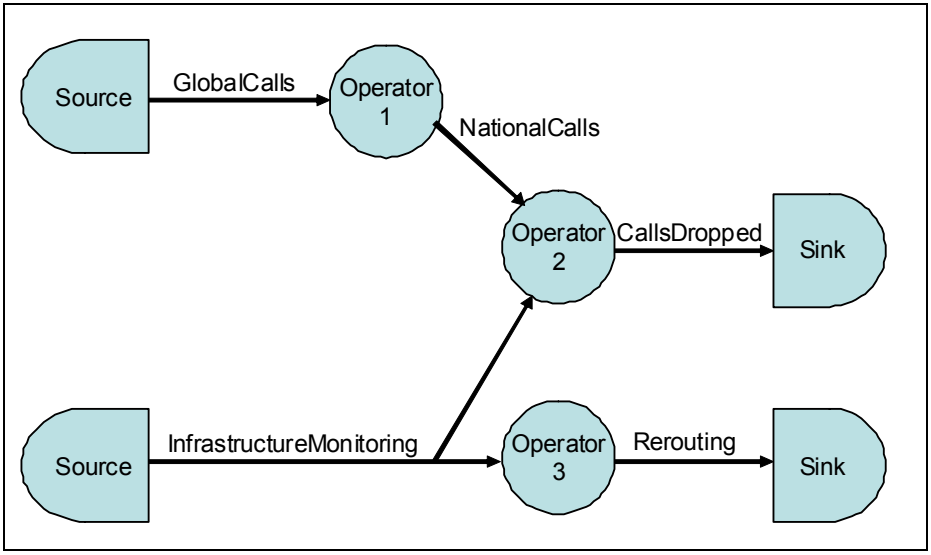


Figure 5-1 Example data flow for a mobile call re-routing application

The sources and sinks shown in the diagram are used by a Streams application to connect to outside data sources; they are the boundaries of the application as far as we are concerned. The Source operator reads information from an external input and the Sink operator writes the data to the external destination.

This Streams application is designed to receive real-time data feeds that are read by the two Source operators shown on the left side of the flow diagram.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000224

The input streams to the application provide:

- Data related to the calls being made by the company's customers in real-time (GlobalCalls)
- Real-time data related to the health of the company's transmission masts (InfrastructureMonitoring)

The customer call data stream is named GlobalCalls because that stream contains call data for calls irrespective of whether the customer is making the call while in the customer's home country or while abroad. Mast failure information, however, is available only from the masts located in the company's home country.

To avoid the impact of processing all the tuples in the GlobalCalls stream further on in the flow, the Streams application filters the GlobalCalls stream, removing any data from calls made on other countries' mast infrastructure. This filtering is performed in Operator 1.

It is not always possible to determine a call termination reason from the call tuples alone; the customer might have ended the call normally or the call might have been ended due to being interrupted by a fault. For example, a fault could have come from the mast being used while the call is active. Operator 2 correlates the call data coming in through the National Calls stream with the mast-health-monitoring information in the InfrastructureMonitoring stream to determine which calls were terminated specifically due to an infrastructure issue.

Operator 3 contains an internal re-routing table and, on the basis of failures, sends re-routing commands to the lower output sink.

In the Streams Processing Language, the flow described above would be expressed at a high level, as shown in Example 5-1.

Example 5-1 Expressing the data flow in Streams Processing Language

```
stream <...> GlobalCalls = TCPSource() {...}
stream <...> NationalCalls = operator1(GlobalCalls) {...}
stream <...> InfrastructureMonitoring = TCPSource() {...}
stream <...> CallsDropped = operator2(NationalCalls;
InfrastructureMonitoring) {...}
stream <...> Rerouting = operator3(InfrastructureMonitoring) {...}
() as CallsDroppedSink = FileSink(CallsDropped) {...}
() as ReroutingSink = FileSink(Rerouting) {...}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000225

Note that each stream is first defined on the left side of an assignment (an assignment is represented by the equals sign, “=”, in Streams). Immediately following the “=” assignment is the operator name responsible for producing that stream.

Also note that both the source and sinks are also considered operators. Sources read from external sources and convert the data from an external representation to the Streams representation. Similarly, sinks do the reverse, meaning that the Streams internal format is converted to an external representation within a sink.

In Figure 5-1 on page 204, there is no stream shown beyond the sink; processing that occurs outside the source and sink is outside the scope of the Streams Processing Language.

The end of a flow is denoted with the “()" code syntax (instead of a named stream), as shown in the first two characters of the last line of Example 5-1 on page 205.

Note that within the parentheses, immediately following the operator name, is the list of one or more streams that are inputs to that operator. For example, in the sample code shown, 'GlobalCalls' is the input stream to operator1.

Also, note that the code example contains two types of braces: “() and {}”. The contents of the braces will be elaborated later in this chapter, but at a high level might contain stream metadata, operator input arguments, operator parameters, and operator output attribute derivation expressions.

5.1.1 Structure of an SPL program file

Streams Processing Language files (source files) are ASCII text files that commonly have a “.spl” extension. As text files, they can be written and edited using any text editor of your choice, but most commonly are developed and maintained using the InfoSphere Streams Studio editor. Source files are compiled into binary code files using the supplied Streams compiler.

Tip: If you prefer to use a text editor rather than the Streams Studio to edit your Streams source files, syntax highlighters for vi, JEdit, and GNU emacs text editors are provided with the software.

Each `.spl` file commonly expresses a flow from source(s) to sink(s) through one or more operators connected by named streams.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000226

The Streams Processing Language supports comments. Single line comments are prefixed by the "//" character string combination. Anything after these characters, meaning from these characters to the end of the line, is ignored by the compiler. If comments are required to span multiple lines, the start of the comment section should be preceded by "/*" and ended with "*/".

Each source file is composed of up to six sections, of which two are mandatory. The six sections that are headed using square braces as follows:

1. Namespace
2. Use
3. Composite
4. Type
5. Graph
6. Config

The composite and graph sections are mandatory. As such, they are described together. The other sections are optional and will be discussed subsequently.

- Composite and graph

  A simple `.spl` file containing the mandatory sections is shown in Example 5-2.

  Example 5-2 A sample .spl file for the re-routing application

```
// Call Re-routing application
composite Routing
   graph
      stream <...> GlobalCalls = TCPSource() {...}
      stream <...> NationalCalls = operator1(GlobalCalls) {...}
      stream <...> InfrastructureMonitoring = TCPSource() {...}
      stream <...> CallsDropped = operator2(NationalCalls;
InfrastructureMonitoring) {...}
      stream <...> Rerouting = operator3(InfrastructureMonitoring)
{...}
      () as CallsDroppedSink = FileSink(CallsDropped) {...}
      () as ReroutingSink = FileSink(Rerouting) {...}
```

  Following the composite keyword is the name of the application (Routing in this example) which is followed by a keyword that indicates the beginning of the application sub-graph.

  Following the keyword of `graph` is the body of the code. In this case, we have inserted the code from the prior example (which is incomplete at this stage, as evidenced by the ellipses).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000227

- Namespace

  At the top of the `.spl` program file, the developer can specify an optional namespace keyword followed by a namespace for the composite, as shown in Example 5-3. The presence of a namespace allows the application developers to avoid collisions between two different SPL composites that carry the same name.

  Example 5-3 Namespace

```
namespace com.ibm.sample.regex;
```

- Use

  Following the optional namespace keyword, developers can optionally declare other composites and types that are used within this `.spl` file. The “Use” section allows the developers to improve the code readability by not expanding the fully qualified name for the external types and composites that are used in the `.spl` file. As an example, a developer could define a “use” statement to indicate that an external composite will be used later in the code section of this `.spl` file, as shown in Example 5-4.

  Example 5-4 Use

```
use
com.ibm.streams.myUtils::FileWriter;
```

- Type

  Inside a composite, developers can optionally declare all the different data types that will get used in the SPL application. In this section, user-defined types, constants, and full-fledged tuple types can all be defined. Example 5-5 shows how such types can be defined.

  Example 5-5 Type

```
type
   employeeSchema = tuple<rstring name, uint32 id, uint16 age,
float32 salary, rstring location>;
   myIntegers = list<int32>;
   expression <rstring> $RSTRING_FILE_MODE : "w+";
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000228

- Config

  The SPL language allows for an extensive set of configuration directives that can either be applied at the individual operator level or at the composite level. These configuration directives range in functionality from assigning a default node pool size, providing a list of host pools, setting the level of logging, determining whether a PE should be restarted or not, establishing a threaded port for an input port with an optional queue size, fusing multiple operators, co-locating multiple operators on a specific host, ensuring two operators are not placed on a same host, and so on. Some of these SPL configuration directives are shown in Example 5-6.

Example 5-6 Using SPL configuration directives

```
config
   logLevel: info; // Log level verbosity increases in this order':
error, info, debug, trace
   placement: host("myhost.acme.com");
   restartable: true;
   relocatable: true;
   threadedPort: queue(Port1, Sys.Wait, 1500);
```

5.1.2 Streams data types

In this section, we describe the data types supported by Streams.

A stream has an associated schema, which is the particular set of data types that it contains.

A schema consists of one or more attributes. The type of each attribute in the schema may be selected from one of the following high-level categories:

- Primitive types
- Composite types

These categories are further described below:

- Primitive types

  The following primitive types are supported:

  - boolean: True or false
  - enum: User-defined enumeration of identifiers
  - intb: Signed b-bit integer (int8, int16, int32, and int64)
  - uintb: Unsigned b-bite integer (uint8, uint16, uint32, and uint64)
  - floatb: b-bit floating point number (float32 and float64)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000229

- decimalb: Decimal b-bit floating point number (decimal32, decimal64, and decimal128)
- complexb: b-bit complex number (complex32 and complex64)
- timestamp: Point in time, with nanosecond precision
- rstring: String of raw 8-bit characters
- ustring: String of UTF-16 Unicode characters
- blob: Sequence of raw bytes
- string[n]: Bounded length string of at most n raw 8-bit characters

► Composite types

InfoSphere Streams provides two kinds of composite types. They are called built-in collection types and tuple types.

The following are the three built-in collections, which can be either bounded or unbounded, making a total of six actual types as follows:

- list<T>: A list of random access zero-indexed sequence of SPL types
- set<T>: Unordered collection without duplicates
- map<K,V>: Unordered mapping from key type K to value type V
- list<T>[n]: list<T> with a bounded length
- set<T>[n]: set<T> with a bounded length
- map<K,V>[n]: map<K,V> with a bounded length

Another kind of composite type is a tuple, which is a sequence of attributes also known as named value pairs. For example, the tuple {name=”aname”, age=21} has two attributes, specifically name=”aname” and age=21. The “type” for this tuple is tuple<rstring name, uint32 age>. Tuples are similar to database rows in that they have a sequence of named and typed attributes. Tuples differ from objects in Java or C++ in that they do not have methods. You can access the attributes of a tuple with dot notation. For example, given a tuple t of type tuple<rstring name, uint32 age>, the expression t.name yields the name attribute's value. Attributes within a tuple type are ordered.

5.1.3 Stream schemas

The data types may be grouped into a schema. The schema of a stream is the name that is given to the definition of the structure of data that is flowing through that stream.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000230

For the simplified call routing application, the GlobalCalls schema contains the following attributes:

- callerNumber: uint32, // the customer number
- callerCountry: uint16, // the country code where the customer is located
- startTimeStampGMT: rstring, // call start time
- endTimeStampGMT: rstring, // call end time
- mastIDs: list<uint32> // list of mast(s) used to make the call

Note that mastIDs is a list<uint32>attribute. In the example, it is assumed that each mast in the network has a unique numeric identifier. The mastIDs attribute contains a list of the mast identifiers associated with the set of masts used to process the customer's call.

Within the Streams application code, the schema of a stream can be referred to in several ways. Consider the following:

- The schema may be explicitly listed.

  The schema can be fully defined within the operator output stream parameters, as in schema fully defined in operator output stream, which shows the GlobalCalls schema, as shown in Example 5-7.

  Example 5-7 Schema fully defined in the operator output stream

```
stream <int32 callerNumber, int16 callerCountry, rstring
startTimeStampGMT, rstring endTimeStampGMT, list<uint32> mastIDS>
GlobalCalls = TCPSource() {...}
```

- The schema may reference a previously defined type.

  The schema definition can be created ahead of the operator definitions in the type section of the SPL program file. In the type section, Streams developers can associate a name with that schema and subsequently refer to that name within the code rather than the entire definition in full each time, as shown in Example 5-36 on page 258.

  In Example 5-8, the schema is given the name callDataTuples, which is then referred to in the definition of the GlobalCalls. Note that it is also possible to define a schema by referencing a previously defined stream.

  Example 5-8 Defining the tuple in the type section

```
callDataTuples = tuple<int32 callerNumber, int16 callerCountry,
rstring startTimeStampGMT, rstring endTimeStampGMT, list<uint32>
mastIDS>;

..
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000231

```
..
stream <callDataTuples> GlobalCalls = TCPSource(...){...}
```

- The schema may be defined using a combination of the previously described methods.

  The last variant is that a schema for a stream may be defined by a combination of the methods, as shown in Example 5-9. In this example, another attribute is being added to the list of attributes already present.

  Example 5-9 Defining a stream schema using a combination of methods

```
type
callDataTuples = tuple<int32 callerNumber, int16 callerCountry,
rstring startTimeStampGMT, rstring endTimeStampGMT, list<uint32>
mastIDS>;

graph
stream <callDataTuples, tuple<int32 anotherAttribute>> = ...
```

  Where required, streams of the same schema may be collected together on a single input port of any operator.

5.1.4 Streams punctuation markers

InfoSphere Streams supports the concept of punctuation marks, which are markers inserted into a stream. These markers are designed in such a way that they can never be misinterpreted by streams as tuple data. Some Streams operators can be triggered to process groups of tuples based on punctuation boundaries or they can insert new punctuation marks into their output stream(s).

5.1.5 Streams windows

In previous chapters, we have described the need for accessing data through “windows” on the stream. Conceptually, we are working on streams of data that have no beginning or end. However, the streams operator's processing might need to group sections of the data stream within similar attributes or time intervals. The Streams Processing Language enables this type of grouping using a feature called windows.

Streams can be consumed by operators either on a tuple-by-tuple basis or through windows that create logical groupings of tuples.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000232

Windows are associated and defined on the input streams flowing into particular operators. If an operator supports windows, each stream input to that operator may have window characteristics defined. Not all Streams operators support windows; this is dependant on the operator's functionality. (For details about the windowing support in Streams, read the “Window Handling” chapter in the `SPLToolkitDevelopmentReference.pdf` file that is shipped with the Streams product.)

The characteristics of windows are as follows:

- The type of window
- The window eviction policy
- The window trigger policy
- The effect of adding the partitioned keyword on the window

These characteristics are described in the following sections:

- The type of window

  Two window types are supported, namely tumbling windows and sliding windows.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000233

– Tumbling windows

In tumbling windows, after the trigger policy criteria is met, the tuples in the window are collectively processed by the operator and then the entire window contents are discarded, as shown in Figure 5-2.

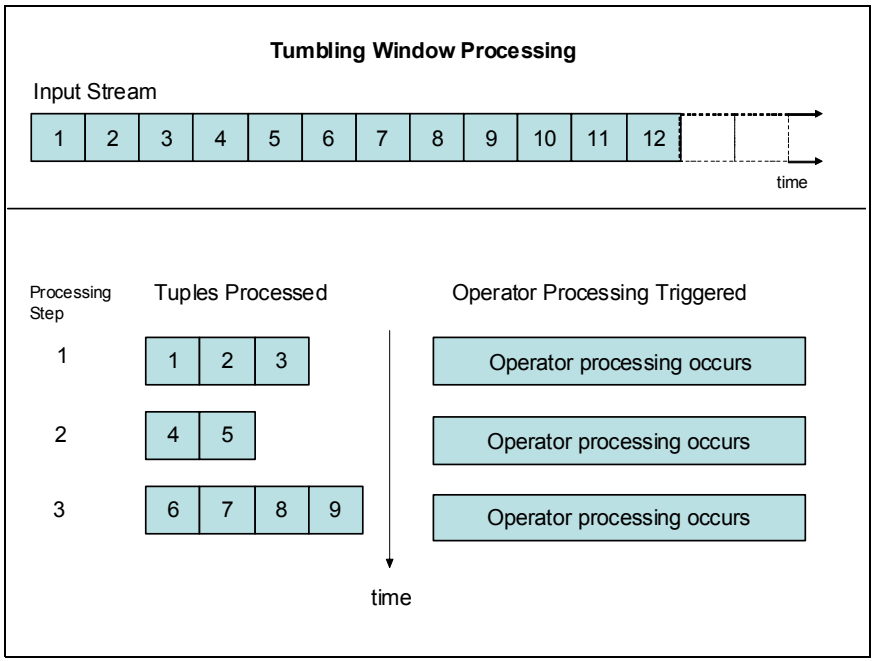


Figure 5-2 Tumbling windows

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000234

– Sliding windows

  In sliding windows, newly arrived tuples cause older tuples to be evicted if the window is full. In this case the same tuple(s) can be present for multiple processing steps, as shown in Figure 5-3. In contrast to tumbling windows, with sliding windows any given tuple may be processed by an operator more than once.

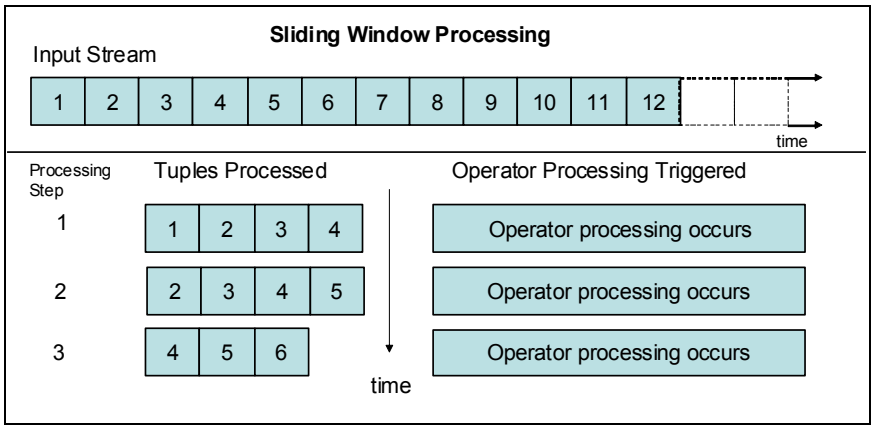


Figure 5-3 Sliding windows

- The window eviction policy

  The eviction policy is what defines when tuples are removed from the window. The eviction policy may be specified, after a selected numeric attribute increases by an amount beyond a threshold, using the following criteria:

  – Quantity of tuple

  – Age of tuples

  – Punctuation marks (special tuple separators understood by Streams)

- The window trigger policy

  The window trigger policy defines when the operator associated with the window performs its function on the incoming data. For example, a trigger policy can be based on a period of time or a number of tuples.

- The effect of adding the “partitioned” keyword on the window

  For supported operators, the use of the optional partitioned keyword will cause multiple window instances to be created for the stream in question. The number of instances will depend on the number of distinct values for the selected attributes, meaning the runtime values of the attributes used to define the group. This is depicted in the use of the partitioned keyword.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000235

As the application runs and new combinations of attribute key values are received, additional window instances are created dynamically. Each window instance on the stream will operate, as described above, independently of each other within the common set of characteristics, as shown in Figure 5-4.

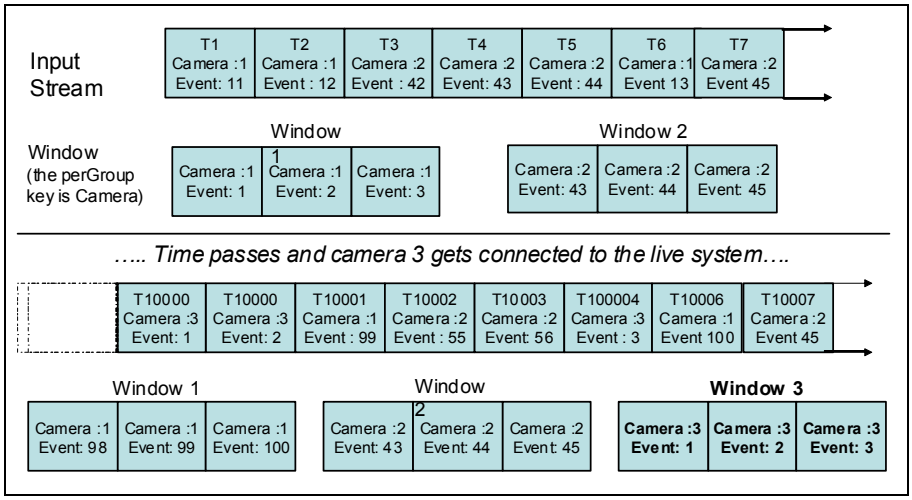


Figure 5-4 Partitioned keyword

Referring to Figure 5-4, InfoSphere Streams is being used as part of a security monitoring solution. The system has motion-sensitive security cameras that send data through captured image events when a security event triggers them. Sliding windows have been defined that have an eviction policy using a counter limit of three tuples. Therefore, in this case, the latest three events will be present in each window. The partitioned keyword has been used and the “group key” is the camera ID, which means that the number of physical cameras does not need to be hard coded anywhere in the application. Rather, the application will dynamically create additional windows as cameras are added and they start sending security event data.

Language syntax for windows

For operators that support windows, and where windows are required by the developer, the language requires the definition of the window characteristics be specified under the window clause for an operator. The window definition is placed after the relevant stream name, as shown in Example 5-10.

Example 5-10 Defining a group of sliding windows

```
stream<BeatL, BeatR> Join1 = Join(BeatL; BeatR) {
   window
      BeatL : partitioned, sliding, count(20);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000236

```
      BeatR : partitioned, sliding, count(1);

   param
      match : BeatR.name == BeatL.firstName + " " + BeatL.lastName &&
department == "HR";
      partitionByLHS: BeatL.id;
      partitionByRHS: BeatR.jobCategory;

   output
      Join1 : salary = salary * 2ul;
}
```

In Example 5-10 on page 216, a left-side sliding window has been defined and will have an eviction policy of count(20). This means that when the 21st tuple arrives, the oldest tuple (in order of arrival) will be evicted. The trigger policy for Join by default is count(1), which means the operator performs its function on the set of rows in the window every time a new tuple arrives and after any evictions that its arrival might cause.

A set of such sliding windows is created on the tuples in the first input stream (BeatL) because the partitioned keyword was used. The number of windows in the set will correspond to the partitionBy parameter supplied by the developer, specifically in the operator parameters section. These parameters need to identify the set of attributes used to define a group.

In Example 5-11, we define a single tumbling window on the input port to the operator. The tumbling window triggers the operator action every 60 seconds on whatever tuples have arrived in that time. The tuples in the window are then completely discarded, thus emptying the window. The time is then reset and the cycle repeats. There is only one window associated with inputStream1 because the partitioned keyword was not used.

Example 5-11 Defining a single tumbling window

```
type
   someSchema = tuple<…>;

graph
   …
   …
   stream <someSchema> myStream = myOperator(inputStream1) {
      param
          window
             inputStream1: tumbling, time(60);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000237

```
        …
        …
    }
```

For tumbling windows, the eviction policy is explicitly specified (with or without the partitioned keyword). The options are the following:

- <punct()>: Punctuation-based tumbling windows
- <count()>: Number of tuples-based tumbling windows
- <time(seconds)>: Time-based tumbling windows
- <delta(attribute-delta)>: Change in a non-decreasing numeric amount

The eviction policy needs to be explicitly defined for tumbling windows. There is no explicit trigger policy definition in the tumbling windows because the eviction policy always matches the trigger policy. When the tumbling window becomes full, it triggers/performs the action intended by that SPL operator and then evicts all stored tuples in the window.

For sliding windows, both the eviction and trigger policy must be specified and they must be specified in that order (again, with or without the partitioned keyword). There are three choices for each policy:

- <count()>: The number of tuples-based sliding windows
- <time(seconds)>: The time-based sliding windows
- <delta(attribute-delta)>: The change in a non-decreasing numeric amount

Excluding partitioned, this results in a choice of nine possibilities of sliding window definition, specified below in <evictionPolicy,triggerPolicy> order:

1. <count(n),count(n)>
2. <count(n),time(seconds)>
3. <count(n), delta(attribute,value)>
4. <time(seconds),count(n)>
5. <time(seconds),time(seconds)>
6. <time(seconds,delta(attribute,value)>
7. <delta(attribute,value),count(n)>
8. <delta(attribute,value),time(seconds)>
9. <delta(attribute,value),delta(attribute,value)>

Sliding windows do not support a trigger or eviction policy based on punctuation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000238

In Example 5-12, we illustrate a sliding window using an attribute-based eviction policy and a time-based trigger policy. In this example, temperature data is collected from national weather monitoring stations. Each weather observation has a weather observation time. The observations are guaranteed to arrive in order of observation by time and observations arrive approximately every minute from each station. However, it is not guaranteed that observations will not be missed or that the number of seconds past the top of the minute will be the same for each station.

Example 5-12 Sliding window with an attribute-based eviction policy

```
// calculate a 6-hour, running average based on temperature
observations
type
   temparatureStream = tuple<rstring monitoringStation, float32
temperature, int32 observationTime>;
   expression <uint32> $UINT32_SECONDS_IN_6_HOURS : 21600;

graph
   …
   stream <rstring monitoringStation, float32 dailyAverageTemp>
dailyAverageTemps =
      Aggregate(temparatureInputStream) {
      window
         temperatureInputStream: partitioned, sliding,
            delta(observationTime, $UINT32_SECONDS_IN_6_HOURS),
time(90);

      param
         partitionBy: monitoringStation;

      output
         dailyAverageTemps: monitoringStation = monitoringStation,
Average(temperature);
   }
```

In Example 5-12, we have made use of the following constant definition:

```
$UINT32_SECONDS_IN_6_HOURS : 21600
```

This definition allows us to use the given meaningful name instead of just the integer constant anywhere in the code that follows this definition, primarily to make the code more readable. This and other features of the type clause are described in more detail in the SPL language specification file.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000239

Note that the sliding window uses an attribute-based eviction policy and a time-based trigger policy.

The time-based policy is specified here as time(90), meaning the aggregator outputs a tuple containing the running average every 90 seconds.

The following line is where the attribute-based eviction policy is used:

```
delta(observationTime, $UINT32_SECONDS_IN_6_HOURS)
```

The attribute observationTime contains a number representing the time stamp of the observation. The value is equal to the number of seconds since January 1,1970.

The selected eviction policy states that a tuple will be evicted from the window when a new tuple arrives that has an observation time that is greater than six hours newer than the tuple being considered for eviction.

The aggregator specifies the partitioned grouping attribute in the monitoringStation code lines by using the syntax partitionBy and then calculates the average using the Average () aggregate function.

5.1.6 Stream bundles

It is important to note that the stream bundle feature was supported in earlier versions of InfoSphere Streams, that is, Version 1.2.1. This feature is no longer available in the SPL language as of Streams V2.0. Even though the stream bundle construct is not available in SPL, the same function can be easily emulated by attaching multiple streams to the same port in the current version of SPL.

5.2 Streams Processing Language operators

In this section, we describe the main features of the Streams Processing Language operators, including code samples for each one.

This section is not intended to be a substitute for the language reference manuals included with the product software, but is intended as a guide for you to more readily understand the examples shown in this chapter.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000240

Which operator(s) are needed

There are 29 SPL standard toolkit operators that are grouped in three categories, as shown in the following list:

- Relational operators
- Adapter operators
- Utility operators

All the operators listed below offer a range of powerful functions:

- Filter: Selectively removes tuples from a stream
- Functor: Filters, transforms, and performs functions on the data
- Punctor: Transforms input tuples into output tuples and adds window punctuation items to the output
- Sort: Sorts Streams data on defined keys
- Join: Joins Streams data on defined keys
- Aggregate: Aggregates Streams data on defined keys
- FileSource: Reads data from a file and produces tuples as a result
- FileSink: Writes tuples to a file
- DirectoryScan: Watches a directory, and generates file names on the output
- TCPSource: Reads data from a TCP socket and creates tuples out of it
- TCPSink: Writes data to a TCP socket in the form of tuples
- UDPSource: Reads data from a UDP socket and creates tuples out of it
- UDPSink: Writes data to a UDP socket in the form of tuples
- Export: Sends a stream from the current application making it available for other applications
- Import: Receives tuples from streams made available from other Streams applications
- MetricsSink: Creates custom operator metrics and updates them with values
- Custom: A special operator that allows user-defined logic from where tuples can be submitted
- Beacon: A utility source that generates tuples when needed
- Throttle: Used to pace a stream to make it flow at a specified rate
- Delay: Used to delay a stream by a given amount
- Barrier: Used to synchronize tuples from two or more streams
- Pair: Used to pair two or more streams

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000241

- Split: Used to split a stream into one or more output streams, based on a user-specified split condition
- DeDuplicate: Suppresses duplicate tuples that are seen within a given time period
- Union: Combines tuples from streams connected to different input ports
- ThreadedSplit: Splits tuples across multiple output ports to improve concurrency
- DynamicFilter: Decides at run time which input tuples will be passed through
- Gate: Controls the rate at which tuples are passed through by using an acknowledgement scheme from a downstream operator
- JavaOp: Calls out to operators implemented in Java

Table 5-1 is provided as a guide in selecting which operator(s) to use for a given type of task.

Table 5-1 A guide to use of the operators

Task	Operator(s) to use	Notes
Read or write external sources and target data.	xxxxSource to read, xxxxSink to write.	The source and sink can connect to TCP or UDP sockets or read and write files.
Aggregate data.	Aggregate.	Window enabled operator.
Join or lookup.	Join.	Window enabled operator.
Pivot groups of data into lists.	Aggregate.	Window enabled.
Derive new or existing attributes.	Functor.	The functor can retain values between tuples, use state variables, and access prior tuple data (history).
Filter data out.	Filter, Functor.	
Drop attributes.	Functor.	
Split data with different characteristics into separate streams.	Split.	

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000242

Task	Operator(s) to use	Notes
Run multiple instances of an operator on different sets or partitions of data in the same stream for performance gains.	Precede that operator with a split and have several copies of the your required operator in the job. Split the input stream on the data characteristics by which you want to partition.	
Collect data streams together.	If the streams have the same column attributes, any input to an operator can perform this task through using a semicolon separated list. If you need to coordinate the timing of data collection, use a barrier or Delay operators.	
Coordinate the timing of data in different streams.	Evaluate the use of windows, the delay, and the Barrier operators corresponding to the distribution, synchronization, and downstream processing requirements.	
Sort data to meet downstream required dependencies.	Sort.	Sort is a window enabled operator.
Drop one or more columns.	All operators can perform this task; just do not define a matching output attribute for the input attribute to be dropped.	
Poll a directory for new files to appear (when copied by external applications).	DirectoryScan.	
Provide application-specific custom metrics that can be externally viewed using the streamtool and the Streams Live Graph tool.	MetricsSink.	

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000243

Task	Operator(s) to use	Notes
Add custom logic in your SPL program with an added option to maintain state.	Custom.	
Generate random test tuples with some amount of customization to the generated attribute values.	Beacon.	
Control the rate at which tuples flow in the application.	Throttle.	
Delay a stream by a user-specified amount.	Delay.	
Pair tuples from two or more streams.	Pair.	
Eliminate duplicate tuples during a specified time period. (Commonly needed in telco applications.)	DeDuplicate.	
Combine tuples from streams arriving at different input ports.	Union.	
Fan out the incoming tuples in a way such that the tuples emitted on different output ports are processed concurrently.	ThreadedSplit.	
Control or filter which tuples will be passed through at run time.	DynamicFilter.	
Build a load balancer to control the rate at which a downstream operator can consume tuples.	Gate.	
Call out to pre-existing Java class implementations.	JavaOp.	

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000244

5.2.1 Operator usage language syntax

In Figure 5-5, each input port can receive tuples from multiple streams, providing that when this is done, each of those streams has the same schema. However, each separate input and output port can have different schemas as required.

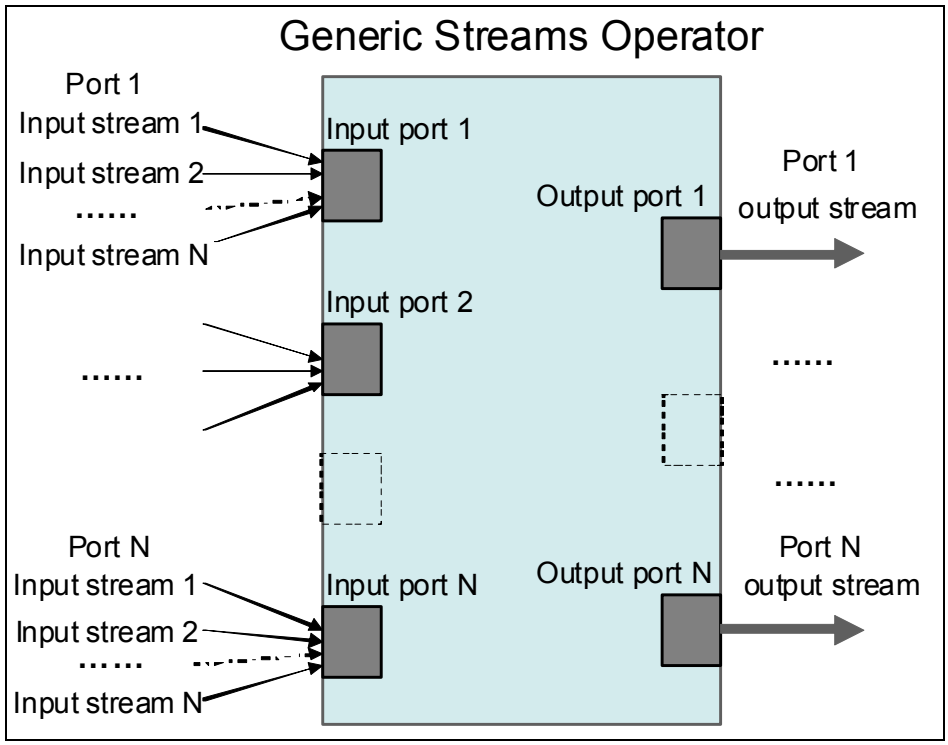


Figure 5-5 Operator inputs / outputs

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000245

The language syntax for all operators follows the structure shown in Example 5-13.

Example 5-13 Operator language syntax

```
(stream <outputStream1Schema> outputStream1name; stream
<outputStream2Schema> outputStream2name; …; stream
<outputStreamNSchema> outputStreamsNname) =
   operatorName(inputPort1Streams1, inputPort1Stream2,
inputPort1StreamM; …; …;
   inputPortNStream1, inputPortNStream2, …, inputPortNStreamM) {
   param // Operator parameters are specified below
      …

   output // Output stream attribute assignments are done below
      ….
}
```

Note that the example shows the general case of an operator having several input and output streams.

We have also shown several input streams going to the same input port on that operator. Be aware that when this is done, the streams on the same port must have the same schema.

Not all built-in operators support multiple input ports or multiple output ports. The minimum is one input port or one output port, as in the case of a source or sink.

When operators support more than one input or output port, in many cases the number of ports used for input or for output for any given operator instance is determined at compile time, corresponding to the required application.

The use of each operator must include both types of parenthesis, (), and {}. The contents of the param clause may contain optional operator parameters and the output clause may optionally contain derivations of the outputSteams attributes. If the derivation for an output attribute is not explicitly listed, the attribute derivation is “auto matched” to a corresponding input-named attribute.

A detailed description of the use of each of the operators is provided in the following sections.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000246

5.2.2 The Source operator

Source operators may be used to read external data into a Streams application from files and TCP/IP and UDP sockets. The file or socket is specified using a uniform resource identifier (URI). In SPL, there are separate Source operators for dealing with files and sockets.

Using the FileSource operator to read tuples from a file

An example of how to use the FileSource operator to read from a file is shown in Example 5-14.

Example 5-14 Using the Source operator to read from a file

```
stream <int32 callerNumber, rstring name> myStream = FileSource() {
   param
      file: "myFile.csv";
      format: csv;
      hasDelayField: false;
}
```

Where:

- The file parameter is used to specify the file name from where the tuples will be read.
- The schema of the output stream is int32, rstring, that is, the output tuples will have two attributes each.
- The file name specified above does not include a full path. In this case, the Streams program expects to find the file in a data directory beneath the directory of the `.spl` file containing the code, so in this example the relative file path would be `./data/myFile.csv`.
- The hasDelayField parameter is specified so that the Source operator does not expect a delay duration attribute in the input stream.
- The format parameter indicates that the developer expects the data for each tuple to be on a separate line with each attribute comma separated in order of the schema definition.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000247

- Omitting the format parameter would cause the operator to expect each incoming attribute to be denoted in comma separated name:value pairs with each tuple new line delimited, as shown in Example 5-15, with an ASCII file containing two attributes per tuple in the default format.

Example 5-15 Example ASCII file

```
445558444123,Priti
15553120213,Davis
555551239871, Lee
```

Attributes that are one of the supported list types are represented using square braces [ ]. For example, [1,2,3] could be used for an list<int32>.

Boolean, true or false, attribute values are represented in a file by the characters T and F, respectively.

Using the TCPSource operator to read from a socket

The TCPSource operator can read a stream of data from a TCP/IP socket as well as from a network socket, as shown in Example 5-16. In this case, the corresponding data source may be a client or a server source. The use of an external service by reference to a name, rather than the traditional hostname:portname, may also be used. However, this setup requires a available name lookup server in the network that can perform the translation of a service name to a hostname:portname. Because it is a source, TCPSource does not have any input ports.

Example 5-16 Reading from a TCP/IP socket

```
stream <…> mySourceStream = TCPSource() {
   param
      role: client;
      address: "myexternalhost.ibm.com";
      port: 4512;
 }
```

In Example 5-16, the TCPSource operator is being asked to connect to the host adapter with network address myexternalhost.ibm.com. The connection will attempt to be made over port 4512.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000248

There is another operator to connect to UDP data sources and it is called UDPSource. In Table 5-2, we summarize important parameters for TCPSource and UDPSource operators.

Table 5-2 Supported protocol specifiers

Parameter	Description
role	A mandatory parameter that specifies whether the operator is server-based or client-based. It can either be server or client.
address	A destination server address if the role is specified as client.
port	Depending on the role, it specifies either the port where the connections will be accepted or the destination server port.
name	A developer can decide to use a descriptive name that will be automatically register the name:port pair with the name service available within the Streams run time. Other operators in the application use the name:port to communicate with this Source operator.
receiveBufferSize	An optional parameter that can be used to override the default kernel receive buffer size.
format	It specifies the format of the incoming tuples (csv, line, txt, and so on).

The above listing is only a sample. There are more operator parameters available within the SPL documentation.

The Source operator and tuple inter-arrival timing

The Source operator might expect an additional, first attribute to the output stream, which is the inter-arrival delay measured in seconds between the tuples. This inter-arrival delay is frequently not required. Where timing is important, the tuples may contain a time stamp attribute. If inter-arrival delays are not present in the incoming data tuples, the hasDelayField parameter should be set to false in the operator parameters section.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000249

The Source operator and initialization timing delay

For testing purposes, where sample input data is collected from a file rather than from a continuous input stream, it is often required that the Streams application flow, as a whole, is fully initialized prior to the Source operator sending the first tuple. This action avoids the possibility of the first few tuples being processed and discarded. In this case, the optional initDelay=<value> parameter can be added to the Source operator parameters section. The term <value> is a numeric constant representing the number of seconds to wait before sending the first tuple.

Additional useful parameters for FileSource

FileSource can optionally have one input port when it is used in combination with the DirectoryScan operator. In that case, the DirectoryScan operator will send a file name in its output tuple, which will be fed into the FileSource input port. The FileSource operator can take any of the following operator parameters (which will be helpful in many real-life application scenarios):

1. hotFile: A Boolean type used to indicate if the input file will keep growing.
2. deleteFile: A Boolean type that specifies whether the file should be removed after processing is completed for that file. This parameter cannot be specified if hotFile is specified.
3. moveFileToDirectory: An rstring type used to specify that the file should be moved to a directory after processing is completed for that file. This parameter cannot be specified if hotFile or deleteFile is specified.

The three variations of Source operator that introduce punctuation shows the usage of these parameters, and are shown in Example 5-17.

Example 5-17 The three variations of Source operator

```
type
   cityData = tuple<rstring city, rstring country, uint32 population,
uint32 medianAge, uint32 percentageEducated>;

graph
   stream <cityData> CityDataRecord1 = FileSource() {
      param
         file: "../test1/Population.txt";
         hotFile: true;
   }

   stream <cityData> CityDataRecord2 = FileSource() {
      param
         file: "../test2/Population.txt";
         deleteFile: true;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000250

```
   }

   stream <cityData> CityDataRecord3 = FileSource() {
      param
         file: "../test3/Population.txt";
moveFileToDirectory: "/tmp/Processed-Files/";
   }
```

5.2.3 The Sink operator

The Sink operator is used to perform the reverse function of the Source operator. It receives an input stream and externalizes the tuples to the desired output format, which may be a continuing transmission to a TCP/IP socket or writing the tuple data to a csv-formatted file on the Linux file system.

Accordingly, the Sink operator has one input port and no output ports.

Sink operators also come in three types:

- FileSink is used to write tuples into a file.
- TCPSink is used to send tuples targeted to a TCP address.
- UDPSink is used to send tuples targeted to an UDP address.

Because there is no output port for Sink operators, a typical Sink SPL statement may appear as in the following example:

```
() as FileWriter1 = FileSink(CityDataRecord3) {
      param
         file: "../test3/Population-Output.txt";
         hasDelayField: false;
   }
```

Note the use of the parentheses, “()” on row one in positions 1 and 2 to indicate that there is no output stream emitted by a Sink operator.

Punctuation and the Sink operator

By default, the Sink operator does not write punctuation markers to text-based output. For example, they are marked by the # character. To disable this behavior, use the writePunctuations parameter and set it to true.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000251

5.2.4 The Functor operator

The Functor operator is used where the developer needs to perform functional transformations, including filtering and dropping attributes from the input stream.

The Functor has one input port and one or more output ports. It processes the input on a tuple by tuple basis, so window functionality is not required or supported.

A Functor operator is declared as shown in Example 5-18.

Example 5-18 Declaring a Functor operator

```
(stream <rstring accountHolder, rstring accountNumber> FirstOutput;
 stream <rstring accountHolder, rstring ticker> SecondOutput;
 stream <rstring accountHolder, int32 quantity> ThirdOutput) =
Functor(AccountRecord) {
    param
       filter: quantity < 8000;
 }
```

A Functor operator takes an optional filter parameter, which specifies the condition that determines which input tuples are to be operated on. It takes a single expression of type Boolean as its value. If the input tuple is not filtered, any incoming tuple results in a tuple on each of its output port.

In Example 5-19, we show code that calculates a running total that is output after every tuple and reset after a change of value of the key attribute. This action is done by using the state-keeping facility available within an operator.

Example 5-19 Using Functor logic to derive a new output attribute

```
type
   mySchema = tuple<int32 key, int32 value>;

graph
   stream <mySchema, tuple<int32 total>> myStreamWithTotal =
Functor(myInputStream) {
      logic
         static: mutable int32 _total = 0, int32 _lastKey = -1;

      param
         filter: true;

      onTuple myInputStream: {
         if (_lastKey != key) {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000252

```
            _total = data;
            _lastKey = key;
         } else {
            _total += data;
         } // End of if (_lastKey != key)
      } // End of onTuple myInputStream

      output myStreamWithTotal: {
         key = key;
         data = data;
         total = _total;
      } // End of output myStreamWithTotal
   } // End of Functor.
```

Functors can also easily access previous input tuples, which is referred to as *history access*. For example, to achieve access to the previous tenth tuple attribute, called temperature on input port named MyInput, you can use the expression MyInput[10].temperature.

Knowing this, Example 5-19 on page 232 can be simplified, eliminating the need for the lastkey variable. The custom logic section could now read as follows:

```
if (key != myInputStream[1].key) {
        _total = data
     } else {
        _total += data
     }
```

Where the myInputStream[1].key refers to the value of the previous tuple key attribute.

Note the following points regarding Functor history processing:

- The language currently supports a literal number in the code (known at compile time) that defines the number of tuples to go back. Using a variable as an index here is not supported. The number “to go back” has no limit, but be aware that the compiler may allocate additional buffering space to achieve larger index number processing.
- If your code refers to tuple numbers, specifically ones prior to the first tuple, you should be aware of what happens prior to data being received by the program. In this case, the attributes will have default values. Those are 0 for numerics, the empty string for strings, false for Boolean variables, and the empty list for lists.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000253

Punctuation and the Functor operator

The Functor operator preserves any punctuation it receives. Therefore, punctuation passes through the Functor operator unchanged, in order, on receipt and irrelevant of any Functor processing or filtering.

If it is required to insert punctuation based on tuple attribute input condition(s), the Punctor operator should be used.

5.2.5 The Aggregate operator

The Aggregate operator is able to perform summarization of user-defined groups of data where the summarization is based on data of all values. Also, processing may be divided into groups where one or more attributes have the same values. This operator has one input port, one output port, and supports both sliding and tumbling windows. In Table 5-3, we show the choice of functions on a group of tuples:

Table 5-3 Aggregator functions

Function	Description
Count()	Count the tuples in the group.
Min()	Determine the minimum value of an attribute.
Max()	Determine the maximum value of an attribute.
Average()	Determine the average of an attribute.
Sum()	Accumulate the total of an attribute.
First()	The first value of an attribute.
Last()	The last value of an attribute.
CountDistinct()	The count of distinct values of the attribute.
Collect()	Create a Streams List data type containing all the values of an attribute.
CollectDistinct()	As Collect(), except the list contains only distinct values.
CountAll()	Total number of tuples in windows of all groups.
CountGroups()	Number of groups.
CountByGroup()	Create a Streams List data type containing the group sizes for all groups.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000254

In Example 5-20, we show an example use of the Aggregate operator.

Example 5-20 Aggregate operator usage

```
// Assumed aggregator input stream schema
type
   stockPrices = tuple<rstring stockSymbol, float32 price>;
   avgStockPrices = tuple<rstring stockSymbol, float32 avgPrice>;

graph
   …
   stream <avgStockPrices> avgStockPrice = Aggregate(inputStockStream)
{
      window
          inputStockStream: partitioned, sliding, count(30), count(1);
          partitionBy: stockSymbol;

      output
         avgStockPrice: stockSymbol = stockSymbol, avgPrice =
Average(price);
   }
```

In regards to Example 5-20:

- We are using a sliding window along with the partitioned keyword. The window has an eviction policy that is “count-based” at 30 tuples and the trigger policy is also count-based for each tuple. The attribute being used for the group key is stockSymbol.
- We are using the Aggregate operator Average() function, which calculates averages, which means that we are calculating the moving average for each stock symbol. The example will output a calculation of the average of the last 30 tuples received for each stock symbol. Such a calculation might be used in company-based, stock-trend analysis.

Punctuation and the Aggregate operator

The prior text has noted that the Aggregate operator may use punctuation as a method to activate the trigger and eviction policy actions for tumbling windows (but not sliding windows).

The Aggregate operator inserts a window marking punctuation into the output stream after the aggregation operation is completed and a batch of tuples are emitted.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000255

5.2.6 The Split operator

The Split operator has one input port and may have one or more output ports. It may be used to split an input stream into various output streams. Parameters specified within the Split operator are used to decide which input tuples are dispatched to which output tuples. This type of operator will typically route a tuple based on specific attribute characteristics.

The Split operator does not (need to) support windows because it functions on each input tuple one at a time, routing it to the required output port(s).

The input and output stream schemas must be the same for the Split operator.

The Split operator is declared as shown in the following example:

```
(stream <OutputStreamSchema> OutputStream1;
 stream <OutputStreamSchema> OutputStream2;
 …
stream <OutputStreamSchema> OutputStreamN) = Split(InputStream1, …,
InputStreamN) {
   param
      index: hashCode(name);
}
```

There are three mutually exclusive forms of splitParameters:

- An expression returning an uint64
- An expression returning an list<uint64>
- A key-mapping file

These forms are described in more detail in the following text. In each of the descriptions, it is assumed that the Split operator has N output ports numbered consecutively 1 through N. The expressions that make up the Split operator parameters typically call for runtime comparisons with values of specific tuple attributes.

- An expression returning an Integer

  In this form, a tuple can either be dropped or sent to exactly one output port. If the integer expression evaluates to -1, the input tuple is discarded. If the expression evaluates to 0, the tuple is dispatched to output port 0 (the first port). If the expression evaluates to 1, the tuple is dispatched to output port 1, and so on. In general, a non-negative expression value of e means the tuple should be dispatched to output port (e mod *N*), where N is the number of output ports.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000256

Example 5-21 shows use of the Split operator with a integer expression.

Example 5-21 Split operator with an integer expression split parameter

```
type
   businessAccounts = tuple<rstring accounted, float64
averageBalance>;

graph
   …
   (stream <businessAccounts> highValueAccounts;
    stream <businessAccounts> mediumValueAccounts;
    stream <businessAccounts> lowValueAccounts) =
Split(inputStreamOfAllAccounts) {
       param
          index: (averageBalance >= 10000.00 ? 0 : (averageBalance
>= 5000.00 ? 1 : 2));
    }
```

High value business accounts are classified with average balances great than or equal to 10000, medium value accounts are great than or equal to 5000 and less than 10000, and low value accounts have average balances below 5000.

Note we use the conditional expression. The format is as follows:

```
if (averageBalance >= 10000.00) then send the tuple to port 0.
if (averageBalance >= 5000.00 and averageBalance < 10000.00) then
send the tuple to port 1.
if (averageBalance < 5000.00) then send the tuple to port 2.
```

Note that in the example we nest the ternary conditional operators to achieve the required result.

- An expression returning an IntegerList

  In this case, the tuple can be dropped or sent to one or more output ports simultaneously. In this form of split parameter, a list of integers is provided using a streams list<uint64> type. Each integer in the list is then interpreted by the operator in the same way as the integer expression case previously shown. The benefit of this option over the Integer operator is that a tuple can be sent to several output ports simultaneously.

  Example 5-22 shows a split using an IntegerList.

Example 5-22 A split using an IntegerList

```
// in this example we split book authors into streams of the types
of #genre of books they have written
// authors often write books of more than one genre..
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000257

```
// schema of allAuthors stream:
type
   authorGenres = tuple<rstrng authorId, list<uint64>
genresWritten>;

graph
   …
   (stream <authorGenres> SciFi;
    stream <authorGenres> Horror;
    stream <authorGenres> Crime;
    stream <authorGenres> Romance;
    stream <authorGenres> Fantasy) = Split(AllAuthors) {
       param
          index: genresWritten;
}
```

In Example 5-22 on page 237, the input stream allAuthors contains tuples that contain author identifiers and a list of the genre(s) of book(s) they have written.

For example, if an input tuple genresWritten attribute contained the values [0,4], this would be used to indicate that the associated author has written books of both SciFi and Fantasy genres. So this input tuple would be dispatched to two of the five output streams.

- A key-mapping file

  In this form, the operator parameters contains a URI file and an attribute on which to perform the split. The URI file points to the key-mapping file.

  The key-mapping file is a csv-formatted file. Each line contains a record of the following format:

  ```
  <atrribute-value>, <comma seperated list of ports>
  ```

  If the input tuple attribute value matches the first value on any of the lines, the tuple is dispatched to the list of output ports given on the remainder of the line.

  The benefit over the previous example is that the key-mapping file is an external configuration data file that could be changed or updated without requiring a code change.

  Example 5-23 provides an example using a key-mapping file.

Example 5-23 Split operator using a key-mapping file

```
// this is the same example as in A split using an IntegerList, that
has been re-written #to use a key-mapping file
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000258

```
// so the list<uint64> is now not part of
// the input or output stream schema..
// in this example we split book authors into streams of the types
of #genre of books they have written
// authors often write books of more than one genre..

// schema of allAuthors stream:
// schema of allAuthors stream:
type
   authorGenres = tuple<int32 authorId>;

graph
   …
   (stream <authorGenres> SciFi;
    stream <authorGenres> Horror;
    stream <authorGenres> Crime;
    stream <authorGenres> Romance;
    stream <authorGenres> Fantasy) = Split(AllAuthors) {
       param
          file: "MyMapping.txt";
          key: authorId;
}

// the contents of the MyMapping.txt might contain
default, -1
1,0,3,4
2,3
3,1,2

// this would mean
// author 1 has written SciFi, Romance and Fantasy
// author 2 has written Romance
// author 3 has written Horror and Crime
```

Punctuation and the Split operator

Similar to the Functor operator, the Split operator passes punctuation through unchanged. If the split has multiple output ports, an incoming punctuation is passed to all output ports unchanged. The Split operator itself does not use punctuation to determine split functionality

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000259

5.2.7 The Punctor operator

The Punctor operator has one input port and one output port. Window grouping functions are not supported or required because the operator works on a single tuple at a time.

The primary use of the Punctor operator allows the solution designer to insert punctuation marks into a stream based on required conditions. Typically, this action is done in preparation for downstream operator behavior when it is required for a tumbling window trigger and eviction policy to be based on punctuation.

The operator parameters sections support a Boolean expression, which when evaluated to true, will cause a punctuation mark to be inserted in the output stream. The expression may refer to prior tuples using the syntax for history processing described for the Functor operator. A before or after positional keyword is also included to determine if the punctuation is inserted before or after the tuple causing the expression to evaluate to true respectively.

The output attribute assignment section of the operator definition may include attribute derivation expressions as per the Functor operator.

In Example 5-24, we demonstrate the use of the Punctor operator.

Example 5-24 Using the Punctor operator

```
// in this example we insert punctuation on a key change in
// the data stream. The punctuation marker is inserted at the end of
each series of a particular key value before the next series of tuples
having the key attribute with a different value.

type
   exampleSchema = tuple<int32 key, rstring data>;

graph
   …
   stream <exampleSchema> punctuatedStream = Punctor(someInputStream) {
       param
          punctuate: key != someInputStream[1].key;
          position: before;
   }
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000260

5.2.8 The Delay operator

The Delay operator has one input port and one output port. Its function is to delay a stream by a number of seconds specified by a float64 parameter. It does not use windows because it operates on a tuple-by-tuple basis. A Delay operator is typically used when it is required to more closely synchronize tuples from different streams and where relative timing is important to downstream operator processing. In Example 5-25, we demonstrate the use of the Delay operator.

Example 5-25 Using the Delay operator

```
// delay a stream by 1.2 seconds
stream <someSchema> delayedStream = Delay(inputStream) {
   param
      delay: 1.2;
}
```

Punctuation and the Delay operator

The Delay operator passes through any punctuation marks in the stream unchanged after the specified delay while maintaining the relative position of the punctuation marker against the surrounding tuples.

5.2.9 The Barrier operator

The Barrier operator is used to combine (or merge) multiple input streams to one output stream where the input streams are logically related. It has two or more input ports and one output port. An output tuple is emitted only when an input tuple has been received by each and every input port. The output tuple attributes can be explicitly derived in the output attributes section of the operator definition from some or all of the input tuples set of attributes.

The Barrier operator does not require or support windows because it operates on a tuple-by-tuple basis for each input stream.

A typical usage is where a stream has earlier been split into two or more streams because performance can be gained by processing each of the two streams together before subsequently recombining them using the Barrier operator. In this scenario, it may be important that the order and number of tuples in the prior split stream is maintained. In other scenarios of merging or combining streams, the Join operator may be more appropriate.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000261

The Barrier operator does not require any parameters in the parameters section. We show a demonstration of the use of the Barrier operator in Example 5-26.

Example 5-26 Using the Barrier operator

```
type
   // schemas of the input streams to the barrier
   stream1Schema = tuple<int32 i, rstring a>;
   stream2Schema = tuple<int32 i, rstring b>;
   stream3Schema = tuple<int32 i, rstring c>;

   // schema of the output stream from the barrier
   outputSchema = tuple<int32 i, rstring x, rstring z>;

graph
   …
   stream <outputSchema> outputStream = Barrier (stream1; stream2;
stream3) {
      output
         outputStream: i = stream1.i, x = a, z = c;
   }


// note that where input streams contain identicailly named
// attributes e.g 'i' in the above example we explicity refer the
// the input stream number to derive the output attribute from to
// eliminate any ambiguity as to which input stream attribute of that
// name we are referring to.
```

Punctuation and the Barrier operator

The Barrier operator does not propagate punctuation. The rationale for this is that although incoming streams may have punctuation, the resulting output stream is an interleaved combination of the input tuples and therefore it would not be possible to relate punctuation to the relevant stream.

5.2.10 The Join operator

The Join operator is used to perform a relational join of two input streams. A relational join may be inner, left, right, or full outer join. The join condition and the type of join are specified in the operator parameters section.

In the description of the join functionality that follows, it is essential that you are familiar with the description of window characteristics provided earlier in 5.1.5, “Streams windows” on page 212.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000262

The Join operator has two input ports and one output port. It supports sliding windows but not the tumbling windows for both its input ports.

The eviction policy for a join may be count, time, or attribute-delta-based in common with normal sliding window characteristics.

The trigger policy for a join is always count(1). A count(1) trigger policy means that the join processing always occurs as each input tuple arrives on either input port. Because the trigger policy for each input port window is always count(1), Streams does not require it to be explicitly stated in the code of the operator definition. Note that this may make the window definition, in the code for the join, initially appear to you to look like a tumbling window definition. However, do not be deceived; recall that the join only supports sliding windows.

The operation of the join is as follows: As each new tuple is received on either port, the new tuple is matched against all of the tuples that are in the opposite port window. For each set of tuples that satisfies the join condition, an output tuple is constructed using the rules defined in the operator attribute derivation section of the code.

The partitioned keyword may also optionally be used where required on either or both input streams. Note, in this case, that the time-based eviction policy is not supported with the partitioned keyword for the join. The choices are confined to either count-based or attribute-delta-based eviction. In this case, a number of windows are created on the relevant input streams and then grouped on the relevant distinct runtime values of the attribute keys specified in the join criteria.

In Example 5-27, we demonstrate a left outer join.

Example 5-27 Streams application performing a left outer join

```
namespace my.sample;

composite Joins {
   graph
      stream <int32 key, rstring decode> decode = FileSource() {
         param
            file: "decode.dat";
            hasDelayField: false;
            format: csv;
      }

      stream <int32 key, rstring someData> primary = FileSource() {
         param
            file: "primary.dat";
            hasDelayField: false;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000263

```
            format: csv;
      }

      stream <int32 key, rstring decode, rstring someData> joined =
Join(primary; decode) {
         window
            primary: sliding, count(0);
            decode: partitioned, sliding, count(1);

         param
            match: primary.key = decode.key;
            algorithm: leftouter;
            partitionByRHS: primary.key;

         output
            joined: key = primary.key, decode.decode, primary.someData;
      }

      () as FileSink1 = FileSink(joined) {
         param
            file: "decode-output.dat";
      }
}
```

This example joins two csv-formatted input files on the integer key attribute. The decode file is assumed to contain a text description associated with numeric key identifiers. A left outer join is used in case there is a key value that cannot be decoded. In that case, the key is still output to the output file with a value of the empty string for decode.

The primary stream contains a list of keys to be decoded. Note that it flows into the first or left input port to the Join operator. A <count(0)> eviction policy for this is used, meaning that the input tuple is immediately discarded after processing.

The decode stream uses a <count(1), perGroup > Window definition. This means that a number of windows are created in the stream on the second input port to the join, each with a size of one tuple. The number of windows will equate to the number of distinct values for the key in the decode file. This approach means that we hold the decode values continuously within the memory of the operator.

Assume we have the tuple data shown below for the `primary.dat` and `decode.dat` files:

```
primary.dat
1,def
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000264

```
3,ghi
0,abc

decode.dat
0,alpha
1,beta
```

Then, the following output would be in the `decode-output.dat` file:

```
1,beta,def
3,,ghi
0,alpha,abc
```

The second line of output contains an empty string for the decode attribute, as there is no matching key value for 3 in `decode.dat`.

Punctuation and the Join operator

The Join operator inserts punctuation on its output stream to delineate the sets of resulting, matching tuples for each pair of windows processed.

5.2.11 The Sort operator

The Sort operator sorts a set of tuples in a window by a defined set of attribute keys. Single and multiple sort keys are supported, each of which may be specified as ascending or descending for ascending or descending sorts, respectively.

Sorts are typically used either as a requirement prior to exporting the tuples through a sink to meet an external target requirement, or for downstream operator window requirements. For example, attribute-delta-based windowing requires the tuples to be sorted on the relevant attribute.

The Sort operator has one input port and one output port and supports sliding and tumbling windows, with some restrictions.

Tumbling windows may be defined by count, time, or punctuation. After the tumbling window is full, the sort is performed on the defined attribute keys. In Example 5-28, we show a tumbling window based sort.

Example 5-28 Tumbling window based sorts

```
// schema of input and output streams
type
   streamSchema = tuple<int32 key1, int32 key2, rstring someData>;

graph
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000265

```
…
stream <streamSchema> sortedStream = Sort(inputStream) {
   window
      inputStream: tumbling, time(60);

   param
      sortBy: key1, key2;
      order: ascending;
}
```

Note that in this example the operator only ever processes the window when a tuple is received. So, if a tuple is not received in 60 seconds, the window will not be processed. The next tuple to be received after 60 seconds will trigger the sort and ejection of tuples. A consequence is that if we stop receiving tuples, the last tuples in the Sort operator will never be output.

Sliding windows may be defined only by a count-based eviction policy. Other eviction policies are not supported for sliding window based sorts.

The trigger policy for sorting sliding windows is always count(1) and it need not be explicitly specified. This situation is often called a progressive sort.

In Example 5-29, we demonstrate the use of a sliding window with a Sort operator.

Example 5-29 Sliding window based sort

```
type
   streamSchema = tuple<int32 key1, int32 key2, rstring someData>;

graph
   …
   stream <streamSchema> sortedSchema = Sort(inputStream) {
      window
         inputStream: sliding, count(100);

      param
         sortBy: key1, key2;
         order: descending, descending;
   }
```

In this example, 100 tuples are read into the window. Then as each new tuple is read, it is placed in order in the window. Because the window is larger than the eviction policy size, the tuple in the window with the highest values of key1, key2 is output.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000266

Punctuation and the Sort operator

The Sort operator may process punctuation based tumbling windows.

Also, the Sort operator inserts a window marking punctuation into the output stream following the completion of the sort processing on the group of tuples in the window for all supported variations of windowing schemes described.

5.2.12 Additional SPL operators

In this chapter, we have described fourteen different SPL standard toolkit operators in detail. You should now have a good background for using the SPL standard toolkit operators that are available with the Streams product. There are additional SPL standard toolkit operators:

- Filter: Selectively removes tuples from a stream.
- DirectoryScan: Watches a directory, and generates file names in the output.
- MetricsSink: Creates custom operator metrics and updates them with values.
- Custom: A special operator that allows user-defined logic from where tuples can be submitted.
- Beacon: A utility source that generates tuples when needed.
- Throttle: Used to pace a stream to make it flow at a specified rate.
- Pair: Used to pair two or more streams. Similar to Barrier, but it will output the two tuples individually.
- DeDuplicate: Suppresses duplicate tuples that are seen within a given time period.
- Union: Combines tuples from streams connected to different input ports.
- ThreadedSplit: Splits tuples across multiple output ports to improve concurrency.
- DynamicFilter: Decides at run time which input tuples will be passed through.
- Gate: Controls the rate at which tuples are passed through by using an acknowledgement scheme from a downstream operator.
- JavaOp: Calls out to operators implemented in Java.

Thorough descriptions about these additional operators as well as code snippets that explain different ways to use them are available in the `SPLToolkitReference.pdf` file or within the InfoSphere Streams Information Center website, found at the following address:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000267

You should extend the SPL standard toolkit operator skills you have gained from reading this book by referring to the additional resources mentioned above. In addition, work through the many examples targeted for learning the SPL operators. Details about these examples can be found in the `samples/spl` subdirectory of the InfoSphere Streams installation. More hands-on SPL examples can be found at the IBM developerWorks® website at the following address:

`https://www.ibm.com/developerworks/mydeveloperworks/files/app?lang=en#/collection/ef9f8aa9-240f-4854-aae3-ef3b065791da`

By taking advantage of dozens of available SPL examples, new Streams developers can get a quick start on building Streams applications that use SPL operators.

5.3 The preprocessor

It is important to note that two preprocessors were supported in earlier versions of InfoSphere Streams (Version 1.2.1): One preprocessor with features such as “for loops” and another preprocessor for mixed-mode (Perl embedded within Streams code). In the current version of SPL, the first preprocessor has been dropped altogether. However, SPL provides constructs at the language level to perform some of the previously-supported features of the absent preprocessor. Mixed-mode PERL is fully supported by SPL.

Even though some of the features listed below are not offered through a preprocessor, in this section we describe how they can be done in SPL:

- Defining constants
- Including logic from other SPL files
- Application parameterization
- Loops
- Mixed mode processing with Perl

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000268

In the following paragraphs, we provide descriptions and examples for each of these features:

- Defining constants

  If you are familiar with the C or C++ programming languages, this description will be familiar. In C or C++, the #define functionality enables the developer to define literal textual replacements for a given keyword. The coding convention is to define the text to be replaced in all upper case. Similarly, we can define constants using the SPL language feature rather than a preprocessor directive, as shown in Example 5-30.

Example 5-30 Using constants

```
composite Main {
   param
      expression <uint32> $SECONDS_PER_DAY : 86400u;
      expression <rstring> $MY_MESSAGE : "DAY_CHANGE_EVENT";

   type
      …

   graph
      stream <someSchema> someOutputStream =
Functor(someInputStream) {
         logic
            state: {
               mutable rstring _message = $MY_MESSAGE;
               mutable boolean _dayChange = false;
            }
            onTuple someInputStream: {
               if ((getTimestamp()/1000000) % $SECONDS_PER_DAY) ==
0) {
                  _dayChange = true;
               } else {
                  _dayChange = false;
               }
            }

         param
            filter: _dayChange == true;

         output someOutputStream: message = _message;
      }
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000269

The code snippet in Example 5-30 on page 249 uses the more meaningful text SECONDS_PER_DAY instead of 86400, and also defines a message to be output. The benefit is that it is easier to change the message text later on (for example, due to being referenced in several places) rather than searching through the code for all occurrences.

- Including/using logic from other SPL files

  The SPL language does not have syntax similar to #include, which is supported by the C/C++ preprocessor. However, SPL allows a developer to access/use logic residing in other SPL files.

  Typically, application logic from different SPL files can be accessed from an SPL application for the following reasons:

  – Common code that needs to be used in more than one Streams source file. The benefit is that the common, shared code can be maintained in one place.

  – More complex logic, with the benefit that such logic can be more easily developed and tested in a separate file by a test harness prior to using it as part of the full application as well as to assist readability of the overall code structure.

  – SPL metadata type definitions, with the benefit of also simplifying repeating metadata of a stream, that might be referenced in several source files.

  We demonstrate the use of Streams **`use directives`** in Example 5-31.

Example 5-31 Using Streams use directive (part 1)

```
namespace my.sample;
use com.acme.utils::MyTypes;

graph
   …
// testStream is a type defined in the MyTypes comosite and
// it is referenced in several source files
stream <testStream> outputStream = FileSource() {
   …
}
```

We demonstrate another use of the Streams **`use directive`** in Example 5-32.

Example 5-32 Using Streams use directive (part 2)

```
// streams metadata used in several applications could be defined in
an SPL files that
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000270

```
// can be accessed in other Streams applications through the use
directive.
namespace com.acme.types;
type
   vOstream = tuple<uint32 customerKey, rstring customerName,
rstring customerType>;
   vInputStream = tuple<rstring name, uint32 id, float32 balance>;
   vTransactionStream = tuple<rstring accountNumber, uint32
accountType>;
   …
   …

composite Main {}


Once all the common types are defined in an .spl file as shown
above, any of these type definitions can be used in other
applications by declaring use com.acme.types::*.
```

Example 5-31 on page 250 and Example 5-32 on page 250 both show examples of storing common SPL type definitions in an SPL file.

- Application parameterization

  In SPL, applications can be externally parameterized by passing special arguments to the applications, either during compile time or during application submission time. The Streams compiler (sc) supports arguments that are passed in at compile time. These arguments can be accessed from within SPL code through the following three compile time argument access functions.

  ```
  rstring getCompileTimeValue(rstring name)
  rstring getCompileTimeValue(rstring name, rstring default)
  list <rstring> getCompileTimeListValue(rstring name)
  ```

  These functions can appear in any location within an SPL file. The first function returns the value of a named argument as a string. The second function returns the value of a named argument, or a default value in case the argument is not specified at compile time as a string. The third function expects the value to be a string with commas. The string will be split at the commas and the result is returned as a list<rstring>.

  The named arguments are specified as <name>=<value> on the **sc** compiler command line, as shown below:

  ```
  sc  -M my.sample::MyApp  hello=a,b,c   aname=bar
  ```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000271

SPL also supports another way to pass parameters into an application. It is done through submission-time values, which are arguments specified at the time of application launch. For distributed applications that run on the InfoSphere Streams run time, these arguments are specified when submitting a job through the **`streamtool submitjob command`** or through other means supported by the Streams tooling. For stand-alone applications, submission-time values are specified when launching the generated stand-alone application. These values can be accessed from within SPL code through the following submission-time value access functions:

```
rstring getSubmissionTimeValue(rstring name)
rstring getSubmissionTimeValue(rstring name, rstring default)
list<rstring> getSubmissionTimeListValue(rstring name)
list<rstring> getSubmissionTimeListValue(rstring name, list<rstring>
default)
```

To learn more about these two approaches to passing parameters into a Streams application, refer to the *SPL Compiler Usage Reference*, found at the following address:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp`

- Loops

  SPL allows the use of loops inside the application logic in two ways. Any code block that has to be conditionally iterated can be put inside either for or while loops. SPL for loops use the syntax for (typeId in expression) and SPL while loops are just like in C or Java with the syntax while (expression) statement.

  SPL for loops are typically used to iterate over SPL collection types, such as list, map, set, and string. A for loop over a string or list iterates over the elements or characters in index order. In a for loop over a map, the loop variable iterates over the keys. It is important to note that the iterated-over collection becomes immutable for the duration of the loop.

  An SPL while statement evaluates the specified expression and executes the statements in the body of the while loop. There are three loop control statements, named break, continue, and return. A break statement abruptly exits a while or for loop. A continue statement abruptly jumps to the next iteration of a while or for loop. A return statement abruptly exits a function.

  If there is a need to replicate a block of SPL code multiple times during the preprocessor phase, Perl for loops can be used. An example of using a Perl for loop inside an SPL application is explained in Example 5-33 on page 254.

- Mixed mode processing with Perl

  The four previously described SPL features are suitable for many small to medium size Streams applications.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000272

However, for larger-scale applications, more extensive scripting support might be required. In this case, the language supports what are referred to as *mixed mode programs*. For mixed mode programs, the preprocessor executes an additional step where sections of the code, which are written as Perl code, are used to generate Streams code.

Mixed mode programs need to have the extension .splmm rather than .spl. Streams uses Perl to convert `.splmm` files into `.spl` files as a first step, then passes the `.spl` file through any subsequent preprocessor and compilation steps as normal, as previously described.

Any Perl code used in a `.splmm` file must be enclosed in one of two following delimiters:

- `<% perl-code-here %>`

  In this form, the code between the delimiters is passed straight to Perl.

- `<%= perl_print_expression %>`

  The following is a shorthand for a Perl print statement (that is, it is equivalent to it):

  ```
  <% print print_expression; %>
  ```

  Note that this shorthand, as well as saving the typing of the print keyword, also includes the trailing semicolon.

We do not describe the features of the Perl scripting language in this book, as it is covered extensively in many other, publicly available publications.

We also emphasize that this is a preprocessing step used to generate Streams code during compilation. The solution is not using the Perl code examples shown as part of the run time of the application.

We have demonstrated the use of Perl with a mixed mode source file in Example 5-33 on page 254. In this example, the mixed mode application uses a FileSource operator to read stock ticker information from a file. Then, it uses a Split operator to split the incoming stock tuples into one of the 26 available output ports. Each output port in the Split operator will carry stock ticker tuples starting with a particular alphabetic character. To create such a variable number of output ports, it is ideal to use a Perl loop to parameterize the output tuple definition. As you can see from the source code example above, mixing Perl with SPL is a powerful way to parameterize and add parallel operator invocations within your Streams applications. Just for illustration, the SPL code above includes three FileSink operators to log output tuples coming from three of the twenty six output ports within the Split operator.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000273

Example 5-33 A mixed mode Streams source file

```
namespace my.sample;

composite Main {
   type StockReportSchema = tuple <rstring symbol, rstring dateTime,
float64 closingPrice, uint32 volume>;

   graph
      stream<StockReportSchema> StockReport = FileSource() {
         param
            file: "stock_report.dat";
            format: csv;
            hasDelayField: true;
      } // End of FileSource.

      <% my $cnt=0; %>
      (<% for ($cnt=1; $cnt<=26; $cnt++) {%>
         stream<StockReportSchema> StockReportOutput<%=$cnt%>

         <% if ($cnt < 26) { %>
            ;
         <% }%>
      <% } %>)
      = Split(StockReport) {
         param
         // index: hashCode(toCharacterCode(symbol, 0) -
toCharacterCode("A", 0));
         file: "mapping.txt";
         key:  symbol;
      } // End of Split

      () as FileWriter1 = FileSink(StockReportOutput1) {
         param file: "split_ticker_output_stream_1.result";
      } // End of FileSink(StockReportOutput1)

      () as FileWriter2 = FileSink(StockReportOutput7) {
         param file: "split_ticker_output_stream_7.result";
      } // End of FileSink(StockReportOutput7)

      () as FileWriter3 = FileSink(StockReportOutput9) {
         param file: "split_ticker_output_stream_9.result";
      } // End of FileSink(StockReportOutput9)
} // End of composite Main
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000274

5.4 Export and Import operators

In addition to making use of the Source and Sink operators, InfoSphere Streams applications can also communicate with each other using streams directly. This situation is achieved by using the concepts of exporting and importing streams. SPL provides two standard toolkit operators named Export and Import for this purpose.

Named streams can be imported or exported at any point in the flow.

The following two types of export and import are supported:

- Point to point
- Publish and subscribe

In the following list, we describe those two types:

- Point to point

  In point to point, one Streams application exports a stream and a second Streams application imports it.

  A stream is received by the Export operator's input port and exported using an alias output port name. The Export operator sends a stream from the current application, making it available to Import operators of applications running in the same streaming, middleware instance.

  Example 5-34 shows the output stream of a Source operator being exported. In addition, note that this stream may optionally be consumed as normal by other operators within the same application (as is the case in this example, the Sink operator also consumes the stream in the same application that exports it).

  Note that any given application can export multiple streams, but each stream must have a unique name within that application, which is a normal practice for stand-alone applications. As shown in Example 5-34, the Export operator can have one of these two optional parameters (properties that specify a tuple literal giving name-value pairs and 'streamId' as another optional parameter, of type rstring, which specifies the external name of the stream being exported).

Example 5-34 A Streams application exporting a stream

```
…
…
graph
      stream<StockReportSchema> StockReport = FileSource() {
         param
         file: "stock_report.dat";
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000275

```
            format: csv;
              hasDelayField: true;
      }

         () as ExportedStream = Export(StockReport) {
            param
               streamId: "exportedStream";
         }

         () as FileWriter1 = FileSink(StockReport) {
            param
               file: "exported_stream.result";
         }

…
…
```

The Import operator receives tuples from streams made available by Export operators of applications running in the same streaming middleware instance. Import streams will be matched to Export streams. The match may be done by subscription / properties or by streamId name.

An imported stream is given a local stream name. This name might or might not have the same name as it had in the exported application.

Example 5-35 shows an application that imports the stream exported from the application shown in Example 5-34 on page 255. In this example, the imported stream with the name ImportedStream in the original exporting application is given the localname ExportedStream. This example reads the imported stream and saves it to a `.csv` file.

Example 5-35 An application importing a stream

```
…
…
graph
   …
   …
   stream <StockReportSchema> ImportedStream = Import() {
      param
         applicationScope: "TickerProcessing";
      applicationName: "my.sample::Main";
      streamId: "exportedStream";
   }

   () as FileWriter1 = FileSink(ImportedStream) {
      param
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000276

```
            file: "imported_stream.result";
    }

…
…
```

You might want to create a source file, named `input.csv`, in the `program data` directory containing some test tuples with the format of the StockReportSchema.

If the run is successful, you should see two newly created files in the data directory: `exported_stream.result` and `imported_stream.result`, both containing identical copies of the input data. This simple example demonstrates the similar usage of both internal and exported / imported streams.

Sequential input files are commonly used for testing, as in this case, but have the disadvantage that they are finite in length. The exporting application's Source operator parameter may include initdelay=*n* seconds, which causes the Source operator to wait for n seconds before sending the data through, giving time for the importing application to fully initialize. If you launch the importing application after this time, you would not expect it to see the exported file data in this case.

You should also note that with an infinite source of tuples (as may be typical in a Streams application), the importing application could be launched at any time and it would pick up tuples from the exporter, only following the launch and initialization of the importer. If the exporting application also contained operator code that consumed the streams (as in the example), those downstream operators would not be held up waiting for the importing application to also start.

Conversely, an importing application will wait for tuples forever if a corresponding exporting application, of the required name and stream name, is never started.

Finally, even though this method is called point to point, multiple importing applications can exist, each tapping into the same exported stream.

- Publish and subscribe

  The publish and subscribe model extends the point to point model by adding the ability to associate named properties to a stream. An importing application can import streams from any application that exports streams, providing it has the properties named by the importer, as shown in Example 5-36 on page 258.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000277

Therefore, the publish and subscribe model removes any need for the importing application to know both the name of the application exporting the stream and the exported stream name. Instead, an importing application can connect to an exported stream of any arbitrary name coming from any application exporting it, provided it has the specified named properties.

Within the exporting application (the application publishing the stream), the export properties key-value pair is used together with a list of named properties that the designer is free to specify.

Within the importing application, the Import operator uses a subscription parameter with the required key-value pair expression.

In Example 5-36, we have two code snippets, one showing an Export operator with published properties and an Import operator with property subscription.

Example 5-36 Publishing and subscribing through Export and Import operators

```
graph
   …
   …
   () as ExportedStream = Export(SomeStream) {
      param
         properties : { call_type="outgoing",
call_mode="international",
            carriers=["Vodafone", "Telefonica", "Orange"] };
   }

   stream <SomeStreamSchema> CallTracer = Import() {
      param
         subscription: call_type == "outgoing" && call_mode ==
"international" &&
                           "Telefonica" in carriers;
   }
```

In Example 5-36, the Export operator includes the properties parameter that specifies properties of the stream as a tuple literal giving name-value pairs. In another application, the Import operator can have a subscription parameter with a list of exported stream properties in which it is interested. The Streams middleware connects the Import and Export streams if the subscription predicate matches the exported stream properties.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000278

5.5 Example Streams program

In this section, we describe a Streams Processing Language example that you could compile and run.

The infamous *Hello World* example program is widely accepted as originating from the book *The C Programming Language* by Kernighan and Ritchie, first published in 1978. The scenario given is that your first program in a new programming language should do something simple, such as output a message that says “Hello World”.

The benefit is that to achieve this, the developer must:

- Have gained access to a development environment
- Have accessed the compiler and a properly installed development and runtime environment for the programming language concerned
- Know how to use a compatible language editor
- Be able to compile and run the software

If the program successfully runs the simple Hello World code, the developer then knows the steps taken, as well as the environment, are, at a basic level, good to use going forward to use to build more and more complex programs.

In Example 5-37, the program uses the SPL Beacon operator to generate five tuples filled with its single rstring-typed attribute, set to the string value of “Hello World”. If the program runs correctly, these Beacon-generated tuples will be received by a custom operator and displayed on the console along with the “Hello World” message(s).

Example 5-37 Hello World example

```
/*
This example is the simplest possible SPL application.
It uses a Beacon operator to generate tuples that carry
"Hello World" messages. A custom sink operator receives
the tuples from Beacon and displays it on the console.
*/
composite HelloWorld {
   graph
      stream <rstring message> Hi = Beacon() {
         param
            iterations: 5u;

         output
            Hi: message = "Hello World!";
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000279

```
      } // End of Beacon.

      () as Sink = Custom(Hi) {
         logic
            onTuple
               Hi: printStringLn(message);
      } // End of Custom.
} // End of HelloWorld composite.
```

5.6 Debugging a Streams application

In this section, we describe some options and approaches for debugging a Streams application. There are nearly as many methods to debug a Streams application as there are business problems that you can solve with Streams.

The sample Streams application used in this section is listed in Example 5-38.

Example 5-38 Streams application used to demonstrate Streams debugging

```
namespace com.acme.streams.test;

composite Main {
   type
      InputSchema = tuple<int32 col1, int32 col2>;

   graph
      stream <InputSchema> InputStream = FileSource() {
         param
            file: "InputFile.txt";
            hasDelayField: false;
            format: csv;
      }

      stream <InputSchema> ThrottledInputStream = Throttle(InputStream)
{
         param
            rate: 1000.00;
      }

      () as FileWriter1 = FileSink(ThrottledInputStream) {
         param
            file: "OutputFile.txt";
            format: csv;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000280

```
    }

  config
    logLevel: debug;
}
```

Here are some considerations related to the code shown in Example 5-38 on page 260:

- The example listed is a self-contained Streams application, which means that it does not read from any imported streams, it does not export any streams itself, and it contains no references to externally defined user routines or functions.
- The Streams application reads from one operating system file (`InputFile.txt`), and outputs two columns (col1 and col2) on the Stream named InputStreams.

  This stream is consumed by the Throttle operator to control the rate and then it is consumed by a Sink operator. This Sink operator writes to an operating system file named `OutputFile.txt`.

  In effect, this Streams application is a simple file copy of every row and every column of `InputFile.txt` to `OutputFile.txt`.
- The contents of `InputFile.txt` are not displayed. Per the example, this file contains two numeric (integer) columns separated by a comma delimiter and a new-line character as the record separator.

  This file contains 10,000 records, with values ranging from 10,000 to 20,000, increasing sequentially and by a value of one. The sample contents are shown in the following list:

  ```
  10000,10000
  10001,10001
  10002,10002
    … {lines deleted}
  19998,19998
  19999,19999
  20000,20000
    {end of file}
  ```
- Unique to this example are two modifiers of specific interest:
  - At the end of the SPL file, you can see a configuration directive named *debug*.

    This directive allows the Streams run time to output a given level of diagnostic information to a log file on the operating system hard disk,

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000281

specifically the Streams PE log file. (This file or files, of variable name and location, is configurable.)

Other values that may be placed in this location include trace, (debug), info, and error. Each value determines an increasing level of verbosity to the logging file contents.

– An SPL operator called Throttle will slow down the output of records from the stream named InputStream to, in effect, an output of 1000 records.

This operator would likely be removed for any production-level, Streams application, but we find the ability to slow down the execution of a Streams application to be useful when debugging.

5.6.1 Using Streams Studio to debug

The Eclipse-based Streams Studio has many visual components that enable effective debugging. To create an error condition, we misspelled the name of the operating system file shown in Figure 5-6 on page 263. It is spelled `InputFile2.txt`, but the correct name is `InputFile.txt`; `InputFile2.txt` does not exist.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000282

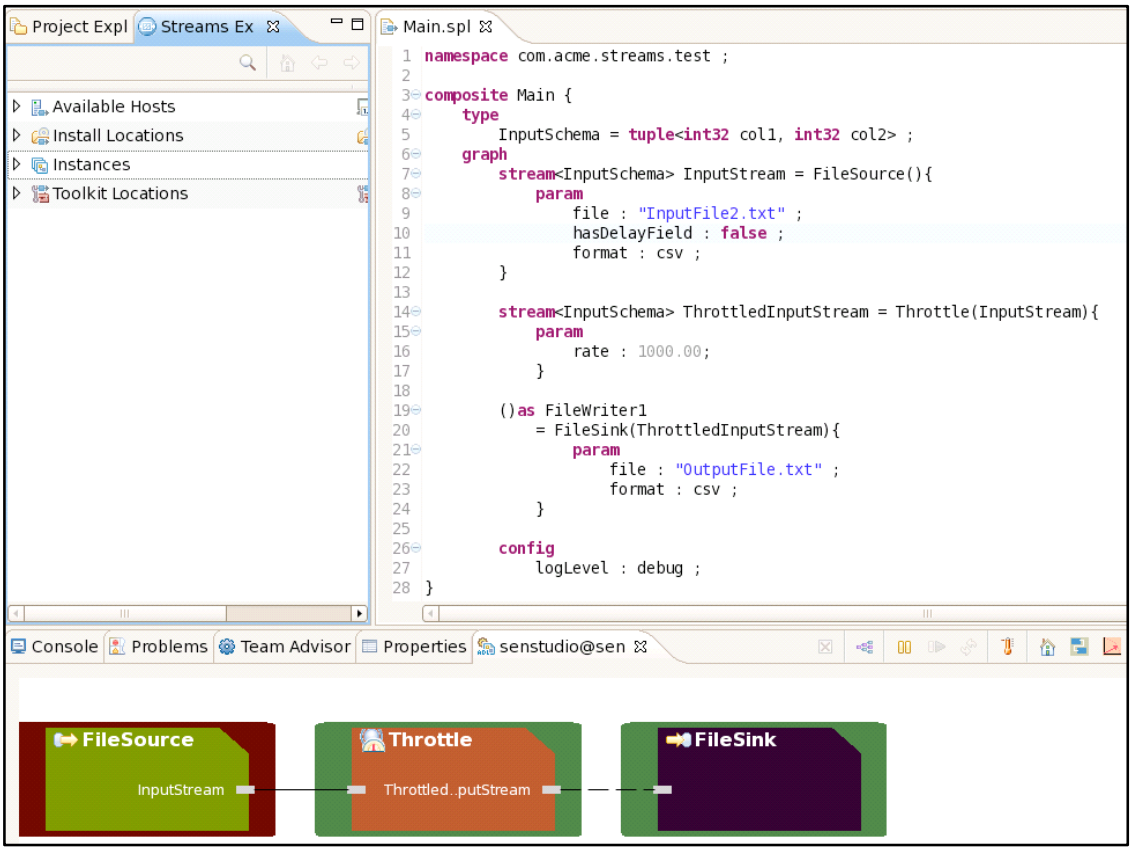


Figure 5-6 Streams Studio, Streams Live Graph - No rows being output

In Figure 5-6, we see that the Streams application is currently running. We know this because the Streams Live Graph View displays the Streams application. Also in Figure 5-6, we see there is no output from the left side operator to the one on the right side. This is because the left side operator is in a STOPPED state (red in color) and it is not outputting records because the input source file was not found. You can also notice the broken connection / dashed line between the Throttle and the FileSink operators, which also indicates no data is flowing.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000283

Figure 5-7 displays the Streams Live Graph View with no Streams applications running.

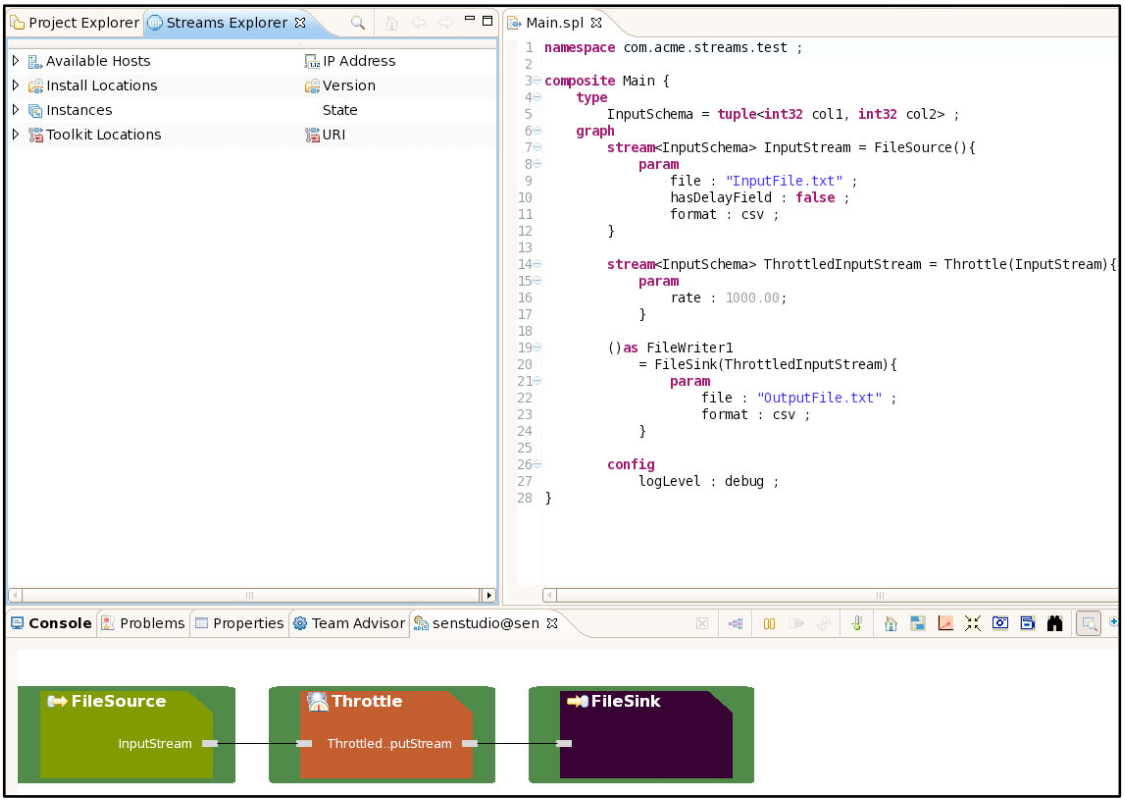


Figure 5-7 Streams Live Graph - No application running

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000284

Figure 5-8 displays the Streams Live Graph View with the correction to the input file name. The display in the Streams Live Graph View will change color to indicate the effect of throttledRate and the various pauses and then propagation of records.

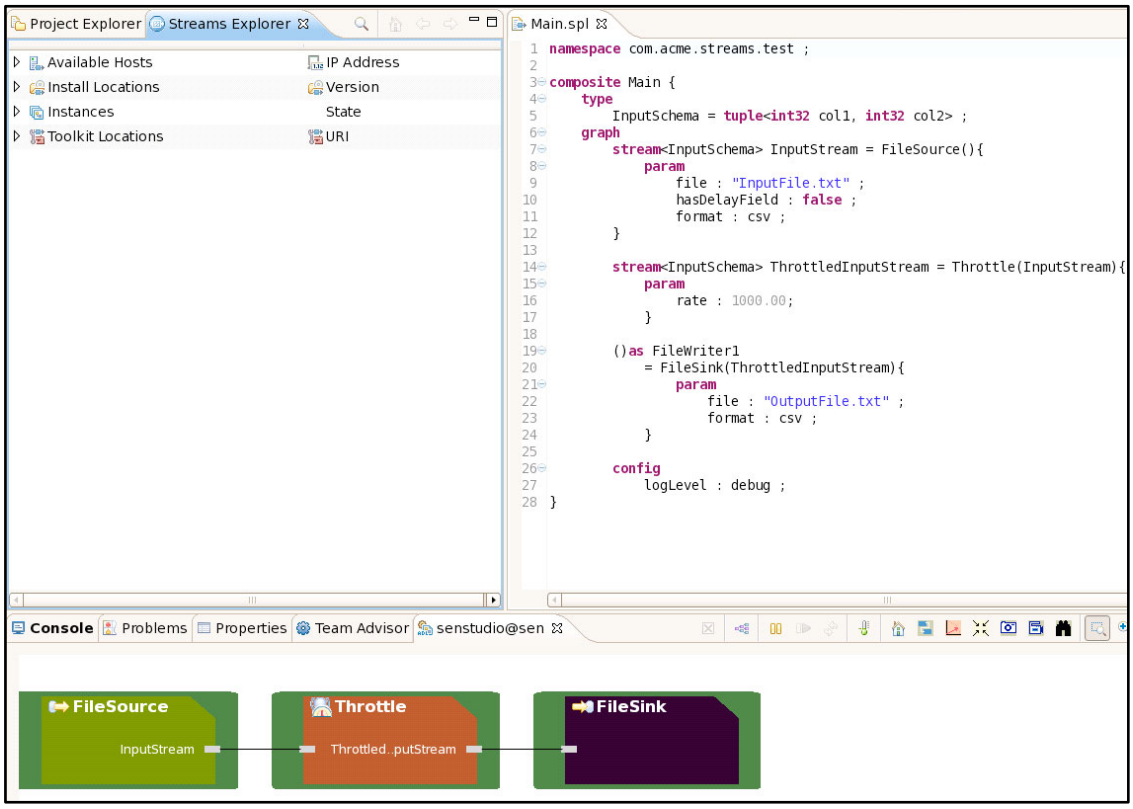
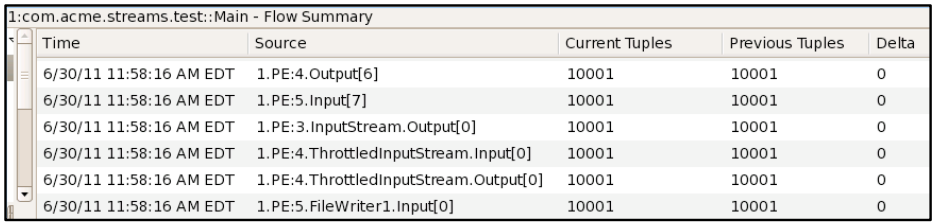


Figure 5-8 Streams Live Graph View with correction - Application running

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000285

Figure 5-9 displays the Streams Studio, Streams Live Graph Metrics View, where you can observe the live progress of records consumed and sent down stream.


1:com.acme.streams.test::Main - Flow Summary

Time	Source	Current Tuples	Previous Tuples	Delta
6/30/11 11:58:16 AM EDT	1.PE:4.Output[6]	10001	10001	0
6/30/11 11:58:16 AM EDT	1.PE:5.Input[7]	10001	10001	0
6/30/11 11:58:16 AM EDT	1.PE:3.InputStream.Output[0]	10001	10001	0
6/30/11 11:58:16 AM EDT	1.PE:4.ThrottledInputStream.Input[0]	10001	10001	0
6/30/11 11:58:16 AM EDT	1.PE:4.ThrottledInputStream.Output[0]	10001	10001	0
6/30/11 11:58:16 AM EDT	1.PE:5.FileWriter1.Input[0]	10001	10001	0

Figure 5-9 Streams Studio - Streams Live Graph View

5.6.2 Using streamtool to debug

In Figure 5-10, we added the Streams configuration directive debugging tool to the Application-level scope. This allows us to capture runtime diagnostics from this Streams application in the Streams system log. Using the Streams command-line utility streamtool, we can now view this diagnostic output.

For this example, we return to the condition where the input text file name is not found, because the file does not exist. Figure 5-10 displays the first step in gathering this diagnostic output.

```
@a0217b10e1/~
> streamtool man
usage: man [<cmd>]

Streamtool is the InfoSphere Streams command line interface.

The other major command line tool is "sc", the Stream Processing
Language (SPL) compiler.

Streamtool consists of a number of sub-commands.  Various sub-commands
have an abbreviated name.  The command's usage infomation provided by the
"man" sub-command identifies the minimal abbreviation that can be used
to specify the command.  A '/' character in name identifies the minimal
abbreviation. Sub-command names and their option arguments are case-sensitive.
```

Figure 5-10 Using streamtool to get the list of supported commands

With respect to Figure 5-10, consider the following items:

- streamtool is a command-line utility, so each of these commands is entered at the operating system command line prompt.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000286

Note: The following Streams-based terms are in effect here. Keep in mind that this section serves only as a brief review:

- A Streams Instance refers to the collection of memory, process, and disk that is the Streams server proper. Each Streams Instance is identified by an Instance ID.
- The source code to a self-contained Streams application exists in one operating system text file. A Streams application is identified by its Application Name, and by its source code file name (with a .spl file name suffix).
- Streams applications are submitted to run in the form of a Streams Job. One Streams application may be running many copies concurrently. That is, running many concurrent Jobs of the same Streams application.

  Each Streams Job is identified by a unique Job ID.
- A Streams Job is a logical term, and exists as one or more operating system processes, referred to by Streams as Processing Elements (PE). Each PE has a unique PE ID.

- We ran **`streamtool man`** to get the list of available commands. For example, you can query the Streams instances created by you by running **`streamtool lsinstance`** (where lsinstance means list instance).

To get the log for a given Processing Element (PE), you need the PE ID. Figure 5-11 displays the **`streamtool`** command used to get a listing of currently executing Streams Jobs and their associated Processing Elements.

```
@a0217b10e1/~
> streamtool lspes -i senstudio
Instance: senstudio@sen
  Id State      RC Healthy Host         PID JobId JobName                      Operators
   3 Running     - yes     b0401e0     9001     1 com.acme.streams.test::Main  InputStream
   4 Running     - yes     b0405e0    24569     1 com.acme.streams.test::Main  ThrottledInputStream
   5 Running     - yes     b0405e0    24570     1 com.acme.streams.test::Main  FileWriter1
```

Figure 5-11 Getting the Processing Element IDs

In regards to Figure 5-11:

- The syntax to the **`streamtool`** command invocation above is:

  `streamtool lspes -i {Instance Id}`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000287

- spes, entered as one word, is short for list Processing Element IDS.
- - i is followed by the Streams Instance ID gathered using the **`streamtool lsinstance command`**.

- The report shown in Figure 5-11 on page 267 has a two line format.
  - The Streams Job ID being tracked here is 1.
  - The Streams Job has three Processing Elements, as called for by its three operators, one input, one throttle, and one output (Figure 5-11 on page 267).
  - The PE ID for the FileSource operator is 3.
  - If you want to view the log for the FileSource operator, which has an error in its input file name (the file is not found), then you need a method to retrieve the log for PE ID 3.

The example **`streamtool`** command from the continuing example is:

```
streamtool viewlog   -i   <instanceId number here>   --pe  3
```

This command invokes the configured operating system editor of your choice. An example is shown in Figure 5-12.

```
30 Jun 2011 12:24:29.695 [10133] DEBUG spl_operator M[OperatorThread.cpp:run:46]  - Operator thread created
30 Jun 2011 12:24:29.695 [10133] DEBUG PEC M[PECServer.cpp:associateThread:756]  P[6] - associating thread with peid=6
30 Jun 2011 12:24:29.695 [10133] DEBUG PEC M[PECServer.cpp:associateThread:758]  P[6] - associated thread with peid=6
30 Jun 2011 12:24:29.696 [10133] DEBUG #spllog,InputStream,spl_operator M[InputStream.cpp:process:180]  - FileSource st
artup...
30 Jun 2011 12:24:29.696 [10133] DEBUG #spllog,InputStream,spl_operator M[InputStream.cpp:processOneFile:81]  - Using '
InputFile2.txt' as the workload file...
30 Jun 2011 12:24:29.697 [10133] INFO ex M[InputStream.cpp:processOneFile:85]  - Throwing Exception: SPLRuntimeFileSour
ceOperator (Msg: Failed to open input file 'InputFile2.txt', reason: No such file or directory.)
30 Jun 2011 12:24:29.698 [10133] DEBUG spl_operator M[OperatorThread.cpp:run:75]  - Exception received during oper.proc
ess(): SPL::SPLRuntimeFileSourceOperatorException (Failed to open input file 'InputFile2.txt', reason: No such file or
directory.) at 'void SPL::_Operator::InputStream::processOneFile(const std::string&)' [src/operator/InputStream.cpp:85]

Exception Code=NoMessageId with 0 substitution text strings
```

Figure 5-12 Using streamtool to get the PE log

In regards to Figure 5-12:

- The highlighted paragraph (denoted by a rectangle) reports the condition of the input text file not being found.
- The example uses the operating system editor emacs.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000288

5.6.3 Other debugging interfaces and tools

A Streams application can have user-defined code written in the C/C++ programming language or code written in the Java programming language. As Streams Studio is an Eclipse-based developers workbench, which includes an embedded Java visual debugger, it is usually used by users of Streams Studio to debug Java routines. There are also third-party tools that provide C/C++ language debuggers (with similar visualization) for the Eclipse developers workbench. Programming and debugging in C/C++ or Java is not expanded upon here.

A complied Streams application exists as an operating system binary program. Using specific Streams application compilation arguments, the operating system Gdb(C) program can be used to debug Streams applications. Debugging with Gdb(C) is not expanded upon further here.

A given Streams operator can have multiple consumers of its output. For example, multiple downstream operators can consume an upstream operator's output. During the debugging phase of Streams application development, it is common to insert Sink operators merely to dump Stream contents mid-application, solely for the purpose of debugging. These command lines can be left in place, and commented out before inserting the final Streams application into production.

Note: It is particularly useful to use the above technique to create redundant (test only) Sink operators when working with Streams Functor operators.

With the Streams Debugger, it is extremely easy to evaluate input and output records and columns to a given Streams operator. However, seeing inside a given Functor operator, and more specifically its flow control and variable assignments, is another challenge.

5.6.4 The Streams Debugger

Given the volume of information that has been written related to debugging a Streams application, you might think that there was not a debugger to be found inside the Streams product. However, this is not the case. The Streams Debugger is currently documented in *IBM InfoSphere Streams SPL Streams Debugger Reference*, which can be found at:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000289

Note the following items:

- As with other debuggers, the Streams Debugger offers break points, trace points, variable evaluation, and reassignment.

  Generally, a break point is the ability to suspend a program's execution on a given condition, and a trace point is the history of a given variable, meaning how and when it changed over time.

- We continue referring to Figure 5-12 on page 268 in the following discussion and we have set a break point when the input record has a value of 12000. We then update this value and continue the execution of the Streams application.

The Streams Debugger has a command-line interface. The given Streams application being debugged must be compiled with special arguments. As an exercise, you might try to complete the following steps:

1. Create the sample input file and sample Streams application described in Example 5-38 on page 260.
2. From the command line, compile this Streams application as follows:

   ```
   sc -M com.acme.streams.test::Main
   --output-directory=output/com.acme.streams.test.Main/Standalone
   --data-directory=data -T -g
   ```

   This command will output an operating system binary file with the name `{Streams Application File Name}`, with no file name suffix. It will be stored in the output directory inside of your application directory (`output/com.acme.streams.test.Main/Standalone/bin/standalone`).
3. Run the operating system binary file name of the file output in the previous step. That command will place you into the Streams Debugger command-line interface.

> **Note:** Where are all of these files?
>
> Given a user home directory of `/home/chuck/`, a Streams Project name of MyProject, and a Streams application entitled MyApplication, the full path name to the Streams Application File Name becomes:
>
> `/home/chuck/workspace/MyProject/MyApplication/MyApplication.spl`
>
> The **sc** command will output the operating system binary program to:
>
> `/home/chuck/workspace/MyProject/MyApplication/`
> `output/com.acme.streams.test.Main/Standalone/bin/standalone`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000290

4. Run the application ./standalone. Now, run the following Streams Debugger commands in sequence.

   a. Call for the output the operands for this given Streams application. Enter the following Streams Debugger command (which is a small letter “o”):

      o

      The output of this command is displayed in Figure 5-13.

```
IBM Stream Debugger (SDB), pid: 3768
30 Jun 2011 12:48:41.352 [3768] INFO spl_pe M[PEImpl.cpp:runDebugServices:710]  - Started debug services...
30 Jun 2011 12:48:41.352 [3768] INFO spl_pe M[PEImpl.cpp:runDebugServices:711]  - Notifying debug services of
 processing startup...
Standalone application execution is suspended.
Set initial probe points, then run "g" command to continue execution.
30 Jun 2011 12:48:41.353 [3773] INFO spl_app M[ProbePointServices.cpp:run:91]  - run - ProbePointServices sta
rting
(sdb) o
   #in  #out Operator                                    Class
   1    0    FileWriter1                                 FileWriter1
   0    1    InputStream                                 InputStream
   1    1    ThrottledInputStream                        ThrottledInputStream
(sdb)
```

Figure 5-13 Streams Debugger - Getting an operator ID

In regards to Figure 5-13:

- The **o** (output) command lists all of the operators for this given Streams application. Operator names are displayed as they were specified by the developer in the `.spl` file.
- The example has three operators: a FileSource operator, a Throttle operator, and a Sink operator.

  As a Source (device), the input stream only lists an input and will be referred to by its operator name (in this case, InputStream) and a numeric zero.

  As a destination (device), the Sink operator lists only output and will be referred to by its operator name (in the case, FileWriter1) and a numeric one.

  Because the Throttle operator has an input port and an output port, you will notice in Figure 5-13 that it shows a numeric zero for input and a numeric one for output.
- In the use case being created, we want to set a break point when a given condition occurs. That condition is when the input column on a given record is equal to the integer value of 12000.

   b. Create a break point by running the following Streams Debugger command:

```
b   InputStream o   0   p   col1   ==   12000
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000291

Where:

- `b` is the Streams Debugger command to create a new break point.
- InputStream o 0 is the identifier of the specific Streams operator and its specific input streams we want to monitor.
- `p` indicates we are entering a conditional break point, using the Perl(C) expression language. A conditional break point suspends execution of this Streams application when its associated expression evaluates to true.
- `col1` is a reference to the input value of col1 from InputStream.
- `== 12000` will be true when col1 equal 12000.

You must use two equal symbols here to test for equality. One equal symbol would set col1 equal to 12000, which would be true for every input record, which is not the intended goal.

Note: This evaluation expression is written in Perl(C). Perl(C) allows many types of constructs, such as multi-level if statements and complex variable assignments.

c. Call this program by entering the following Streams Debugger command:

```
g
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000292

This Streams application will run to completion or until the conditional break point that was recorded above is encountered. An example is displayed in Figure 5-14.

```
IBM Stream Debugger (SDB), pid: 13974
06 Sep 2011 13:41:55.226 [13974] INFO spl_pe M[PEImpl.cpp:runDebugServices:711]  - Started debug services...
06 Sep 2011 13:41:55.226 [13979] INFO spl_app M[ProbePointServices.cpp:run:91]  - run - ProbePointServices st
arting
06 Sep 2011 13:41:55.227 [13974] INFO spl_pe M[PEImpl.cpp:runDebugServices:712]  - Notifying debug services o
f processing startup...
Standalone application execution is suspended.
Set initial probe points, then run "g" command to continue execution.
(sdb) o
   #in  #out Operator                                    Class
   1    0    FileWriter1                                 FileWriter1
   0    1    InputStream                                 InputStream
   1    1    ThrottledInputStream                        ThrottledInputStream
(sdb)
(sdb)
(sdb) b InputStream o 0 p col1 == 12000
CDISP0727W WARNING: Input path '/homes/hny5/sen/InfoSphereStreams20_64/lib/spl/toolkit' is not a directory.
    Set +  0   Breakpoint  InputStream                                o   0     predicate:eval(col1 ==
    12000) stopped:false
(sdb)
(sdb)
(sdb) g
(sdb)
(sdb)
    +  0   Breakpoint  InputStream                                o   0     dropped:false
    predicate:eval(col1 == 12000) stopped:true
    col1, 12000, int32
    col2, 12000, int32
(sdb)
(sdb)
(sdb) u 0 col1 90000
    col1, 90000, int32
    col2, 12000, int32
(sdb)
(sdb)
(sdb) c
```

Figure 5-14 Remainder of Streams Debugger example

In regards to Figure 5-14:

- The entire example being created is displayed in Figure 5-14.
- Extra carriage returns are entered between commands, which is allowed.
- After a g (go) command, the Streams application executes until it encounters the conditional break point.

  The line in Figure 5-14 that begins with `+ 0 InputStream` identifies when the Streams application suspended execution on the arrival of the break point event.

  At this point, you could enter a variety of Streams Debugger commands, such as evaluate input columns, change the value of output columns, insert new output records, delete output records, and so on.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000293

- The line in Figure 5-14 on page 273 that reads `u 0 col1 90000` calls to update (u) the column entitled col1 in the zero-th input stream for this operator. This condition is reflected in the sample output displayed in the sample output file shown in Figure 5-15.
- In Figure 5-14 on page 273, the **c** command calls to continue execution of this Streams application, which it does to completion.
- The **q** command exits the Streams Debugger.

In Figure 5-15, we show a sample output file from the Streams application.

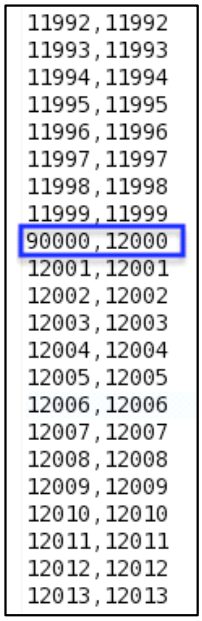


Figure 5-15 Sample output file

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000294

Note: Why stop program execution upon receipt of the value 12000?

The point is to demonstrate that using the Streams Debugger, you can halt execution of a Streams application on any single condition or even a set of conditions.

Why then set the new output column value to 90000?

You could use the Streams Debugger to set column values to such things as invalid values for testing or for duplicate values, potentially also for testing. You can additionally delete records from the output Streams using the **x** Streams Debugger command.

There are many, many more Streams Debugger commands than those listed here. Using the **savecfg** and **loadcfg** Streams Debugger commands, you can save and then load items, such as complex sets of break points and trace points, and use them for reference or automated testing of Streams applications.

Summary and further topics

In this chapter, we reviewed different aspects of the Streams programming model, including how some of the SPL standard toolkit operators work, developing simple SPL applications, and debugging the SPL applications. There is so much more to the Streams Processing Language. There are multiple documentation PDF files shipped with the InfoSphere Streams product, such as *SPL Language Reference*, *SPL Compiler Reference*, *SPL Configuration Reference*, *SPL Debugger Reference*, *SPL Operator Model Reference*, *SPL Operator Development Reference*, and others. Refer to those documents to get a complete overview of all the features available in the Streams programming mode.

There are many other advanced SPL concepts, including primitive operators in C++ and Java, that you should also take time to review. With the solid background you gained in this chapter on SPL basics, this is the right time for you to dive into the next chapter to learn about the advanced SPL topics.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000295

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000296

6

Advanced Streams programming

In the previous chapters, we described and discussed the concepts of streaming applications, the development process, and the fundamentals of the Streams Processing Language, including the operators in the standard toolkit. In this chapter, we present advanced topics for users who have already mastered writing SPL applications using the standard toolkit operators, and are ready to learn how to develop more powerful applications that take advantage of the extensibility mechanisms in Streams.

In the first section of this chapter, we discuss how to write native functions to extend the set of functions available for use in SPL expressions. Next, we give an overview of extending SPL through Primitive operators. In the remaining sections, we provide details about how to develop non-generic and generic C++ Primitive operators.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000297

6.1 Developing native functions

SPL comes with a set of functions that are part of the standard toolkit. These functions can be used in SPL expressions and statements. Furthermore, users can write their own functions using SPL statements. Such functions that are authored purely using the SPL language are called SPL functions. Although SPL functions are quire extensive, there are various reasons why applications may need to bring functions written in other languages into SPL, such as:

- General purpose programming languages often provide a rich set of language features as well as runtime libraries that are not directly available at the SPL level.
- There exists a large number of libraries and assets written in other programming languages, and rewriting them in SPL is not practical or sometimes not possible.

SPL supports registering functions written in C++, so they can be used from within SPL applications. These are called *native functions*. The general process of creating native functions is as follows:

- Design the SPL signatures of the native functions you want to add and write a sample application that showcases their use.
- Add a function model to your toolkit or application, which describes these native functions and their dependencies on external libraries.
- Write a C++ library that implements the native functions (or wrap an existing library that implements the core logic).

In the rest of this section, we showcase these steps with an example. We develop a native function called *pinToCpu* that will pin the current thread of execution to a given processor core on the system. This is a low-level functionality not directly available to SPL programmers without the use of a native function.

6.1.1 Signature and usage in a sample application

We first design the signature of a native function. The signature is simple, such as **`public stateful int32 pinToCpu(int32 core)`**. This function has a single integer parameter, which is the zero-based index of the core to which the thread will be pinned. The return value is the error code of the function. It will return zero on success and an error code otherwise. However, we ignore the details of the error code for the purpose of this discussion. We further decide that the native function will live in a namespace called *system*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000298

Next, we write a sample application that showcases the use of the function. The sample application is shown in Example 6-1.

Example 6-1 Application using a native function called pinToCpu

```
// file sample/ThreadPinner.spl
namespace sample;

composite ThreadPinner {
  graph
    stream<int8 dummy> Beat = Beacon() {
      param iterations : 1u;
    }
    () as Sink = Custom(Beat) {
      logic
        onTuple Beat: {
          int32 res = sys.helper::pinToCpu(1);
          assert(res==0);
          println("Pinned thread to CPU 1");
          // do more work
        }
    }
}
```

In this application, called ThreadPinner, there is a Beacon operator instance that is a Source operator and produces a single tuple. This tuple is used to start a Custom operator instance, which is a Sink operator. In the Custom operator, the current thread of execution is pinned to the CPU at index 1. This action is performed through a call to the sys.helper::pinToCpu function. At this time, this program will not compile, as the call to the pinToCpu function will not be resolved. Next, we see how this function can be registered with the SPL compiler.

6.1.2 Writing a function model

To register native functions with SPL, function models are used. A function model can be used to register several functions that are under the same namespace. A function model includes the following crucial information that establishes the mapping between the SPL signature and the native implementation of a function:

- The SPL signature of the native function
- The header file that provides the C++ declaration of the function
- Any library dependencies the function may have

The function model is an XML file. It always has the name `function.xml` and is always located under the directory `native.function`, which in turn is located

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000299

under a namespace directory. For the pinToCpu function, we have already decided that the function lives in the `sys.helper` namespace, and as a result, the function model is located at `sys.helper/native.function/function.xml`.

Example 6-2 shows the function model for the pinToCpu function. Any other native function to be registered under the system namespace can also go into this function model.

Example 6-2 The function model for the pinToCpu function

```
<!-- system/native.function/function.xml -->
<functionModel
  xmlns="http://www.ibm.com/xmlns/prod/streams/spl/function"
  xmlns:cmn="http://www.ibm.com/xmlns/prod/streams/spl/common"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/xmlns/prod/streams/spl/function
functionModel.xsd">
  <functionSet>
    <headerFileName>ThreadPinner.h</headerFileName>
    <functions>
      <function>
        <description>Pin to a given Cpu</description>
        <prototype><![CDATA[
            public stateful int32 pinToCpu(int32 cpu)
        ]]></prototype>
      </function>
    </functions>
    <dependencies>
      <library>
        <cmn:description>Thread pinning library</cmn:description>
        <cmn:managedLibrary>
          <cmn:includePath>../../impl/include</cmn:includePath>
        </cmn:managedLibrary>
      </library>
    </dependencies>
  </functionSet>
</functionModel>
```

In this function model, we have specified the header file `ThreadPinner.h` that declares the C++ function, and listed the SPL signature of the function.

Finally, we have specified a library dependency. In this case, the library is a header-only one, and thus there is only specified an include path. This path is relative to the location of the model file. For dependencies that require shared or

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000300

static libraries, additional details such as the library path and the name of the library should be specified. An example can be found in the $STREAMS_INSTALL/samples/spl/feature/NativeFunction sample that comes with the Streams product, as well in the *SPL Toolkit Development Reference*, which can be found at the following address:

http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_8

6.1.3 Implementing the native function in C++

To be able to implement a native function, you need to understand the following two mappings defined by the native function support of SPL:

- How SPL function signatures map to their C++ counterparts
- How SPL types map to their C++ counterparts

The namespace used for the SPL representation of the function and the C++ representation of it should match by default. Similarly, the name of the function in the SPL signature and the name used for the C++ function should match by default. A different C++ function name and a different C++ namespace can optionally be specified in the function model. All non-mutable SPL parameters are mapped to constant reference parameters in C++. All mutable SPL parameters are mapped to non-constant reference parameters in C++.

For each type in SPL, there is a matching type in C++. This mapping is given in the *SPL Toolkit Development Reference*. Most of the C++ types are either C++ standard library types, extensions to C++ standard library types, or types from well-established C++ libraries. For the pinToCpu function, the SPL int32 type maps to the int32_t type in C++.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000301

Example 6-3 provides the C++ implementation of the pinToCpu function. Note that the function is defined inside the `sys::helper` namespace and has the same name as the SPL function signature defined in the function model. The function is contained in a file named `ThreadPinner.h`, as indicated in the function model. Furthermore, the header file is located under the `impl/include` directory, which matches the include directory specified in the library dependencies section of the native function model. All native function implementation related artifacts should be placed under the `impl` directory, which is among the set of top-level application and toolkit directories recognized by the SPL tooling.

Example 6-3 Native function for pinning the current thread to a given CPU core

```
// file impl/include/ThreadPinner.h
#include <sched.h>
#include <sys/types.h>

namespace sys { namespace helper {
{
     inline int32_t pinToCpu(int32_t const & cpu)
     {
         cpu_set_t cpuSet;
         CPU_ZERO(&cpuSet);
         CPU_SET(cpu, &cpuSet);
         pid_t pid = gettid();
         return sched_setaffinity(pid, sizeof(cpu_set_t), &cpuSet);
     }
}}
```

6.1.4 Compiling and running the application

The application can be compiled by running **`sc -M sample::ThreadPinner -T`** as a stand-alone application and executed by running **`./output/bin/standalone`**.

6.1.5 Additional capabilities

There are several additional capabilities associated with native functions in SPL. Here we give a brief overview of them.

Generic types are types that are parameterizable through other types. As an example, collection types are generic types. A list type in SPL has a configurable element type, such as list of integers (`list<int32>`) or list of maps from strings to integers (`list<map<rstring, int32>>`). Such types naturally map to C++ counterparts that employ class templates, such as `SPL::list<int32_t>` and `SPL::list<SPL::map<SPL::rstring,int32_t>>`.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000302

It is often the case that you need to register native functions that take a generic type as a parameter. For example, a reverse function defined on a list should be able to work with any list. The C++ implementation of such a function can use function templates. However, given the infinite number of SPL function signatures this creates, registering these functions in the function model requires additional syntactic flexibility. Hence, SPL supports generics as part of native function signatures. For example, the SPL signature of the native function reverse is given as `<any T> public void reverse(mutable list<T> l)`. Here, `any` is a meta-type that represents any SPL type. Other meta-types, such as ordered or integral, exist as well. Additional details can be found in the *SPL Language Specifications*, which can be found at the following address:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_5`

Generated types are those types that do not have a predefined mapping in C++. The C++ versions of these types are generated during application compilation. As such, during native function development, such types are not accessible. Tuples (tuple) and enumerations (enum) in SPL are examples of these types. SPL provides the polymorphic base classes SPL::Tuple and SPL::Enum to work with types that contain tuples and enumerations as part of writing native functions. SPL C++ runtime APIs support runtime reflection on the value and structure of all SPL types. This support can be used to write native functions that inspect and manipulate types containing tuples and enumerations. Additional details can be found in *SPL Toolkit Development Reference* and *SPL Runtime C++ APIs*, the latter of which can be found at the following address:

`http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.doxygen.runtime.doc%2Fdoc%2Findex.html`

C++ types and functions defined in the SPL C++ runtime APIs are often useful in writing native functions. For example, the native functions in the SPL standard toolkit all have C++ implementations. By using these functions, you can write higher-level functions with ease. The header file `SPL/Runtime/Function/SPLFunctions.h` can be included to get access to these functions. Similarly, SPL C++ types can be accessed by including the header file `SPL/Runtime/Type/SPLType.h`. Additional details can be found in *SPL Runtime C++ APIs*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000303

Library dependencies are used to specify the name, include path, and library path of the external libraries required by the native functions. Some of these libraries are completely independent of SPL runtime APIs and the native function implementations simply provide a wrapper for them, performing the necessary type mappings and gluing. However, sometimes external libraries are written using the SPL runtime APIs and work with the SPL types. In such cases, the compilation of these libraries would require the necessary compiler flags, such as include paths, library paths, and library names. For this purpose, Streams provides the $(STREAMS_INSTALL)/bin/dst-pe-pkg-config.sh script that can be executed with the parameters --cflags dst-spl-pe-install to retrieve all the compiler flags necessary for compilation of SPL applications.

Another issue that arises with libraries is platform dependence. Most native libraries have separate versions for 32-bit and 64-bit platforms. There could be other variations of libraries that are dependent on platform specifics. To shield the SPL applications from these details, the native function developers can define a dependency script in the function model, rather than hardcoding the library names, include paths, and library paths. The dependency script will be called by the SPL compiler to retrieve this information at compile time. The dependency scripts can inspect the environment and return the correct names and the paths accordingly. Additional details can be found in *SPL Toolkit Development Reference*.

6.2 Overview of Primitive operators

Primitive operators are used in SPL to extend the set of operators provided by the standard toolkit with new cross-domain or domain-specific operators. These operators can have the same level of expressiveness as the standard toolkit operators and are built using the same underlying support for extending SPL.

Primitive operators are built using general purpose programming languages. C++ and Java are the languages supported for this purpose at this time. There are two main parts to a Primitive operator: the operator model and the operator implementation.

- The operator model is an XML document that describes semantic and syntactic properties of the operator, including what the operator expects from the compiler and the run time, as well as what behavior it promises to exhibit.
- The operator implementation provides the core logic of the operator. Depending on the kind of the Primitive operator being written, the implementation mechanisms differ. There are two kinds of Primitive operators: generic and non-generic.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000304

6.2.1 Generic Primitive operators

Generic primitive operators support a completely declarative interface and employ a compile time reflection and code generation mechanism to provide this interface. For example, a generic operator can take arbitrary expressions with references to stream attributes as parameters, have output assignments, and verify that its invocation is valid at compile time (beyond what can be verified with just the operator model).

As an example of a generic operator, consider the Join operator from the standard toolkit, which takes a match parameter. This parameter can be a Boolean expression containing references to stream attributes from the first and the second input port, describing the conditions under which two tuples are considered as correlated. Generic operators allow specifying such expressions to provide a truly declarative interface for operators.

Again consider the Join operator, but this time in equi-join configuration with the matchLHS and matchRHS parameters. These parameters should always coexist in the operator invocation. Furthermore, the number of values they have in an invocation and the pair-wise types of these values should always match. These kinds of verification checks can be performed at compile time with generic operators to make sure that the invocation is valid.

Generic operators are developed as code-generator templates. At compile time, code is generated by inspecting the configuration of the particular operator invocation at hand. As such, code-generator templates are a mix of generator code and template code. The generator code is used to inspect the operator invocation and generate code that is tailored to the operator invocation. The template code, conversely, makes it into the generated code as is. Code generator templates are automatically converted into code generators by the SPL tools, and are used to generate code for various invocations of a given operator found in SPLsource code. SPL supports generic operators through C++ only, where Perl is used as the generator language.

6.2.2 Non-generic Primitive operators

Non-generic Primitive operators are developed using only C++, without the involvement of code-generation. As a result, they lack some of the flexibility of generic Primitive operators. They can still encapsulate a great deal of reusable logic. Non-generic operators support basic parameterization, as well as a variable number of input and output ports with variable stream schemas.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000305

Non-generic operators rely on runtime reflection to achieve its goals. For example, a non-generic operator may take as a parameter the name of an attribute to inspect, and at run time inspect the incoming tuples to locate the given attribute, its type, and its value. Non-generic operators are supported through C++ and Java. In the case of C++, the operator code is still placed in a code generator template, but contains only C++ code. This setup provides a unified way of creating generic and non-generic C++ operators and simplifies moving from non-generic to generic operators. In the case of Java, operators are developed as Java classes in the traditional way.

6.3 Developing non-generic C++ Primitive operators

In this section, we describe how non-generic C++ Primitive operators are developed. The common process is as follows:

1. Create an SPL application that showcases the use or uses of the operator.
2. Create the skeletons for the operator model and the operator implementation.
3. Populate the operator model.
4. Populate the operator implementation.
5. Compile and test the application that uses the operator.

Let us look at these steps in detail as part of creating a source and a non-source operator.

6.3.1 Writing a Source operator

Here we develop, step-by-step, an operator called VMStatReader. This operator is a Source operator, that is, it does not have any input ports. It periodically reads metrics from the operating system's virtual memory subsystem and pushes them out as tuples. It is configurable with a time period, and uses the names of the output stream attributes to decide on the particular virtual memory statistics to read.

Using a Source operator in a sample application

Example 6-4 on page 287 shows a sample use of the VMStatReader operator as part of an application called sample::VMStat. In this application, the VMStatReader operator is configured with a period parameter, which takes its value from a submission time value of the same name. The output stream produced by the operator is named Stats, and has the following attributes:

- nr_page_table_pages
- nr_mapped, nr_slab
- pgalloc_high

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000306

These attributes are also the names of the metrics in the virtual memory subsystem. As a result, the operator implementation will inspect the output schema to discover which metrics to read. This operator will periodically output tuples that contain the most recent values of these metrics and the Custom operator will in turn write these metrics to the standard output (or print “no change” when the values have not changed since the last time).

Example 6-4 Calling a non-generic C++ Primitive operator from an application

```
// file sample/VMStat.spl
namespace sample;
composite VMStat {
  type
    StatType = tuple<uint64 nr_page_table_pages,
                     uint64 nr_mapped,
                     uint64 nr_slab,
                     uint64 pgalloc_high>;
  graph
    stream<StatType> Stats = VMStatReader() {
      param period: (float64)getSubmissionTimeValue("period");
    }

    () as Writer = Custom(Stats) {
      logic
        onTuple Stats: {
          if(Stats!=Stats[1])
            println(Stats);
          else
            printString("-- no change\n");
        }
    }
}
```

In this particular example, the output schema can be changed or a different invocation of the same operator with a different output schema can be used. As a result, we write the operator in a way that avoids hardcoding any of the virtual memory metric names into the code. Later in this section, we also look at examples where this level of flexibility is not needed and accessor methods (getters and setters) that depend on the names of the stream attributes can be used in the implementation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000307

Creating skeleton files

To create skeleton files for the operator model and the operator implementation (code generator templates), the SPL-Make-Operator tool can be used. The generated files can be used as a baseline on top of which the operator developer can make changes. The tool can take the following options:

- **`--kind`**: The kind of operator wanted (generic or non-generic).
- **`--directory`**: The directory where the operator artifacts will reside.

In this case, we want a non-generic C++ operator, which is specified by **`--kind c++`**. Because the operator is in the namespace sample, we specify its directory location by specifying **`--directory sample/VMStatReader`**. The optional **`--silent`** option is also specified so that we are not prompted for the creation of the skeleton files. The entire command is given as follows:

```
spl-make-operator --silent --kind c++ --directory sample/VMStatReader
```

After this command is run, the following files are created:

- `sample/VMStatReader/VMStatReader.xml`: The operator model.
- `sample/VMStatReader/VMStatReader_h.cgt`: The header file for the operator logic.
- `sample/VMStatReader/VMStatReader_cpp.cgt`: The implementation file for the operator logic.
- `sample/VMStatReader/VMStatReader(_cpp|_h).pm`: The header and implementation code generators. These are auto-generated files and are not edited by the operator developer.

Editing the operator model

Example 6-5 shows the operator model for the VMStatReader operator. Following the example, there are a few relevant notes:

Example 6-5 Operator model for the VMStatReader

```
<operatorModel
  xmlns="http://www.ibm.com/xmlns/prod/streams/spl/operator"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/xmlns/prod/streams/spl/operator
operatorModel.xsd">
  <cppOperatorModel>
    <context>

<providesSingleThreadedContext>Always</providesSingleThreadedContext>
    </context>
    <parameters>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000308

```
      <allowAny>false</allowAny>
      <parameter>
        <name>period</name>
        <optional>false</optional>
        <rewriteAllowed>true</rewriteAllowed>
        <expressionMode>AttributeFree</expressionMode>
        <type>float64</type>
      </parameter>
    </parameters>
    <inputPorts/>
    <outputPorts>
      <outputPortSet>
        <expressionMode>Nonexistent</expressionMode>
        <autoAssignment>false</autoAssignment>
        <completeAssignment>false</completeAssignment>
        <rewriteAllowed>true</rewriteAllowed>
        <windowPunctuationOutputMode>Free</windowPunctuationOutputMode>
        <tupleMutationAllowed>true</tupleMutationAllowed>
        <cardinality>1</cardinality>
        <optional>false</optional>
      </outputPortSet>
    </outputPorts>
  </cppOperatorModel>
</operatorModel>
```

- The model specifies that the VMStatReader operator always provides a single threaded context by specifying the providesSingleThreadedContext element. All instances of this operator, no matter how their invocation is configured, provide a single threaded context. An operator that provides a single threaded context does not make concurrent submission calls. A more detailed definition of single threaded context can be found in the *SPL Operator Model Reference*, which can be found at the following address:

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_4

- We have specified that the operator does not allow arbitrary parameters by specifying the allowAny element, because this operator will support a single parameter called period. This situation is specified through the parameter element. The parameter is not optional and has type float64. Its expression mode is AttributeFree, meaning that the input stream attributes cannot appear as part of the parameter value expression. Given that a source operator does not have input streams, this setting is a natural choice. Rewrite is allowed for this parameter, which means that the compiler is free to rewrite the parameter expression into a different but equivalent form.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000309

Enabling expression rewrite is always appropriate for non-generic operators, because they do not inspect the parameter value expressions at code-generation time and as such cannot tell the difference between the rewritten and the original expression. Enabling it provides the compiler additional flexibility to minimize the amount of code generated and improve the performance.

- Note that this operator does not have any input ports. However, it has a single output port. The ports are specified using port sets in the operator model. A port set is a series of ports that all share the same configuration. In this case, the operator has a single non-optional port set of cardinality 1 for the output ports. The cardinality of a port set element indicates the number of ports described by the port set, which is 1 in this case. The expression mode for the output port is Nonexistent, meaning that this operator does not support, at the operator invocation syntax level, making assignments to the output attributes on this port, which is the case for all non-generic operators. The auto-assignment is also set to false. This setting is useful when output port attributes are assigned from the matching attributes of the input ports. Because we have a Source operator, which does not have any input ports, this is set to false. The operator does not generate any window punctuations on its output port, so we have specified the window punctuation output mode as Free. We also specify that the operator permits the run time to modify the tuples submitted on this output port, which is why the tupleMutationAllowed option is set to true. When the operator allows the submitted tuples to be modified, the run time can be more efficient in how it passes tuples across operators. Additional details about port mutability can be found in the *SPL Toolkit Development Reference*.

Implementing the operator

A non-generic operator is implemented as a C++ class. This class overrides various virtual methods of the SPL::Operator class, and uses various runtime APIs to execute its core logic. For example, it implements process functions of the SPL::Operator class to provide handlers to act upon tuple and punctuation arrivals. It uses the submit functions of the SPL::Operator class to send tuples on a desired output port. It uses the runtime APIs related to type manipulation to inspect, create, and modify tuples.

Here are the commonly used SPL::Operator APIs in the Source operator:

- Constructor is used to inspect the operator parameterization and perform any resource initialization that is required for the operator's core logic.
- Destructor is used to finalize resources.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000310

- void allPortsReady() is a function that is overridden by an operator to notify it about the readiness of all operator ports. An operator must not make tuple submissions before all ports are ready, which is particularly important for Source operators. Unlike non-Source operators for which the arrival of an input tuple is indicative of port readiness, Source operators should wait for the allPortsReady call to start making tuple submissions.
- void prepareToTerminate() is a function that is overridden by an operator to notify it about the pending termination of the operator.
- void process(uint32_t index) is the function that will be executed by any threads that the operator may spawn.
- void submit(Tuple & tuple, uint32_t port) is a function that is called by an operator to submit tuples on a mutating or non-mutating output port.
- void submit(Tuple const & tuple, uint32_t port) is a function that is called by an operator to submit tuples on a non-mutating output port.

The header file for a non-generic operator contains the class definition. In the example, this file would be the `sample/VMStatReader/VMStatReader_h.cgt` file. This file will be processed by the SPL compiler and code will be generated from it. For our purposes, the SPL_NON_GENERIC_OPERATOR_HEADER_PROLOGUE and SPL_NON_GENERIC_OPERATOR_HEADER_EPILOGUE pragmas are used at the top and at the bottom of the file.

Example 6-6 shows the header file for the VMStatReader operator. Following the example, there are a few relevant notes.

Example 6-6 Header file for the VMStatReader operator

```
// file sample/VMStatReader/VMStatReader_h.cgt
#pragma SPL_NON_GENERIC_OPERATOR_HEADER_PROLOGUE
class MY_OPERATOR : public MY_BASE_OPERATOR {
public:
    MY_OPERATOR();
    void allPortsReady();
    void process(uint32_t);
protected:
    float64 period_;
};
#pragma SPL_NON_GENERIC_OPERATOR_HEADER_EPILOGUE
```

- The code is surrounded by the aforementioned pragmas. These pragmas cause generation of some boilerplate code. The generated boiler-plate code defines the macros MY_OPERATOR and MY_BASE_OPERATOR.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000311

Independent of the name of the non-generic operator, we have the same class declaration: class MY_OPERATOR : public MY_BASE_OPERATOR.

- For the VMStatReader operator, we need to implement the constructor, the allPortsReady function, and the process(uint32_t) function. The constructor is used to initialize the period parameter. The allPortsReady function is used to spawn an operator thread and the process function is used to perform the core operator logic in the context of that thread.

The implementation file for a non-generic operator contains the definitions of the class member functions. In the example, this file would be the `sample/VMStatReader/VMStatReader_cpp.cgt` file. This file will be processed by the SPL compiler and transformed into code finally digested by the C++ compiler. For our purposes, the SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_PROLOGUE and SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_EPILOGUE pragmas are used at the top and at the bottom of the file.

Example 6-7 shows the implementation file for the VMStatReader operator. For brevity, the code includes no error checking for the core operator logic. A more complete version of the code can be found with the $STREAMS_INSTALL/samples/spl/application/VMStat sample that comes with the Streams product

Following the example, we make a few observations about the implementation.

Example 6-7 Implementation file for the VMStatReader operator

```
// file sample/VMStatReader/VMStatReader_cpp.cgt
#pragma SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_PROLOGUE
#include <fstream>
using namespace std;

MY_OPERATOR::MY_OPERATOR() : MY_BASE_OPERATOR() {
    ConstValueHandle handle = getParameter("period");
    assert(handle.getMetaType()==Meta::Type::FLOAT64);
    period_ = handle;
}

void MY_OPERATOR::allPortsReady() {
    createThreads(1);
}

void MY_OPERATOR::process(uint32_t i) {
    ifstream f("/proc/vmstat");
    OPortOType otuple;
    while(!getPE().getShutdownRequested()) {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000312

```
        f.clear();
        f.seekg(0, ios::beg);
        while(!f.eof() && !getPE().getShutdownRequested()) {
            string metricName;
            uint64 metricValue;
            f >> metricName >> metricValue;
            if (f.eof()) break;
            TupleIterator iter = otuple.findAttribute(metricName);
            if(iter==otuple.getEndIterator()) continue;
            ValueHandle handle = (*iter).getValue();
            assert(handle.getMetaType()==Meta::Type::UINT64);
            uint64 & value = handle;
            value = metricValue;
        }
        submit(otuple, 0);
        getPE().blockUntilShutdownRequest(period_);
    }
    f.close();
}
#pragma SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_EPILOGUE
```

- As part of the constructor, we use the getParameter function to access the value of the period parameter. The parameter's value is returned as an ConstValueHandle, which is a handle to a value of any SPL type. This handle can be cast to a specific type if the type of the value it contains is known. In this particular case, the period parameter of type float64 is assigned from the handle after making sure that the handle is for a value of the same type. We could have accomplished the same task using the getParameter_period() function, which is a auto-generated helper API that returns a float64 value directly.
- We see that the allPortsReady function is used to create an operator thread by specifying the createThreads API call. Because source operators do not have input ports, they need a thread of their own to drive the processing. The thread runs the void process(int32_t index) function of the operator, where the index is the zero-based index of the operator thread.
- In the process function, we have a loop that runs until the application is shut down. The getPE() API call is used to get a reference to the ProcessingElement object, which in turn provides the API call getShutdownRequested() that checks if a shutdown has been requested. After each iteration of the loop, the operator sleeps using the blockUntilShutdownRequested() API call, passing it the sleep amount, for example, the value of period.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000313

- The output tuple is declared as `OPort0Type otuple;`. Here, OPort0Type is a typedef provided by the generated code. There is one such type defined for each input and output port.
- In each iteration of the loop, we read the virtual memory metrics from the `/proc/vmstat` file. For each metric, we check if there is a tuple attribute of the same name in the output tuple, using the findAttribute() API call provided by the tuple.
- The ValueHandle is used to access the tuple attribute's value and assign it from the metric value.
- The tuple is submitted after all the metrics have been processed, using the submit() API call.

Building and running the application

The application we have created can be built as a stand-alone application by running **`sc -M sample::VMStat -T`** and execute by running **`./output/bin/standalone period=0.5`**. The operator code is compiled when the application is compiled and any errors in the code will be reported on the `(_h|_cpp).cgt` files.

The output from running the application is as follows:

```
{nr_page_table_pages=13640,nr_mapped=32266,nr_slab=104357,pgalloc_high=0}
{nr_page_table_pages=13656,nr_mapped=32345,nr_slab=104358,pgalloc_high=0}
-- no change
-- no change
-- no change
{nr_page_table_pages=13667,nr_mapped=32345,nr_slab=104355,pgalloc_high=0}
-- no change
-- no change
-- no change
{nr_page_table_pages=13860,nr_mapped=32345,nr_slab=104354,pgalloc_high=0}
```

6.3.2 Writing a non-Source operator

Here we develop, step-by-step, an operator called Pinger. This operator is a non-Source operator, meaning it has input ports. It receives tuples that contain host names. It has a fixed schema for its single input port, which contains a single attribute called host, which is of type rstring. For each input tuple it receives, it produces one output tuple on its single output port. This port contains an additional attribute called ping of type rstring, which contains the results of running the ping1 utility on the given host.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000314

Using the Pinger operator in a sample application

In Example 6-8, we show a sample use of the Pinger operator as part of an application called sample::Ping. In this application, we have a Beacon operator that produces a single tuple that contains a host value that is read from a submission time argument. It is connected to the Pinger operator. The pinger operator's result is sent to a Custom operator, which writes the ping results to the standard output.

Example 6-8 Application using a non-generic C++ Primitive operator called Pinger

```
// file sample/Ping.spl
namespace sample;
composite Ping {
  param
    expression<rstring> $host : getSubmissionTimeValue("host");
  graph
    stream<rstring host> Host = Beacon() {
      param iterations : 1u;
      output Host: host = $host;
    }

    stream<rstring host, rstring ping> Result = Pinger(Host) {}

    () as Writer = Custom(Result) {
      logic
        onTuple Result :
          printStringLn(ping);
    }
}
```

In this particular example, the output schema and the input schema are fixed. As a result, we write the operator code by using the accessor methods (getters and setters) that depend on the names of the input and output stream attributes.

Creating skeleton files

To create skeleton files for the operator model and the operator implementation (code generator templates), the SPL-Make-Operator tool can be used, as we did before for the VMStat operator. The entire command is as follows:

```
spl-make-operator --silent --kind c++ --directory sample/Pinger
```

After this command is run, the following files are created:

- `sample/Pinger/Pinger.xml`: The operator model.
- `sample/Pinger/Pinger_h.cgt`: The header file for the operator logic.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000315

- `sample/Pinger/Pinger_cpp.cgt`: The implementation file for the operator logic.
- `sample/Pinger/Pinger(_cpp|_h).pm`:The header and implementation code generators. These are auto-generated files and are not edited by the operator developer.

Editing the operator model

In Example 6-9, we show the operator model for the Pinger operator. Following the example, there are a few relevant notes.

Example 6-9 Operator model for the Pinger operator

```
<operatorModel
  xmlns="http://www.ibm.com/xmlns/prod/streams/spl/operator"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/xmlns/prod/streams/spl/operator
  operatorModel.xsd">
  <cppOperatorModel>
    <context>

<providesSingleThreadedContext>Always</providesSingleThreadedContext>
    </context>
    <parameters>
      <allowAny>false</allowAny>
    </parameters>
    <inputPorts>
      <inputPortSet>
        <tupleMutationAllowed>false</tupleMutationAllowed>
        <windowingMode>NonWindowed</windowingMode>

<windowPunctuationInputMode>Oblivious</windowPunctuationInputMode>
        <cardinality>1</cardinality>
        <optional>false</optional>
      </inputPortSet>
    </inputPorts>
    <outputPorts>
      <outputPortSet>
        <expressionMode>Nonexistent</expressionMode>
        <autoAssignment>false</autoAssignment>
        <completeAssignment>false</completeAssignment>
        <rewriteAllowed>true</rewriteAllowed>

<windowPunctuationOutputMode>Preserving</windowPunctuationOutputMode>
        <tupleMutationAllowed>true</tupleMutationAllowed>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000316

```
        <cardinality>1</cardinality>
        <optional>false</optional>
      </outputPortSet>
    </outputPorts>
  </cppOperatorModel>
</operatorModel>
```

- The model specifies that the Pinger operator always provides a single threaded context by specifying the providesSingleThreadedContext element. All instances of this operator, no matter how their invocation is configured, provide a single threaded context. An operator that provides a single threaded context does not make concurrent submission calls. Because the operator will simply submit a tuple every time it receives a tuple, it is not involved in any asynchronous or concurrent processing. A more detailed definition of single threaded context can be found in the *SPL Operator Model Reference*.
- We have specified that the operator does not allow arbitrary parameters through the allowAny element. This is because this operator will not support any parameters, including ones that are not declared in the operator model (arbitrary ones).
- Note that this operator has a single input port. In particular, it is configured with a single non-optional input port set of cardinality 1. The port does not allow specification of windows, so it is specified to have a windowing mode of NonWindowed. It does not expect punctuations to work properly, so the window punctuation mode is specified as Oblivious. The operator will not modify the tuples it has received on this input port, so the tuple mutation allowed is set to false.
- This operator has a single output port. In particular, it is configured with a single non-optional output port set of cardinality 1. The expression mode for the output port is Nonexistent, meaning that this operator does not support, at the operator invocation syntax level, making assignments to the output attributes on this port. This is the case for all non-generic operators.
  The auto-assignment is also set to false, because non-generic operators do not support output attribute assignments at the invocation syntax level. The operator forwards any window punctuations it receives on its input port to its output port, so we have specified the window punctuation output mode as Preserving. We also specify that the operator permits the run time to modify the tuples submitted on this output port, which is why the tupleMutationAllowed option is set to true. When the operator allows the submitted tuples to be modified, the run time can be more efficient about how it passes tuples across operators. Additional details about port mutability can be found in the *SPL Toolkit Development Reference*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000317

Implementing the operator

As previously mentioned, the non-generic operator in this example is implemented as a C++ class. We now look at the commonly used SPL::Operator APIs in a non-Source operator. This list is complementary to the set of APIs that have already been introduced for the Source operators as part of the VMStatReader example.

- void process(Tuple & tuple, uint32_t port) is a function that is overridden by an operator so that it are notified about the arrival of tuples in its mutating input ports.
- void process(Tuple const & tuple, uint32_t port) is a function that is overridden by an operator so that it is notified about the arrival of tuples in its non-mutating input ports.
- void process(Punctuation const & tuple, uint32_t port)is a function that is overridden by an operator so that it is notified about the arrival of punctuations.

Example 6-10 shows the header file for the Pinger operator.

Example 6-10 Header file for the Pinger operator

```
// file sample/Pinger/Pinger_h.cgt
#pragma SPL_NON_GENERIC_OPERATOR_HEADER_PROLOGUE

class MY_OPERATOR : public MY_BASE_OPERATOR {
public:
    MY_OPERATOR() : MY_BASE_OPERATOR() {}
    void process(Tuple const & tuple, uint32_t port);
    void process(Punctuation const & punct, uint32_t port);
};

#pragma SPL_NON_GENERIC_OPERATOR_HEADER_EPILOGUE
```

For the Pinger operator, you need to implement the constructor, the void process(Tuple const &, uint32_t) function, and the void process (Punctuation const &, uint32_t) function. The constructor is simply empty. The tuple processing function is used to act on arrival of tuples. The punctuation processing function is used to forward window punctuations from the input port to the output port.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000318

Example 6-11 shows the implementation file for the Pinger operator. For brevity, the code includes no error checking for the core operator logic. A more complete version of the code can be found in the $STREAMS_INSTALL/samples/spl/application/Ping sample that comes with the Streams product.

Example 6-11 Implementation file for the Pinger operator

```
// file sample/Pinger/Pinger_cpp.cgt
#pragma SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_PROLOGUE

#include <cstdio>

static string getPingResult(string const & host);

void MY_OPERATOR::process(Tuple const & tuple, uint32_t port) {
    IPortOType const & ituple = static_cast<IPortOType const&>(tuple);
    OPortOType otuple;
    otuple.assignFrom(ituple, false);
    rstring const & host = ituple.get_host();
    otuple.get_ping() = getPingResult(host);
    // alternatively: otuple.set_ping(getPingResult(host));
    submit(otuple, 0);
}

string getPingResult(string const & host) {
    FILE * fp = popen(("/bin/ping -c 3 " + host).c_str(), "r");
    char buffer[1024];
    string result;
    while (fgets(buffer, sizeof(buffer), fp))
        result += buffer;
    pclose(fp);
    return result;
}

void MY_OPERATOR::process(Punctuation const & punct, uint32_t port) {
    if(punct==Punctuation::WindowMarker)
        submit(punct, 0);
}

#pragma SPL_NON_GENERIC_OPERATOR_IMPLEMENTATION_EPILOGUE
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000319

We use the process tuple method to act on the arrival of each tuple. For each tuple, we complete the following steps:

1. We cast the polymorphic tuple type to the concrete tuple type for the input port, that is, IPort0Type. The input tuple with the concrete type is represented as a local reference variable called ituple.
2. We declare a local variable otuple that is of type OPort0Type, which represents an output tuple.
3. We call the assignFrom() function of the output tuple to assign its attributes from the input tuple. The second argument to this function, which is false, indicates that the input tuple is not of the same type as the output, and thus only the attributes with matching names and types are to be assigned.
4. We call the get_host() function on the input tuple to access the host attribute value of the input tuple. This is a generated accessor method that returns an rstring as a result.
5. We pass the host name to a getPingResult function, which calls the external ping utility. The result from this function is assigned to the output tuple's ping attribute through the get_ping() function of the output tuple, which returns a reference to the ping attribute. This action could have been also achieved through a call to the set_ping() method. The former is preferable when the attribute contains nested types that need to be modified in-place due to performance considerations.
6. The output tuple is submitted through the submit() API call.

The operator code will be compiled when the application is compiled and any errors in the code will be reported in the `(_h|_cpp).cgt` files.

Building and running the application

The application can be built as a stand-alone application by running **`sc -M sample::Ping -T`** and executed by running **`./output/bin/standalone host='localhost'`**.

The output from running the application will look like as follows:

```
PING localhost.localdomain (127.0.0.1) 56(84) bytes of data.
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=1 ttl=64
time=0.018 ms
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=2 ttl=64
time=0.021 ms
64 bytes from localhost.localdomain (127.0.0.1): icmp_seq=3 ttl=64
time=0.022 ms
--- localhost.localdomain ping statistics ---
3 packets transmitted, 3 received, 0% packet loss, time 2000ms
rtt min/avg/max/mdev = 0.018/0.020/0.022/0.004 ms
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000320

6.3.3 Additional capabilities

There are several additional capabilities for the non-generic operator support of SPL. Here we give a brief overview of the most important ones.

- State can be built and manipulated as part of the operator logic. Class member variables can be used to store state. Each instance of an operator will have its unique state that is not available to others. When state is maintained by an operator, a pointer or a reference to the tuples received from a process call should never be stored as part of the state. The tuple object passed as a parameter to the process function is only valid during the lifetime of the process call. A copy (through the = or clone functions) can be made if the tuple is to be stored as part of the operator's state.
- Multi-threading is an important aspect of operator development. SPL operators can be run in a multithreaded context, where multiple threads are concurrently executing the process functions of an operator. As a result, operators that access and manipulate state should be written with concurrency in mind. SPL provides an AutoPortMutex class that can be used to protect state from concurrent access by multiple process calls running concurrently.

The AutoPortMutex is an intelligent synchronization primitive. Depending on whether a given instance of the operator is running in a multi-threaded context or not, it either reduces to an untaken branch or creates a mutual exclusion block.

A sample use case is shown in Example 6-12, where a counter (count_) is being maintained by an operator. Every time a tuple is received, the counter is incremented by the amount specified in one of the tuple attributes (increment), and a different tuple attribute (sum) is updated with the current value of the counter. Without the use of the AutoPortMutex for mutual exclusion around the use of the counter, the operator will result in data races in a multithreaded context.

Example 6-12 Use of AutoPortMutex to protect state

```
class MY_OPERATOR {
    ...
private:
    uint64_t count_;
    Mutex mutex_;
};

void MY_OPERATOR::process(Tuple & tuple, uint32_t port) {
    IPort0Type & ituple = static_cast<IPort0Type&>(tuple);
    { AutoPortMutex apm(mutex_, *this);
      // the apm object will acquire the lock if needed
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000321

```
      count_ += ituple.get_increment();
      ituple.set_sum(count_);
    } // the apm object will release the lock if needed
    submit(ituple, 0);
}
```

Blocking and shutdown handling should be performed with care. An operator should not block for a long time without special consideration to shutdown handling, as that might render the application unresponsive in the event of a shutdown. The prepareToShutdown Operator API can be used to handle this situation. An operator can override this method to be notified about the pending shutdown. This call is delivered asynchronously to the operator, and can be used to unblock any currently blocked process calls.

Similarly, Source operators or operator threads that have their own event loops should check if there is a pending shutdown at each iteration of the event loop. This task can be easily done by using the getPE().getShutdownRequested() API call.

Additional details and more advanced scenarios can be found in the *SPL Toolkit Development Reference*.

Punctuations come in two different varieties in Streams, namely window punctuations and final punctuations.

Window punctuations mark a window of tuples in a stream. Some operators require receiving streams that contain window punctuations to operate correctly. Some other operators generate punctuations on their output ports so that the operators expecting window punctuations can use these punctuations. Moreover, some operators can forward the window punctuations they receive, rather than generating their own, for use by the downstream operators.

On a per-port basis, an operator developer needs to determine the correct way to handle window punctuations, which relates to how the window punctuation modes of the input and output ports are specified in the operator model.

If an input port has a window punctuation mode of Oblivious, then the operator does not need to perform any action related to the internal logic of this operator when window punctuations are received on this port. However, if there is at least one output port, which specifies a window punctuation mode of Preserving and the window punctuations are to be preserved from the input port under consideration, then the window punctuations received on this input port should be handled, for the sole purpose of forwarding them. Window punctuations are not forwarded automatically, and it is the operator developer's responsibility to forward window punctuations.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000322

Example 6-13 shows a single input port, single output port operator, where the input port has a window punctuation mode of Oblivious and the output port has a window punctuation mode of Preserving. Note that it is the developer's responsibility to add the necessary forwarding logic into the operator code.

Example 6-13 Window punctuations

```
void MY_OPERATOR::process(Punctuation const & punct, uint32_t port)
{
    assert(port==0);
    if(punct==Punctuation::WindowMarker)
        submit(punct, 0);
}
```

If an input port has a window punctuation mode of Expecting, then the operator would perform actions related to the internal logic of this operator when window punctuations are received on this port. Like the Oblivious case, it may also need to perform forwarding depending on the window punctuation mode settings of the output ports.

In addition to the window punctuation mode of Preserving, there are two other possible modes for an output port, which are Free and Generating. An output port with a window punctuation mode of Free indicates that the operator never submits window punctuations on that output port. A mode of Generating, conversely, indicates that this operator generates and submits its own window punctuations on this output port. How and when the window punctuations are generated and submitted is up to the operator.

Final punctuations indicate that a stream will not contain any additional tuples after the final punctuation. If an operator receives a final punctuation on an input port, it will not receive any additional tuples. If an operator submits a final punctuation to an output port, future tuple submissions on that port will be ignored.

Final punctuations are automatically propagated by the run time. When the run time sees a final punctuation for all input streams connected to an input port, then the port is considered closed and a final punctuation is delivered to the operator on that input port. The operator may decide to take action upon receiving this punctuation using the appropriate process function. After the run time delivers a final punctuation to all of the input ports, a final punctuation will be sent on all output ports. This forwarding behavior can be changed using the operator model. The operator model enables the developer to specify the set of input ports on which final punctuations are to be received before a final punctuation is sent to a particular output port.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000323

Operators may need to handle final punctuations to drain buffers or signal asynchronous workers. For example, an operator that keeps a buffer of tuples on its input port might want to drain the buffer when it receives a final punctuation on this input port. If the buffer is being processed by a different thread, then the operator might want to wait for that thread to complete before returning from the process function associated with punctuations. After the operator code returns from the function, the run time will forward the final punctuation on the output port, effectively closing the port. Another common use case is when there is asynchronous threads blocked that are waiting for additional tuples to arrive on a given input port. When a final punctuation is received for the input port in question, such threads should be signaled.

Additional details and more advanced scenarios are covered in the *SPl Operator Model Reference* and *SPL Toolkit Development Reference*.

Advanced topics related to operator development, including but not limited to window handling, metrics access, checkpointing, dynamic connection handling, reflective tuple manipulation, and logging, can be found in the *SPL Toolkit Development Reference*.

6.4 Generic C++ Primitive operators

In this section, we describe how generic C++ Primitive operators are developed. The common process is as follows.

1. Create an SPL application that showcases the use or uses of the operator.
2. Create the skeletons for the operator model and the operator implementation.
3. Populate the operator model.
4. Populate the operator implementation (code generator template) using C++ as the target language and Perl as the generator language.
5. Compile and test the application that uses the operator.

We look at these steps in detail as part of a sample generic operator that can have parameters and output assignments that involve expressions with references to input tuple attributes.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000324

6.4.1 Writing a generic operator

We develop, step-by-step, an operator called Transform. This operator will be similar to the Functor Standard Toolkit operator. It will have a parameter called filter that will take a Boolean expression that specifies a filter to be used to decide which tuples will be output by the operator. It will also support output assignments. In summary, it is a combination of a relational projection and filter.

Using the Transform operator in a sample application

Example 6-14 shows a sample use of the Transform operator as part of an application called sample::Transformation. In this application, the Transform operator is used to filter and annotate tuples received from a file. The filtering is based on the value of a tuple's age attribute. The tuples are also annotated with an additional attribute called wisdom, which is set to the value of an expression that references the input attribute age. Finally, the tuples output from the Transport operator are written to the standard output.

Example 6-14 Application using a generic C++ Primitive operator called Transform

```
// file sample/Transformation.spl
namespace sample;
composite Transformation {
  graph
    stream<rstring name, int32 age> Src = FileSource() {
      param file : "folks.txt";
    }

    stream<Src, tuple<float64 wisdom>> Result = Transform(Src) {
      param filter: age > 20;
      output Result: wisdom = pow(2.0, (float64)age-20.0);
    }

    () as Writer = FileSink(Result) {
      param file : "/dev/stderr";
    }
}
```

Creating skeleton files

To create skeleton files for the operator model and the operator implementation (code generator templates), the SPL-Make-Operator tool can be used. The entire command is as follows:

```
spl-make-operator --silent --kind generic --directory sample/Transform
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000325

In this case we want a generic C++ operator, which is specified with --kind generic. This example is different than the previous examples we have seen.

After this command is run, the following files are created:

- `sample/Transform/Transform.xml`: The operator model.
- `sample/Transform/Transformh.cgt`: The header file for the operator logic.
- `sample/Transform/Transform cpp.cgt`: The implementation file for the operator logic.
- `sample/Transform/Transform(_cpp|_h).pm`: The header and implementation code generators. These are auto-generated files and are not edited by the operator developer.

There is one important difference in the generated skeletons compared to the case of non-generic operators: The code generator templates have their header and implementation prologues and epilogues as Perl functions rather than as pragmas. This is because for generic operators, Perl is used as the generator code, and there is no requirement to keep the code as pure C++. In fact, the flexibility of generic operators is strongly tied to the ability to perform compile time introspection. As mentioned before, this is achieved by mixing generator code (Perl) with the generated code (C++).

Editing the operator model

Example 6-15 shows the operator model for the Transform operator.

Example 6-15 Operator model for the Transform operator

```
<operatorModel
  xmlns="http://www.ibm.com/xmlns/prod/streams/spl/operator"
  xmlns:cmn="http://www.ibm.com/xmlns/prod/streams/spl/common"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.ibm.com/xmlns/prod/streams/spl/operator
operatorModel.xsd">
  <cppOperatorModel>
    <context>

<providesSingleThreadedContext>Always</providesSingleThreadedContext>
    </context>
    <parameters>
      <allowAny>false</allowAny>
      <parameter>
        <name>filter</name>
        <description>Condition that determines which input tuples are
to be operated on</description>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000326

```
        <optional>true</optional>
        <rewriteAllowed>true</rewriteAllowed>
        <expressionMode>Expression</expressionMode>
        <type>boolean</type>
        <cardinality>1</cardinality>
      </parameter>
    </parameters>
    <inputPorts>
      <inputPortSet>
        <description>Port that ingests tuples to be
manipulated</description>
        <tupleMutationAllowed>false</tupleMutationAllowed>
        <windowingMode>NonWindowed</windowingMode>

<windowPunctuationInputMode>Oblivious</windowPunctuationInputMode>
        <cardinality>1</cardinality>
        <optional>false</optional>
      </inputPortSet>
    </inputPorts>
    <outputPorts>
      <outputPortSet>
        <description>Port that produces generated tuples</description>
        <expressionMode>Expression</expressionMode>
        <autoAssignment>true</autoAssignment>
        <completeAssignment>true</completeAssignment>
        <rewriteAllowed>true</rewriteAllowed>

<windowPunctuationOutputMode>Preserving</windowPunctuationOutputMode>
        <tupleMutationAllowed>true</tupleMutationAllowed>
        <cardinality>1</cardinality>
        <optional>false</optional>
      </outputPortSet>
    </outputPorts>
  </cppOperatorModel>
</operatorModel>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000327

The following items are notes regarding Example 6-15 on page 306:

- The model specifies that the Transform operator always provides a single threaded context, through the providesSingleThreadedContext element. All instances of this operator, no matter how their invocation is configured, provide a single threaded context. An operator that provides a single threaded context does not make concurrent submission calls. Because the operator will simply submit a tuple every time it receives a tuple, it is not involved in any asynchronous or concurrent processing. A more detailed definition of single threaded context can be found in the *SPL Operator Model Reference*.
- We have specified that the operator does not allow arbitrary parameters through the allowAny element, which is because this operator will support only a single parameter called filter specified through the parameter element in the model. The filter parameter is marked as optional because the operator is designed to let all tuples through when the filter parameter is missing. The rewrite allowed is set to true, meaning that as the operator developer we are giving the compiler permission to rewrite the parameter value expression into something equivalent. This setup provides the compiler the ability to fold constants, as well as substitute them with variables that will be loaded at run time to enable code sharing across operator instances. From an operator developer's perspective, allowing rewrite is always the right choice as long as we are not planning to inspect the value of the expression at compile time and take action accordingly. In most cases, the operator developer will simply embed the value of the expression into the generated code, which implies that allowing rewrite is appropriate.
- Note that this operator has a single input port. In particular, it is configured with a single non-optional input port set of cardinality 1. The port does not allow specification of windows, so it is specified to have a windowing mode of NonWindowed. It does not expect punctuations to work properly, so the window punctuation mode is specified as Oblivious. The operator will not modify the tuples it has received on this input port, so the tuple mutation allowed is set to false.
- This operator has a single output port. In particular, it is configured with a single non-optional output port set of cardinality 1. The expression mode for the output port is Expression, which means that the operator supports arbitrary expressions. The auto assignment is set to true, which means that the output tuple attributes that are missing assignments will be assigned (by the compiler) from the attributes of the input tuples based on matching names and types. This situation does not mean that the operator developer is not responsible for generating code for making these assignments. It means that the compiler will rewrite the operator invocation to add the missing assignments.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000328

For example, in the example application, the output assignment will become output Result: wisdom = pow(2.0, (float64)Src.age-20.0), name=Src.name, age=Src.age;. The output port also allows rewrite of the assignment expressions by the compiler. The operator will forward any window punctuations it receives on its input port to its output port, so we also specify the window punctuation output mode as Preserving. We also specify that the operator permits the run time to modify the tuples submitted on this output port, which is why the tuple mutation allowed is set to true. When the operator allows the submitted tuples to be modified, the run time can be more efficient in how it passes tuples across operators. Additional details about port mutability can be found in *SPL Toolkit Development Reference*.

Implementing the operator

As previously mentioned, a generic operator is implemented as a code generator template. The template can have code segments in Perl, which are embedded inside of <% and %> tags. The Perl code has access to a $model object, which provides details about the operator invocation for which we are generating code. This object is called the operator instance model. The kinds of information that are available from the operator instance mode include, but are not limited to:

- The parameters of the operator, including the expressions that represent parameter values.
- The input ports of the operator, including the windowing configurations, the tuple types for the ports, and input port attributes.
- The output ports, including the tuple types of the ports, the output attributes, and their assignment expressions.

The complete details of the API can be found in the *SPL Code Generation Perl API Documentation*, which can be found at the following address:

http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.doxygen.codegen.doc%2Fdoc%2Findex.html

Example 6-16 shows the header file for the Transform operator.

Example 6-16 Header file for the Transform operator

```
// file sample/Transform/Transform_h.cgt
<%SPL::CodeGen::headerPrologue($model);%>

class MY_OPERATOR : public MY_BASE_OPERATOR
{
public:
   MY_OPERATOR()
      : MY_BASE_OPERATOR() {}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000329

```
   void process(Tuple const & tuple, uint32_t port);
   void process(Punctuation const & punct, uint32_t port);
};

<%SPL::CodeGen::headerEpilogue($model);%>
```

For the Transformer operator, we need to implement the constructor, the void process(Tuple const &, uint32_t) function, and the void process(Punctuation const &, uint32_t) function. The constructor is simply empty. The tuple processing function is used to act on arrival of tuples. The punctuation processing function is used to forward window punctuations from the input port to the output port.

Example 6-17 shows the implementation file for the Transform operator.

Example 6-17 Implementation file for the Transform operator

```
// file sample/Transform/Transform_cpp.cgt

<%SPL::CodeGen::implementationPrologue($model);%>

<%
   my $inputPort = $model->getInputPortAt(0);
   my $outputPort = $model->getOutputPortAt(0);
   my $inTupleName = $inputPort->getCppTupleName();
   my $filterParam = $model->getParameterByName("filter");
   my $filterExpr = $filterParam ?
$filterParam->getValueAt(0)->getCppExpression() : "true";
%>

void MY_OPERATOR::process(Tuple const & tuple, uint32_t port)
{
   IPortOType const & <%=$inTupleName%> = static_cast<IPortOType
const&>(tuple);
   if (<%=$filterExpr%>) {
       <%SPL::CodeGen::emitSubmitOutputTuple($outputPort,
$inputPort);%>
   }
}

void MY_OPERATOR::process(Punctuation const & punct, uint32_t port)
{
   forwardWindowPunctuation(punct);
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000330

```
<%SPL::CodeGen::implementationEpilogue($model);%>
```

We open a Perl segment and initialize a few variables that will be used later in the code. We extract the input and output port objects from the operator instance model. Then we extract the name used in C++ expressions associated with the parameter values to reference the current input tuple. This name is stored in the Perl variable $inTupleName. We then extract the C++ expression used as the filter parameter's value, or true if the filter parameter is not supplied. The C++ expression is stored in the Perl variable $filterExpr. This C++ expression has references to the attributes from the current input tuple, where the name of the variable representing the current input tuple in the C++ expression is given by $inTupleName.

In the process method, we first cast the current input tuple to its specific type, and also create a local reference variable with the name $inTupleName. This action ensures that the filter expression can be embedded into the generated code as is, where the references to the input tuple attributes will bind to the reference variable we have just created.

We then embed the filter expression into an IF statement to check if the current tuple passes the filter or not. If it passes, we then call a code generation helper function named SPL::CodeGen::emitSubmitOutputTuple. This function takes as input the object that represents the input port from which the tuple has been received, as well as the object that represents the output port to which the tuple is to be submitted. It examines the output attribute assignments and generates code that creates and submits an output tuple that is populated according to the output attribute assignments provided in the operator invocation.

Example 6-18 illustrates the generated C++ code for the process function, using the invocation from the sample application.

Example 6-18 Generated C++ code for the process function

```
void SPL::_Operator::Result::process(Tuple const & tuple, uint32_t
port)
{
    IPort0Type const & iport$0 = static_cast<IPort0Type const
&>(tuple);
    if ((iport$0.get_age() > lit$0)) {
        OPort0Type otuple(iport$0.get_name(), iport$0.get_age(),
            ::SPL::Functions::Math::pow(lit$1,
                (::SPL::spl_cast<SPL::float64, SPL::int32>::cast(
                    iport$0.get_age()) - lit$2)));
        submit(otuple, 0);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000331

```
    }
}
```

The code generator template is converted to a code generator, either as a result of rerunning the SPL-Make-Operator tool with only the directory parameter, as in **`spl-make-operator --directory sample/Transform`**, or by compiling the SPL application that uses the operator. In case there are errors in the Perl code used to generate C++ code, these errors will be reported on the code generation templates, that is, .cgt files.

However, if there are any errors in the C++ code, these errors will be reported on the generated C++ code files (unlike non-generic C++ operators).

Building and running the application

The application we created can be built as a stand-alone application by running **`sc -M sample::Transformation -T`** and executed by running **`./output/bin/standalone`**. The output from running the application will look like the following (assuming there is an appropriate input file that is named `data/folks.txt`):

```
// data/folks.txt
Fred,15
Foo,100
Bar,80
// program output
"Foo",100,2160228.4620103
"Bar",80,56347.5143531667
```

6.4.2 Additional capabilities

There are several additional capabilities for generic operator support of SPL. In this section, we give a brief overview of these capabilities.

Error reporting APIs exist to print error messages with line and column numbers during code generation, which enables operator developers to perform additional checks during code generation to verify that the operator invocation at hand is correct. Although operator models can be used to specify many syntactic properties of the operators, this is not always sufficient to handle all cases. As an example, the Filter operator from the standard toolkit has to make sure that the tuple types for the input and the output ports are the same. An example code segment for checking this kind of requirement and reporting it as an error is illustrated in the following:

```
my $inputPort = $model->getInputPortAt(0);
my $outputPort = $model->getOutputPortAt(0);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000332

```
my $inTupleType = $inputPort->getCppTupleType();
my $outTupleType = $outputPort->getCppTupleType();
SPL::CodeGen::exitln("Output stream type must match the input one.",
        $outputPort->getSourceLocation()) if($inTupleType ne $outTupleType);
```

In the above example, the tuple type of the input and the output ports are compared, and if they are not the same, an error is printed. It is important to note that the SPL::CodeGen::exitln function is used for this purpose and in addition to the error text, the source code location of the error is passed as an argument. The model object that represents the operator instance for which we are generating code includes source location information for each aspect of the operator invocation (for example, $outputPort->getSourceLocation() for the output port). The above code generates the following message for an incorrect invocation of the Filter operator:

- `FilterTest.spl:5:9: ERROR: Output stream type must match the input one.`
- `FilterTest.spl:5:9: CDISP0232E ERROR: A user error was encountered while generating code for Operator 'B'.`

It is common practice to include code checking functionality in a separate Perl module and call it from the `_h.cgt` and `_cpp.cgt` files. For example, the Filter operator uses a Perl module called FilterCommon in a file called `FilterCommon.pm` and makes calls to a function called verify defined in this file. The following code snippet illustrates this setup:

```
<%
   use FilterCommon;
   FilterCommon::verify($model);
%>

<%SPL::CodeGen::headerPrologue($model);%>
class MY_OPERATOR : public MY_BASE_OPERATOR
{
  ...
};
<%SPL::CodeGen::headerEpilogue($model);%>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000333

Expression mode is used to specify the valid forms the parameter values can take. The options include Expression, which is the most general form where the values can be free form expressions that could reference tuple attributes, AttributeFree, which is an expression that does not include any references to stream attributes, Constant, which covers expression that can be folded down to a constant at compile time, and CustomLiteral, which covers literals defined in the operator model. For expressions that can have references to attributes, the operator model also allows you to specify an input port scope, so that the attribute references can be limited to a certain port.

Custom literals are slightly different than the other categories, as they are not just restrictions on the expressions that are allowed, but also an extension point where the operator model specifies a set of literals that can appear in place of the expression. For example, the FileSource operator defines FileFormat in the operator model as a custom literal type with valid literal values of csv, txt, bin, and line. It defines a parameter called format whose expression mode is CustomLiteral and type is FileFormat. Parameters like format can be used to generate different code at compile time for different values of the parameter in the current invocation.

Expression rewrite is a concept closely tied to the expression modes. Both parameter expressions and output attribute assignment expressions can specify, in the operator model, whether expression rewrite is allowed or not. If an operator developer is planning to inspect the content of an expression at code generation time and generate code conditional on the content of the expression, then the expression rewrite must be set to false. When expression rewrite is set to true, then the compiler is free to rewrite the expressions. It usually performs two kinds of rewrites:

- It could fold sub-expressions into constants (for example, pow(2, 3)+1 might be folded to 9).
- It could replace constants with place holder variables that will be initialized at run time. This second optimization is targeted at reducing the amount of generated code.

For example, consider the following two instances of the Filter operator:

```
stream<T> Out1 = Filter(In) {
  param filter : a < 10;
}
stream<T> Out1 = Filter(In) {
  param filter : a < 20;
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000334

The SPL compiler can rewrite these expressions to a common form by replacing the literals 10 and 20 with a placeholder variable whose value is initialized at run time to the appropriate value for the operator instance. As a result, code generation will be performed once for these two instances of the Filter operator.

There are a few important rules to follow with respect to expression rewrite:

- Set rewrite to true as long as you are not going to generate code that is conditional on the contents of the expressions.
- When rewrite is set to true, do not use getSPLExpression API on anything that involves code generation (using it for error reporting is the only common use case). getCPPExpression API can safely be used because the C++ expression will be the same for different instances of the operator that are sharing their code.

Output assignments that are supported by generic operators come in two varieties:

- Plain assignments
- Assignments with output functions

The Transform operator we developed in this section has plain output assignments. For each output attribute, the operator invocation can specify an expression that is used to set the value of the output attribute. The $model object that represents the operator instance model has APIs to access all the output ports, the output attributes on each port, and the set of assignments for each output attribute. Although we have used the SPL::CodeGen::emitSubmitOutputTuple helper function for generating the code for performing the output assignment as well as the tuple submissions in the Transform operator implementation, we could have performed it in a more explicit way by iterating over all the assignment expressions and generating the code manually.

Output assignments with output functions are similar, but support output functions that can only appear at the outermost level and can take as parameters one or more expressions. Different than regular functions, the output functions have special meaning to the operator. For example, an Aggregate operator can define Max as an output function. In this case, the operator uses this function to compute the maximum value for the expression specified as a parameter to the Max function, across a set of input tuples. Here is an example:

```
stream<rstring name, int32 age> Src = MySource() {}
stream<rstring name, int32 age> Res = Aggregate(Src) {
  window
   Src: tumbling, count(10);
  output
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000335

```
    Res: age = Max(age), name = ArgMax(age, name);
}
```

In this example, the Max and ArgMax output functions are used to specify that the value of the age output attribute be set to the maximum of the values of the age attributes in the set of tuples inside the operator's window. Similarly, the value of the name output attribute is set to the value of the name attribute of the tuple inside the operator's window that has the highest value for the age attribute.

Another use case for output functions is to extract information that is otherwise internal to the operator. For example, the DirectoryScanner operator uses the Ctime output function, which enables application developers to set the value of an output tuple attribute to the change time of the file which name is produced as a tuple. Here is an example:

```
stream<rstring name, uint64 ctime> Src = DirectoryScanner() {
  param
   directory : "/tmp/work";
   pattern   : "^work.*";
  output
   Src : name = FileName(), ctime = Ctime();
}
```

Custom output functions are specified in the operator model, using the same function signature syntax used for native functions. For output tuple attributes that are missing assignments or that do have assignments but are missing an output function, a default output function can be specified in the operator model.

The Perl APIs for the operator instance model provide accessor methods to get the output functions and the expressions that are passed to them as parameters. Additional details can be found in *SPL Toolkit Development Reference*, *SPL Operator Model Reference*, and *SPL Code Generation Perl API Documentation*.

Windowing APIs at the code generation level are provided by SPL to ease the creation of operators that can work with a variety of windowing configurations. Their use is described in *SPL Toolkit Development Reference*.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000336

7

Streams integration approaches

In the previous chapters of this book, we have described and discussed the architecture and components of the IBM InfoSphere Streams products and applications. In this chapter, we look at how to integrate a Streams application with a few several other current technologies. This information provides the developer with guidance about how to approach integrating Streams with other analytic technologies, such as data model scoring and reading and writing data to and from data storage.

We have organized the chapter contents into three sections. First, we discuss the approach of integrating Streams with another key component of the IBM Big Data Initiative, BigInsights.

Then we focus on implementing real-time scoring in a Streams application by integrating it with the IBM key data mining product, IBM SPSS Modeler.

Finally, we discuss approaches for reading and writing data from and to key database products, such as DB2, solidDB, and Netezza®.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000337

7.1 Streams integration with IBM BigInsights

With the growing use of digital technologies, the volume of data generated by our society is exploding into the exabytes. With the pervasive deployment of sensors to monitor everything from environmental processes to human interactions, the variety of digital data is rapidly encompassing structured, semi-structured, and unstructured data. Finally, with better pipes to carry the data, from wireless to fiber optic networks, the velocity of data is also exploding (from a few kilobits per second to many gigabits per second). We call data with any or all of these characteristics *Big Data*. Examples include sources such as the internet, web logs, chat, sensor networks, social media, telecommunications call detail records, biological sensor signals (such as EKG and EEG), astronomy, images, audio, medical records, military surveillance, and eCommerce.

With the ability to generate all this valuable data from their systems, businesses and governments are grappling with the problem of analyzing the data for two important purposes: To be able to sense and respond to current events in a timely fashion, and to be able to use predictions from historical learning to guide the response. This situation requires the seamless functioning of data-in-motion (current data) and data-at-rest (historical data) analysis, operating on massive volumes, varieties, and velocities of data. How to bring the seamless processing of current and historical data into operation is a technology challenge faced by many businesses that have access to Big Data.

In this section, we focus on the IBM leading Big Data products, namely IBM InfoSphere Streams and IBM InfoSphere BigInsights, which are designed to address those current challenges we just presented. InfoSphere BigInsights delivers an enterprise-ready big data solution by combining Apache Hadoop, including the MapReduce framework and the Hadoop Distributed File Systems (HDFS), with unique technologies and capabilities from across IBM. Both products are built to run on large-scale distributed systems, designed to scale from small to very large data volumes, handling both structured and unstructured data analysis. In this first section, we describe various scenarios where data analysis can be performed across the two platforms to address the Big Data challenges.

7.1.1 Application scenarios

The integration of data-in-motion (InfoSphere Streams) and data-at-rest (BigInsights) platforms addresses three main application scenarios:

- Scalable data ingest: Continuous ingest of data through Streams into BigInsights.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000338

- Bootstrap and enrichment: Historical context generated from BigInsights to bootstrap analytics and enrich incoming data on Streams.
- Adaptive analytics model: Models generated by analytics such as data-mining, machine-learning, or statistical-modeling on BigInsights used as basis for analytics on incoming data on Streams and updated based on real-time observations.

These interactions are shows in Figure 7-1 and explained in greater detail in the subsequent sections.

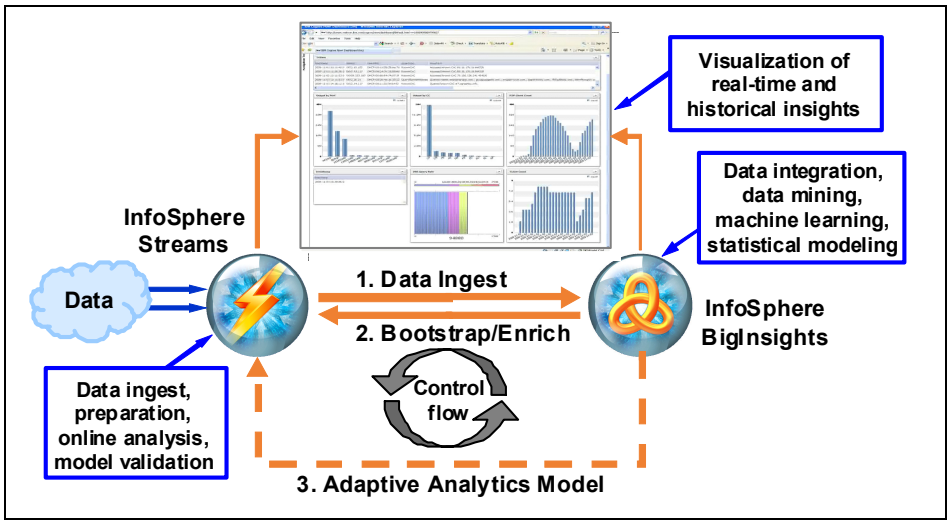


Figure 7-1 Big Data application scenarios

7.1.2 Large scale data ingest

Data from various systems arrives continuously, that is, as a continuous stream, as a periodic batch of files, or other means. Data needs to first be processed for extracting all the required data for consumption by downstream analytics. Data-preparation steps include operations such as data-cleansing, filtering, feature extraction, deduplication, and normalization. These functions are performed on InfoSphere Streams. Data is then stored in BigInsights for deep analysis and also forwarded to downstream analytics on Streams. The parallel pipeline architecture of Streams is used to batch and buffer data and, in parallel, load it into BigInsights for best performance.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000339

An example of this function is clear in the Call Detail Record (CDR) processing use case. CDRs come in from the telecommunications network switches periodically as batches of files. Each of these files contains records that pertain to operations such as call initiation, and call termination for telephones. It is most efficient to remove the duplicate records in this data as it is being ingested, because duplicate records can be a significant fraction of the data that will needlessly consume resources if post-processed. Additionally, telephone numbers in the CDRs need to be normalized and data appropriately prepared to be ingested into the back end for analysis. These functions are most efficiently performed using Streams.

Another example can be seen in a social media based lead-generation application. In this application, unstructured text data from sources such as Twitter and Facebook is ingested to extract sentiment and leads of various kinds. In this case, a lot of resource savings can be achieved if the text extraction is done on data as it is being ingested and irrelevant data such as spam is eliminated. With volumes of 140 million tweets every day and growing, the storage requirements can add up quickly.

7.1.3 Bootstrap and enrichment

BigInsights can be used to analyze data over a large time window, which it has assimilated and integrated from various continuous and static data sources. Results from this analysis provide contexts for various online analytics and serves to bootstrap them to a well-known state. They are also used to enrich incoming data with additional attributes required for downstream analytics.

As an example from the CDR processing use case, an incoming CDR may only list the phone number to which that record pertains. However, a downstream analytic may want access to all phone numbers a person has ever used. At this point, attributes from historical data are used to enrich the incoming data to provide all the phone numbers. Similarly, deep analysis results in information about the likelihood that this person may cancel their service. Having this information enables an analytic to offer a promotion online to keep the customer from leaving the network.

In the example of the social media based application, an incoming Twitter message only has the ID of the person posting the message. However, historical data can augment that information with attributes such as *influencer*, giving an opportunity for a downstream analytic to treat the sentiment expressed by this user appropriately.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000340

7.1.4 Adaptive analytics model

Integration of the Streams and BigInsights platforms enables interaction between data-in-motion and data-at-rest analysis. The analysis can use the same analytic capabilities in both Streams and BigInsights. It not only includes data flow between the two platforms, but also control flows to enable models to adapt to represent the real-world accurately, as it changes. There are two different interactions:

- BigInsights to Streams Control Flow: Deep analysis is performed using BigInsights to detect patterns on data collected over a long period of time. Statistical analysis algorithms or machine-learning algorithms are compute-intensive and run on the entire historical data set, in many cases making multiple passes over the data set, to generate models to represent the observations. For example, the deep analysis may build a relationship graph showing key influencers for products of interest and their relationships. After the model has been built, it is used by a corresponding component on Streams to apply the model on the incoming data in a lightweight operation. For example, a relationship graph built offline is updated by analysis on Streams to identify new relationships and influencers based on the model, and take appropriate action in real time. In this case, there is control flow from BigInsights to Streams when an updated model is built and an operator on Streams can be configured to pick up the updated model mid-stream and start applying it to new incoming data.
- Streams to BigInsights Control Flow: After the model is created in BigInsights and incorporated into the Streams analysis, operators on Streams continue to observe incoming data to update and validate the model. If the new observations deviate significantly from the expected behavior, the online analytic on Streams may determine that it is time to trigger a new model-building process on BigInsights. This situation represents the scenario where the real-world has deviated sufficiently from the model's prediction that a new model needs to be built. For example, a key influencer identified in the model may no longer be influencing others or an entirely new influencer or relationship can be identified. Where entirely new information of interest is identified, the deep analysis may be targeted to just update the model in relation to that new information, for example, to look for all historical context for this new influencer, where the raw data had been stored in BigInsights but not monitored on Streams until now. This situation allows the application to not have to know everything that they are looking for in advance. It can find new information of interest in the incoming data and get the full context from the historical data in BigInsights and adapt its online analysis model with that full context. Here, an additional control flow from Streams to BigInsights is required in the form of a trigger.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000341

7.1.5 Application development

This section describes how an application developer can create an application spanning two platforms to give timely analytics on data in motion while maintaining full historical data for deep analysis. We describe a simple example application that demonstrates the interactions between Streams and BigInsights. This simple application tracks the positive and negative sentiment being expressed about products of interest in a stream of emails and tweets. An overview of the application is shown in Figure 7-2.

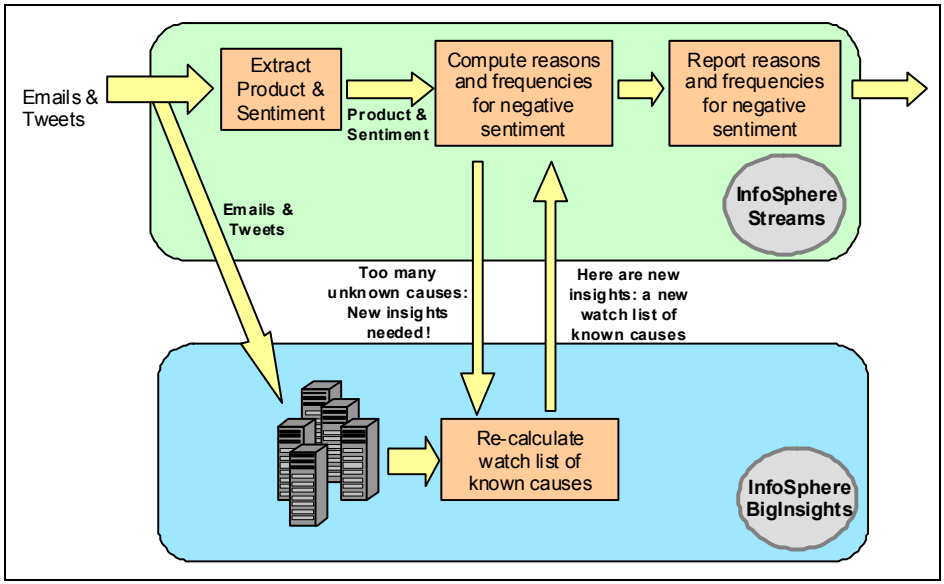


Figure 7-2 Streams and BigInsights application example

Each email and tweet on the input streams is analyzed to determine the products mentioned and the sentiment expressed. The input streams are also ingested into BigInsights for historical storage and deep analysis. Concurrently, the tweets and emails for products of interest are analyzed on Streams to compute the percentage of messages with positive and negative sentiment being expressed. Messages with negative sentiment are further analyzed to determine the cause of the dissatisfaction based on a watch list of known causes. The initial watch list of known causes can be bootstrapped using the results from the analysis of stored messages on BigInsights. As the stream of new negative sentiment is analyzed, Streams checks if the percentage of negative sentiment that have an unknown cause (not in the watch list of known causes), has become significant. If it finds a significant percentage of the causes are unknown, it requests an update from BigInsights. When requested, BigInsights queries all of its data using the same sentiment analytics used in Streams and recalculates the list of known

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000342

causes. This new watch list of causes is used by Streams to update the list of causes to be monitored in real time. The application stores all of the information it gathers but only monitors the information currently of interest in real time, thereby using resources efficiently.

7.1.6 Application interactions

Although this is a simple example, it demonstrates the main interactions between Streams and BigInsights:

- Data ingest into BigInsights from Streams
- Streams triggering deep analysis in BigInsights
- Updating the Streams analytical model from BigInsights.

The implementations of these interactions for this simple demonstration application are discussed in more detail in the following sections.

Data ingest into BigInsights from Streams

Streams processes data using a flow graph of interconnected operators. The data ingest is achieved using a Streams-BigInsights Sink operator to write to BigInsights (refer to 7.1.7, “Enabling components” on page 324). The complexities of the BigInsights distributed file system used to store data are hidden from the Streams developer by the Streams-BigInsights Sink operator. The Sink operator batches the data stream into configurable sized chunks for efficient storage in BigInsights. It also uses buffering techniques to de-couple the write operations from the processing of incoming streams, allowing the application to absorb peak rates and ensure that writes do not block the processing of incoming streams. Like any operator in Streams, the Sink operator writing to BigInsights can be part of a more complex flow graph allowing the load to be split over many concurrent Sink operators that could be distributed over many servers.

Streams triggering deep analysis in BigInsights

Our simple example triggered deeper analysis in BigInsights using the same Streams-BigInsights Sink operator as mentioned in “Data ingest into BigInsights from Streams”. BigInsights does deep analysis using the same sentiment extraction analytic as used in Streams and creates a results file to update the Streams model. For more advanced scenarios, the trigger from Streams could also contain query parameters to tailor the deep analysis in BigInsights.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000343

Updating the Streams analytical model from BigInsights

Streams updates its analytical model from the result of deep analysis in BigInsights. The results of the analysis in BigInsights are processed by Streams as a stream that can be part of a larger flow graph. For our simple example, the results contain a new watch list of causes for which Streams will analyze the negative sentiment.

7.1.7 Enabling components

The integration of data-in-motion and data-at-rest platforms is enabled by three main types of components:

- Common analytics
- Common data formats
- Data exchange adapters

The components used for this simple demonstration application are available on the Streams Exchange at Streams exchange on the IBM developerWorks website (https://www.ibm.com/developerworks/mydeveloperworks/groups/service/html/communityview?communityUuid=d4e7dc8d-0efb-44ff-9a82-897202a3021e) and are discussed in more detail in the following sections.

Common analytics

The same analytics capabilities can be used on both Streams and BigInsights. In this simple example, we use IBM BigInsights System T text analytic capabilities to extract information from the unstructured text received in the emails and tweets. Streams uses a System T operator to extract the product, sentiment, and reasons from the unstructured text in the feeds. The feeds are stored in BigInsights in their raw form and processed using the System T capabilities when the deep analysis is triggered. System T uses AQL, a declarative rule language with a SQL-like syntax to define the information extraction rules. Both Streams and BigInsights use the same AQL query for processing the unstructured text.

Common data formats

Streams formatting operators can transform data between the Streams tuple format and data formats used by BigInsights. In this simple example, we use JSON as the data format for storage in BigInsights. The TupleToJSON operator is used to convert the tuples in Streams to a JSON string for storage in BigInsights. The JSONToTuple operator is used to convert the JSON string read from BigInsights to a tuple for Streams to process.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000344

Common data exchange adapters

Streams source and sink adapters can be used to exchange data with BigInsights. In this simple example, we use HDFSSource, HDFSSink, and HDFSDirectoryScan adapter operators to exchange data with BigInsights. These adapters have similar usage patterns to the fileSource, fileSink, and directoryScan adapter operators provided in the SPL Standard Toolkit. The HFDSSink is used to write data from streams to BigInsights. The HDFSDirectoryScan operator looks for new data to read from BigInsights using the HDFSSource operator.

7.1.8 BigInsights summary

IBM Big Data platforms, InfoSphere Streams and InfoSphere BigInsights, enable businesses to operationalize the integration of data-in-motion and data-at-rest analytics at large scales to gain current and historical insights into their data, allowing faster decision making without restricting the context for those decisions. In this section, we describe various scenarios in which the two platforms interact to address the Big Data analysis problems.

7.2 Streams integration with data mining

In this section, we describe how to write and use an InfoSphere Streams operator to execute an IBM SPSS Modeler predictive model in an InfoSphere Streams application using the IBM SPSS Modeler Solution Publisher Runtime Library API.

7.2.1 Value of integration

IBM SPSS Modeler provides a state of the art environment for understanding data and producing predictive models. InfoSphere Streams provides a scalable high performance environment for real-time analysis of data in motion, including traditional structured or semi-structured data and unstructured data types.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000345

Combining the power of InfoSphere Streams with the sophisticated data modeling capabilities of the IBM SPSS Modeler is valuable. An example of that combination is shown in Figure 7-3.

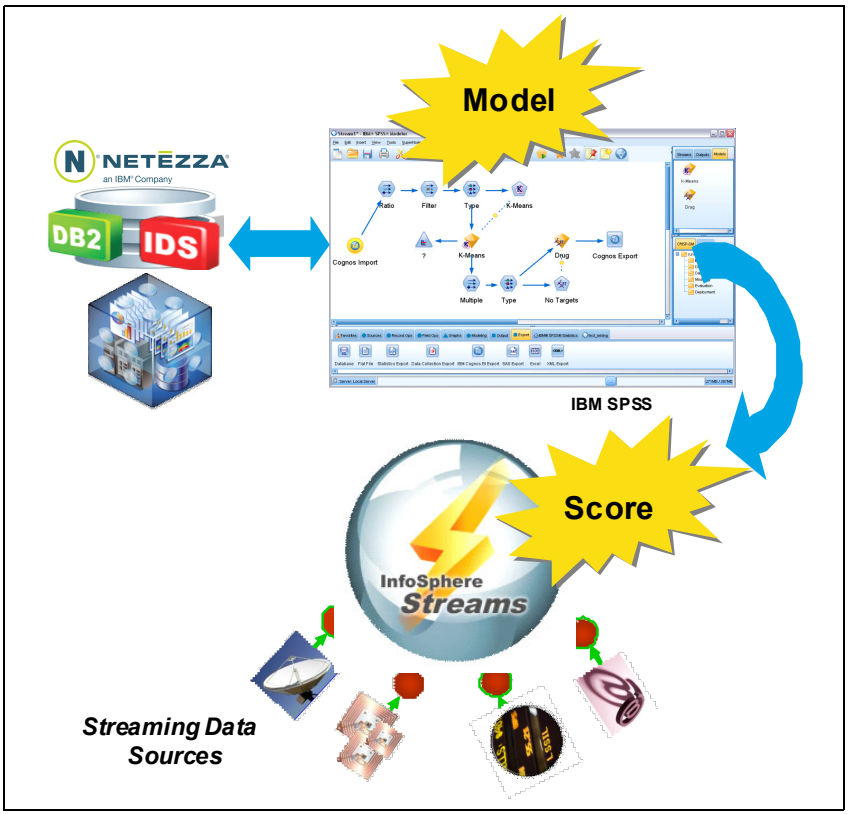


Figure 7-3 Streams and data mining

Roles and terminology

In the following sections, we describe a few roles and their responsibilities and present some terminology that will be used.

Roles

Here are some of the key roles in data mining integration:

- Data Analyst: A modeling expert that knows how to use the IBM SPSS Modeler tools to build and publish predictive models.
- Streams Application Developer: An InfoSphere Streams developer responsible for building applications.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000346

- Streams Component Developer: An InfoSphere Streams developer responsible for developing the components that a Streams Application Developer uses.

The focus of this section is on the Streams Component Developer role and how to write an operator that can execute SPSS predictive models. The other roles are included, as they are needed to understand the work and interaction necessary for the overall solution. Also note that in many cases, the Streams Application Developer and Streams Component Developer may be the same person.

Terminology

Here are some terms commonly used in data mining:

- Predictive model: A prepared scoring branch of an SPSS Modeler Stream sometimes referred to as a model.
- Streams Processing Language (SPL): The language with which InfoSphere Streams applications are written.
- Operator: The basic building block component that InfoSphere Streams applications use. There are a set of operators shipped with the Streams product, and you can write your own custom operators as well.

Note: There is potential for confusion with the use of the term *streams*. The InfoSphere Streams product refers to streams of data, and stream applications built using SPL. The SPSS Modeler product also creates a workflow of connected modeler components known as a stream. For the purpose of this book, the streams of the SPSS modeler will be referred to as *predictive analytics* or *models* and the term *stream* will refer to an InfoSphere Streams data stream.

7.2.2 Sample scenario

Here is an overall flow for integrating model development with real-time scoring:

- A Data Analyst understands the data available and creates the appropriate predictive models and defines what attributes the models require.
- A Streams Component Developer produces the operator that takes those attributes as input and will execute the scoring of the model to produce the desired outputs.
- A Streams Application Developer builds the application that obtains the attributes, calls the operator and takes action based on the resulting scores.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000347

This situation is typically an iterative process with discussions about what attributes are needed from all of those available in the planned stream flow. As such, we work through a sample scenario where a Data Analyst evaluates historical data to produce a useful scoring model, and then passes that model on to the Streams Component Developer, who builds a suitable operator to execute the model in a Streams environment. The Streams Application Developer will produce a Streams application to use the model.

7.2.3 Building the models

Determining what data is available, what models are appropriate, and building those models is typically done by a Data Analyst.

This section builds on the Market Basket Analysis demo and tutorial shipped with the SPSS Modeler Client product. We show here how the model scoring stream built in Modeler can be published and then used in an InfoSphere Streams application to enable real-time scoring and decisions using the SPSS model's logic.

The Market Basket Analysis example deals with fictitious data describing the contents of supermarket baskets (that is, the collections of items bought together) plus the associated personal data of the purchaser, which might be acquired through a loyalty card program. The goal is to discover groups of customers who buy similar products and can be characterized demographically, such as by age, income, and so on.

This example reveals how the IBM SPSS Modeler can be used to discover affinities, or links, in a database, both by modeling and by visualization. These links correspond to groupings of cases in the data, and these groups can be investigated in detail and profiled by modeling. In the retail domain, such customer groupings might, for example, be used to target special offers to improve the response rates to direct mailings or to customize the range of products stocked by a branch to match the demands of its demographic base.

The building and analyzing of models is beyond the scope of this book. Refer to the material provided with the SPSS Modeler Client product for more information about the Data Analyst role and modeling in general.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000348

In this section, we start with a working modeler session, as shown in Figure 7-4.

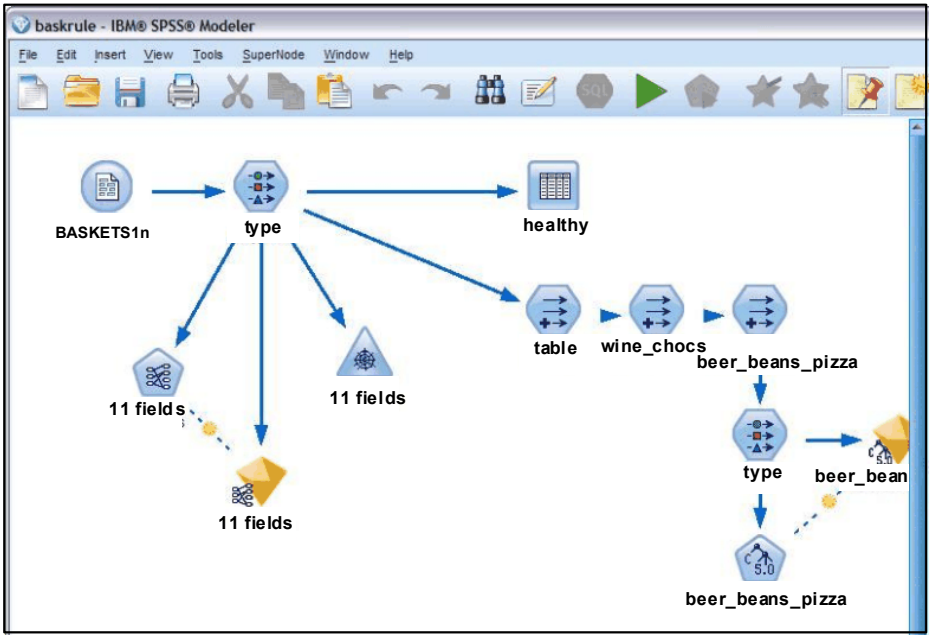


Figure 7-4 Working modeler session

to use the predictive model built in this session, you need to create a scoring branch by connecting an input node to the scoring nugget and then connecting the scoring nugget to an output flat file, as shown in Figure 7-5. Note that this branch in the example is quite trivial and not representative of the typical complexity of real modeling applications. It relies on two input fields, sex and income, to predict whether a customer is likely to purchase a combination of beer, beans, and pizza.

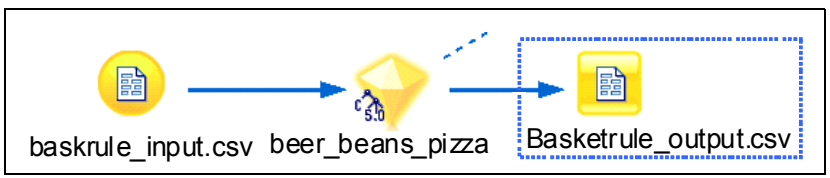


Figure 7-5 Output flat file

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000349

We have also created a sample input file to test the execution of the model, as shown in the preview in Figure 7-6.

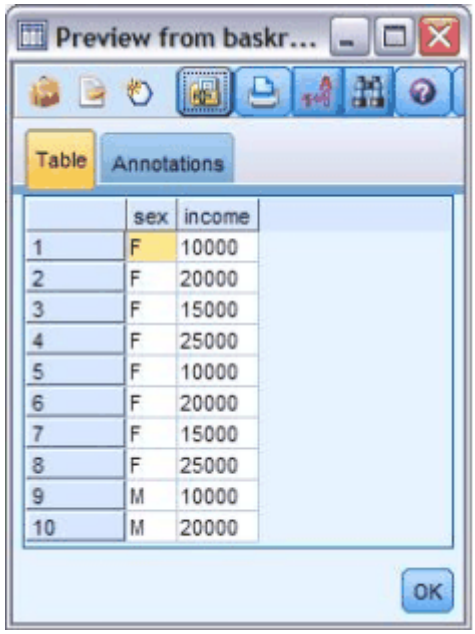
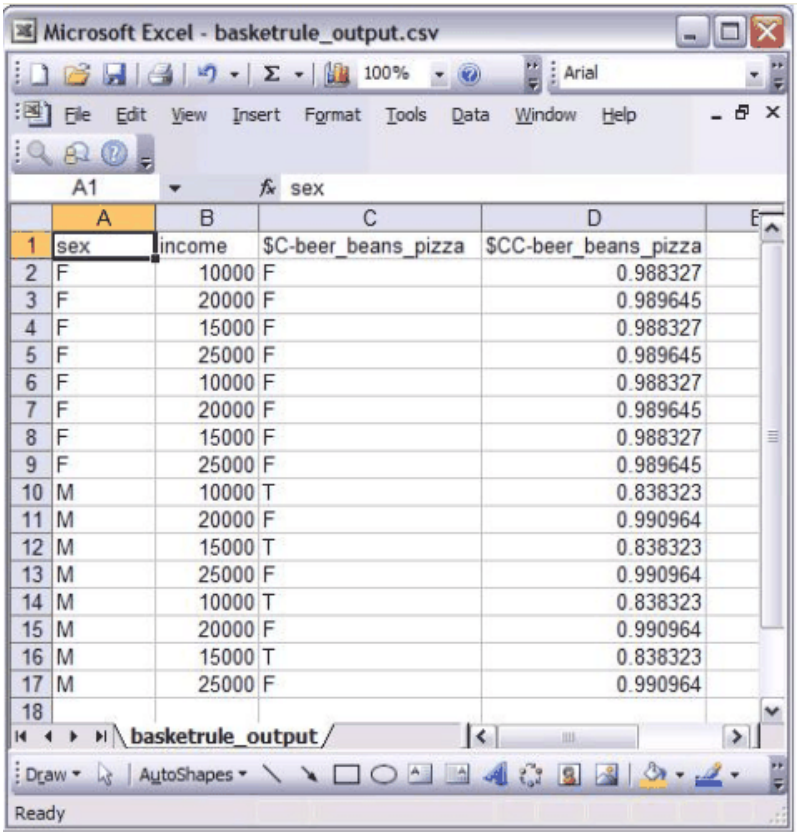


Figure 7-6 Model preview

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000350

Running the branch from within Modeler using the sample set of user inputs, the file shown in Figure 7-7 was produced. Using this same sample data later will help test what is going to happen in the InfoSphere Stream application, where this model will be used within the InfoSphere Streams operator, and input tuple attributes will provide the inputs replacing this data source and the outputs will be added to an output tuple replacing this terminal file.



	A	B	C	D
1	sex	income	$C-beer_beans_pizza	$CC-beer_beans_pizza
2	F	10000	F	0.988327
3	F	20000	F	0.989645
4	F	15000	F	0.988327
5	F	25000	F	0.989645
6	F	10000	F	0.988327
7	F	20000	F	0.989645
8	F	15000	F	0.988327
9	F	25000	F	0.989645
10	M	10000	T	0.838323
11	M	20000	F	0.990964
12	M	15000	T	0.838323
13	M	25000	F	0.990964
14	M	10000	T	0.838323
15	M	20000	F	0.990964
16	M	15000	T	0.838323
17	M	25000	F	0.990964
18				

Figure 7-7 Output file from sample data

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000351

It is probably best to save the modified modeler workbench session after the scoring branch has been created. You use the file node on the scoring branch to produce the artifacts necessary to use the scoring model in the Streams operator. In the Modeler session, change the output file setting to something similar to what is shown in Figure 7-8 and click **Publish**.

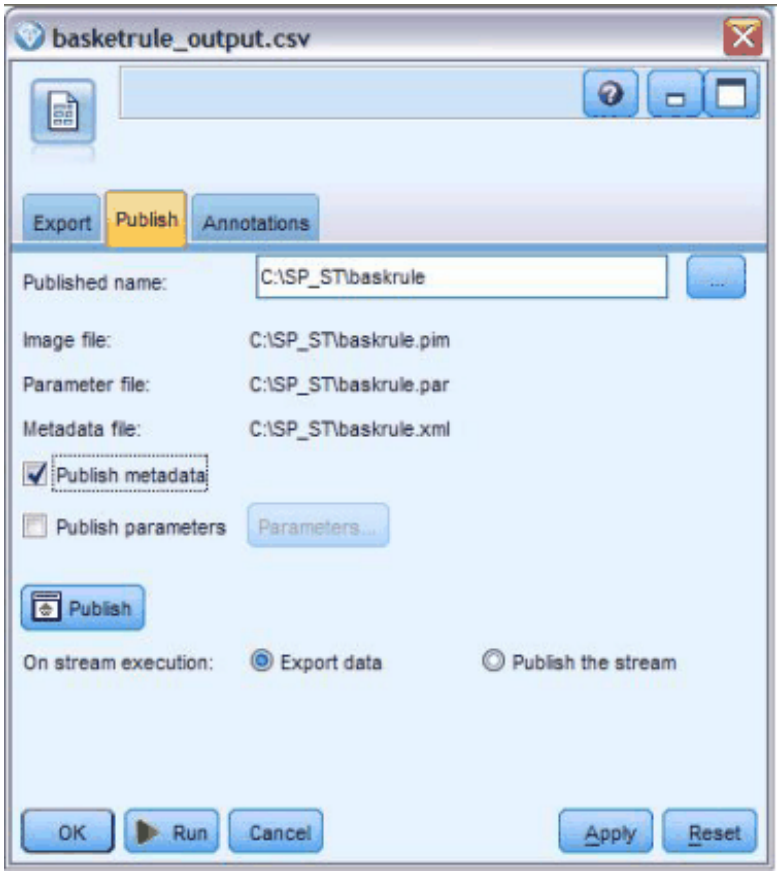


Figure 7-8 Output file setting

Note: Be sure to specify to publish the metadata, which produces an XML document that describes the inputs and outputs of the model, including their field and field types. This is the information necessary to convey to the Streams Component Developers to enable them to build the operator to call this predictive model.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000352

The completed modeler workbench session is provided in the `streamsbasketrule.str file`. For details about accessing and downloading the compressed file and code sample files, refer to “Using the web material” on page 414.

7.2.4 The contract: Data Analyst and Streams Component Developer

In this section, we discuss the contract between the Data Analyst and the Streams Component Developer. To write the Streams operator, the Streams Component Developer needs to know certain information about the inputs and outputs of the predictive model. Specifically, the operator developer requires:

- The install location of the SPSS Solution Publisher component.
- The names and locations of the .pim and .par files produced during the publish.
- The input source node key name can be found in the `baskrule.xml file`, in the following XML fragment:

  ```
  <inputDataSources>
  <inputDataSource name="file0" type="Delimited">
  ```

  For details about accessing and downloading the compressed file and code sample files, see “Using the web material” on page 414.

  Note: Although there is no technical limitation, our example is limited to supporting a single input source for simplicity’s sake.

- The input field names and storage and their order as found inside the <inputDataSource> tag:

  ```
  <fields>
  <field storage="string" type="flag">
  <name>sex</name>
  </field>
  <field storage="integer" type="range">
  <name>income</name>
  </field>
  </fields>
  ```

- The output node or terminal node key name, which can be found in the xml fragment:

  ```
  <outputDataSources>
  <outputDataSource name="file3" type="Delimited">
  ```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000353

Note: Although there is no technical limitation, our example is limited to supporting a single input source for simplicity's sake.

- The output field names and storage and their order, as found inside the <outputDataSource> tag, as shown in the following:

```
<fields>
<field storage="string" type="flag">
<name>sex</name>
</field>
<field storage="integer" type="range">
<name>income</name>
</field>
<field storage="string" type="flag">
<name>$C-beer_beans_pizza</name>
<flag>
<true>
<value>T</value>
</true>
<false>
<value>F</value>
</false>
</flag>
</field>
<field storage="real" type="range">
<name>$CC-beer_beans_pizza</name>
<range>
<min>
<value>0.0</value>
</min>
<max>
<value>1.0</value>
</max>
</range>
</field>
</fields>
```

The Data Analyst has also informed the Streams Component Developer that the model does not modify the input parameters even though they are listed as outputs on the model. While not critical, this bit of information will allow the operator writer to optimize by not recopying those fields to the output tuple.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000354

7.2.5 Preparing to write the operator

Because SPSS Modeler runs in Windows, and Streams runs on a different Linux system, it is a good idea to validate that the Linux environment is properly set up to execute the `.pim` file provided by the Data Analyst. For the sample scenario, because we provided a sample input file and know the expected output, we can use the stand-alone IBM SPSS Modeler Runtime modelerrun script provided as part of Solution Publisher to verify that the Linux environment is set up properly.

Note: The SPSS hotfix to enable scoring through Streams is not compatible with modelerrun. After you apply the SPSS hotfix, the modelerrun program will no longer work. If you want to perform this validation as described below, you must do it before installing the hotfix or install solution publisher to two different locations and only apply the hotfix to the installation used with Streams.

To run the modelerrun script on the sample data and model, extract the sample file `baskrule.zip` (For details about accessing and downloading the compressed file and code sample files, refer to “Using the web material” on page 414) to a location on a Linux system where you have Solution Publisher installed. Then change the directory to the data directory under the `baskrule` directory, and then execute the modelerrun script from the Solution Publisher install location as shown below:

```
bash-3.2$
bash-3.2 cd /homes/hny1/koranda/baskruletest1/baskrule/data
bash-3.2$
~koranda/IBM/SPSS/ModelerSolutionPublisher64P/14.2/modelerrun -p
baskrule.par baskrule.pim
IBM SPSS Modeler Runtime 14.2
(C) Copyright IBM Corp. 1994, 2011
bash-3.2$
```

View the results, as shown in the following:

```
bash-3.2$ cat baskrule.csv
sex,income,$C-beer_beans_pizza,$CC-beer_beans_pizza
F,10000,F,0.988327
F,20000,F,0.989645
F,15000,F,0.988327
F,25000,F,0.989645
F,10000,F,0.988327
F,20000,F,0.989645
F,15000,F,0.988327
F,25000,F,0.989645
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000355

```
M,10000,T,0.838323
M,20000,F,0.990964
M,15000,T,0.838323
M,25000,F,0.990964
M,10000,T,0.838323
M,20000,F,0.990964
M,15000,T,0.838323
M,25000,F,0.990964
bash-3.2$
```

7.2.6 The Streams Component Developer

The Streams Component Developer writes the Streams Scoring operator and the operator's characteristics. The developer takes the information provided by the Data Analyst and builds an operator. In the sample scenario, this operator needs to accept a single tuple that contains two attributes:

- Sex (a string value of M or F)
- Income (an integer value)

The operator produces a single output tuple, passing through all of input tuple attributes and adding the following attributes:

- Prediction (a string value of T or F)
- Confidence (a float value)

We accomplish this task by writing a non-generic operator (an operator that only works on this specific tuple / stream format) that uses the Solution Publisher API interface to execute the .pim file. A link to more information about the Solution Publisher API is provided in , "Online resources" on page 421.

The general outline of an operator using the Solution Publisher API is as follows:

1. In the operator's initialization code:
   a. Initialize the solution publisher clemrtl API library.
   b. Open the model image file (.pim file) and receive an image handle.
   c. Get the necessary information for the input and output field types from the image required for the alternative input and output calls.
   d. Define a function that will be used to map from the input tuple attributes to the input fields required in the model and register it in the image.
   e. Define a function that will be used to map from the output fields produced by the model to the output tuple attributes and register it in the image.
   f. Because the same image is to be executed multiple times without changing parameters, prepare the image once during initialization.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000356

2. In the operator's process method (called for each input tuple):
   a. Populate the data structure registered for the alternate input from the input tuple attributes.
   b. Execute the image.
   c. Populate the output tuple attributes from the data structure registered for the alternative output.
3. In the operator's prepareToShutdown method, close the image.

We now take you through each of these areas, highlighting the specific code required in the Streams operator artifacts. The complete listing for all these code samples (the `baskrule/MyOp/MyOp.xml`, `baskrule/MyOp/MyOp_h.cgt`, and `baskrule/MyOp/MyOp_cpp.cgt` files) is included in the `baskrule.zip` file. For details about accessing and downloading the compressed file and code sample files, refer to “Using the web material” on page 414.

Operator model

In the operator's XML model definition, the dependencies section of the Solution Publisher code location must be specified:

```
<libraryDependencies>
        <library>
          <cmn:description>spss-lib</cmn:description>
          <cmn:managedLibrary>
<cmn:libPath>/opt/IBM/SPSS/ModelerSolutionPublisher/14.2</cmn:libPath>

<cmn:includePath>/opt/IBM/SPSS/ModelerSolutionPublisher/14.2/clemrtl/in
clude</cmn:includePath>
          </cmn:managedLibrary>
        </library>
      </libraryDependencies>
```

The complete operator model can be found in `baskrule/MyOp/MyOp.xml`.

Operator initialization code

Prior to calling any of the Solution Publisher APIs, the operator needs to dynamically load the Solution Publisher APIs and initialize their entry points. For the operator, this requires the following types of definitions in the `_h.cgt file` (refer to `baskrule/MyOp/MyOp_h.cgt` for the complete list of entry points defined):

```
typedef int (*clemrtl_initialise_ext_t)(unsigned, int, void*);
…
void * libclemertl;
  clemrtl_initialise_ext_t clemrtl_initialise_ext;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000357

```
Then in the operator's constructor in the _cpp.cgt file the libraries
are loaded using:

libclemertl = dlopen("libclemrtl.so", RTLD_NOW | RTLD_DEEPBIND);
    if(!libclemertl)
        throw SPLRuntimeOperatorException(getThisLine(), dlerror());
/* get the routine addresses */
clemrtl_initialise_ext = (clemrtl_initialise_ext_t)
dlsym(libclemertl,"clemrtl_initialise_ext");
    if(!clemrtl_initialise_ext)
         throw SPLRuntimeOperatorException(getThisLine(), dlerror());
```

This loading code would be the same for any operator that calls Solution Publisher and would not need to change to use different models or data.

Initializing the solution publisher clemrtl API library

The first necessary Solution Publisher API is to initialize the library using clemrtl_initialise_ext(). The initialize call requires a parameter of the Solution Publisher installation directory. For this situation, we have created a parameter on the operator that can be specified in the `.spl` file. The default is the standard Solution Publisher install location. This location can be seen in the `baskrule/MyOp/MyOp_cpp.cgt` file, shown in the following example:

```
rstring installationDirectory =
"/opt/IBM/SPSS/ModelerSolutionPublisher/14.2";
    if(hasParameter("SP_Install"))
      installationDirectory = getParameter("SP_Install");
    SPLLOG(L_INFO, "About to clemrtl initialize using SP_Install of:
"<< installationDirectory, "MPK");
    clemrtl_init_arg args[] = {
      {"installation_directory", installationDirectory.c_str()},
    };
    const int arg_count = sizeof args / sizeof args[0];
    if (clemrtl_initialise_ext(0,arg_count,args) != CLEMRTL_OK) {
        SPLLOG(L_ERROR, "Clemrtl initialise failed", "MPK");
    }
```

Opening an image

Open an image using clemrtl_openImage() and you receive an image handle. This API requires the location of the `.pim` and `.par` files provided by the Data Analyst. Note that we have hardcoded the `.pim` and `.par` file names and expect them to be found in the data directory. A more general solution would be to add these as parameters passed in to the operator similar to the way the Solution Publisher install location was specified, as shown in the following:

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000358

```
/* open the image */
     int res, status = EXIT_FAILURE;
     image_handle = 0;

     res = clemrtl_openImage("baskrule.pim","baskrule.par",
&image_handle);
     if (res != CLEMRTL_OK) {
      status = EXIT_FAILURE;
      SPLLOG(L_ERROR, "Open Image Failed", "MPK");
      displayError(res, 0);
   }
```

The displayError routine was defined to obtain additional detailed information from the specific error, print it out, and then end the operator by throwing a runtime exception.

Getting the input and output field information

Use the clemrtl_getFieldCount() and getFieldTypes() to obtain the information about the input and output fields and types. Note that the key and keyOut fields must contain the values provided by the Data Analyst from the .xml metadata. In the sample, the <inputDataSource name="file0" type="Delimited"> and <outputDataSource name="file3" type="Delimited">inputDataSources> names must be used, as shown in Example 7-1.

Example 7-1 Sample input and output field information

```
/* Get Input field count and types */
    char* key="file0";
  res = clemrtl_getFieldCount(image_handle, key, &fld_cnt );
  if (res != CLEMRTL_OK) {
   status = EXIT_FAILURE;
   SPLLOG(L_ERROR, "Get Field Count Failed", "MPK");
   displayError(res, image_handle);
  }
  SPLLOG(L_INFO, "Field Count is: "<<(int)fld_cnt, "MPK");

  int fld_types[100]; // needs to be bigger if more than 100 fields
expected
  res = clemrtl_getFieldTypes(image_handle, key, fld_cnt, fld_types );
  if (res != CLEMRTL_OK) {
   status = EXIT_FAILURE;
   SPLLOG(L_ERROR, "Get Field Types Failed", "MPK");
   displayError(res, image_handle);
  }
  field_proc(fld_cnt,fld_types);
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000359

```
  /* Get Output field count and types */
  size_t fld_cnt_out;
  char* keyOut="file3";
  res = clemrtl_getFieldCount(image_handle, keyOut, &fld_cnt_out );
  if (res != CLEMRTL_OK) {
   status = EXIT_FAILURE;
   SPLLOG(L_ERROR, "Get Output Field Type Failed", "MPK");
   displayError(res, image_handle);
  }
  SPLLOG(L_INFO, "Output Field Count is: "<<(int)fld_cnt_out, "MPK");

  int fld_types_out[100]; // needs to be bigger if more than 100 fields
expected
  res = clemrtl_getFieldTypes(image_handle, keyOut, fld_cnt_out,
fld_types_out );
  if (res != CLEMRTL_OK) {
   status = EXIT_FAILURE;
   SPLLOG(L_ERROR, "Get Output Field Types Failed", "MPK");
   displayError(res, image_handle);
  }
  field_proc(fld_cnt_out,fld_types_out);
```

Defining the input function

The Solution Publisher API provides the ability to define an input iterator function that will be called during execute to provide the data used in the model. The function will be used to map from the input tuple attributes to the input fields required in the model. We register it in the image using clemrtl_setAlternativeInput(). When you register this function, it requires that you pass the address of the input field structure that contains pointers to the input data, so the function is defined as the following:

```
void** MY_OPERATOR::next_record(void* arg) {
     SPLLOG(L_INFO, "In next_record iterator", "MPK");
          return *((buffer*) arg)->next_row++;
   }
```

The structure and pointer for the input data is defined in the `header _h.cgt file` as follows:

```
typedef struct {
     void** row[2];
     void*** next_row;
   } buffer;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000360

```
…
buffer myBuf;
```

In this case, we hardcoded the row array size of 2 to allow for processing a single input tuple and then the second row to contain a NULL to indicate no more rows.

The memory for the input array of void* pointers that point to the input fields is allocated as an array of void* pointers with a size of the number of input fields (as determined from the xml metadata file) and should match the number of input fields returned from the earlier call to clemrtl_getFieldCount(image_handle, key, &fld_cnt). This setup is as follows:

```
/* initialize the buffer */
    void* inPointers[2];
    myBuf.row[0] = (void**) &inPointers;
    myBuf.row[1] = NULL;
    myBuf.next_row = myBuf.row;
```

The setAlternativeInput call then passes the key corresponding to the model input section, the input field count and types retrieved earlier, the function address, and the address of the input data structure, as shown in the following:

```
res = clemrtl_setAlternativeInput(image_handle, key, fld_cnt,
fld_types, MY_OPERATOR::next_record, (void*) &myBuf );
if (res != CLEMRTL_OK) {
status = EXIT_FAILURE;
SPLLOG(L_ERROR, "Set Alternative Input Failed", "MPK");
displayError(res, image_handle);
}
```

Defining the output function

The Solution Publisher API provides the ability to define an output iterator function that will be called during execute to capture the data produced in the model. Our function will be used to map from the output fields produced by the model to the output tuple attributes. This function is registered in the image using clemrtl_setAlternativeOutput(). When you register this function, it allows you to pass an object that will be opaquely passed into the function when called during execute. In this case, we pass a structure that maintains a linked list of output buffers that will be created during each iteration of the function. Later, this linked list will be used to move the output field data into an output tuple and sent on the output port. The function is defined as follows:

```
void MY_OPERATOR::next_record_back(void* arg, void** row) {
     outField* ofp;
     ofp = (outField*) arg;

     outBuffer* obp;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000361

```
    obp = (outBuffer*) new outBuffer;
    obp->next=0;
    if (ofp->head == 0) { // first call
      ofp->head = obp;
      ofp->tail = obp;
    } else {
      (ofp->tail)->next = obp;
      ofp->tail = obp;
    }

      if (row[0]) {
     obp->sex = (const char *) row[0];
                    obp->_missing_sex = false;
                   } else {obp->_missing_sex = true;}
                  if (row[1]) {
                    obp->income = *((long long *) row[1]);
                    obp->_missing_income = false;
                  } else {obp->_missing_income = true;}
                  if (row[2]) {
                    obp->prediction = (const char *) row[2];
                    obp->_missing_prediction = false;
                  } else {obp->_missing_prediction = true;}
                  if (row[3]) {
                    obp->confidence = *((double *) row[3]);
                    obp->_missing_confidence = false;
                  } else {obp->_missing_confidence = true;}
}
```

The structure for the output data from the model is defined in the header file as follows:

```
typedef struct {
  void* next;
  const char* sex;
  boolean _missing_sex;
  long long income;
  boolean _missing_income;
  const char* prediction;
  boolean _missing_prediction;
  double confidence;
  boolean _missing_confidence;
} outBuffer;
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000362

When we define this structure, this is where the information provided by the Data Analyst related to the output fields in the XML metadata comes into play. The storage value is used to determine the type of data in the structure returned and the code necessary move it from the output row returned to the structure's memory. Note also that the order of the fields in the structure and returned row are described by the order in the xml metadata provided by the Data Analyst.

In our output iteration function, we test to make sure a value was produced by the model. An address of returned for a field indicates a missing value. We capture the fact that a value is missing in the corresponding _missing_xxx field defined in our structure. Later, this value will be used when populating the output tuple.

The clemrtl_setAlternativeOuput call passes the key corresponding to the model output section, the output field count and types retrieved earlier, the function address, and the pointer to the structure that maintains the linked list to manage the data produced in the model, as shown in the following:

```
/* Set the alternative output */
res = clemrtl_setAlternativeOutput(image_handle, keyOut, fld_cnt_out,
fld_types_out, MY_OPERATOR::next_record_back, (void*) &myOutField );
if (res != CLEMRTL_OK) {
status = EXIT_FAILURE;
SPLLOG(L_ERROR, "Set Alternative Output failed", "MPK");
displayError(res, image_handle);
}
```

Preparing the image

Because the same image will be executed multiple times without changing parameters, we use clemrtl_prepare(), which results in less processing during the execution for each tuple, as shown in the following:

```
/* prepare the image */
    res = clemrtl_prepare(image_handle);
if (res != CLEMRTL_OK) {
status = EXIT_FAILURE;
SPLLOG(L_ERROR, "Prepare Failed", "MPK");
displayError(res, image_handle);
}
```

Operator process method

In this section, we describe the operator process method.

Populating the input fields

As each input tuple arrives, we deal with it in the operator's process method. First, we populate the data structure defined in clemrtl_setAlternateInput () from

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000363

the input tuple attributes. Note that this is where the information provided by the Data Analyst in the XML metadata related to the input fields comes into play. The storage value is used to determine the type of data in the structure to be populated and the attributes on the stream need to match the storage required by the model. In the sample, the tuple attribute s_sex is defined as an rstring. After it is retrieved from the input tuple, the .c_str() method is used to get the underlying c reference, which matches the storage expected by the first input field sex, as described in <field storage="string" type="discrete"><name>sex</name>. Likewise, the tuple attribute s_income is defined as an int64. After it is retrieved, the reference of the underlying storage is returned using the attributes getter, which matches the storage expected by the second input field income, as described in <field storage="integer" type="range"><name>income</name>. Note also that the order of the fields to be populated in the row is described by the order in the xml metadata provided by the Data Analyst.

```
myBuf.row[0] [0] = (void*) (tuple.get_s_sex().c_str());
  myBuf.row[0] [1] = (void*) &(tuple.get_s_income());

  myBuf.next_row = myBuf.row; //set to point to the new data
```

Executing the image

Execute the image using clemrtl_execute(). Note that we use the same displayError routine (which will throw an exception and end the operator), so if an error in execution should occur, the operator is terminated and no more scoring will occur. It may be more appropriate to log the failure for a single tuple and ignore it, or perhaps indicate an error in a different way by writing it to an additional output port used to handle errors or indicate in an additional attribute that the model execution failed and the results are incorrect, but for our simple example we end the operator.

```
/* execute the image */
    res = clemrtl_execute(image_handle);
if (res != CLEMRTL_OK) {
status = EXIT_FAILURE;
SPLLOG(L_ERROR, "Execute Failed", "MPK");
displayError(res, image_handle);
}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000364

Populating the output tuple

After successful execution of the model, myOutField contains the head of the linked list of output data returned. This list is traversed, and for each element, the output tuple attributes from the data are copied during the output iterator function. The tuple is submitted on the output port. You can see that if no output was produced by the model (the linked list is empty), no output tuple will be sent. Again, the types defined for these attributes must match the types in the outBuffer structure, as derived from the xml metadata. In the sample, we define the attributes as rstring predLabel, float64 confidence.

The indication of a missing value will result in the tuple's output attribute being left at its default value after the clear (a 0 for numerics and "" for string fields). Note that these defaults may actually be part of the domain of values the model could return, so different values or different processing might be necessary depending on your model.

```
outBuffer* currentOutBuf = (outBuffer*)(myOutField.head);
while (currentOutBuf) {
  otuple.clear(); //reset all attributes to their default values.
  otuple.assignFrom(tp, false); // move input fields to output tuple

   /* Dig out what was returned and add to output tuple */
  if (currentOutBuf->_missing_prediction == false)
    otuple.set_predLabel(currentOutBuf->prediction);
  if (currentOutBuf->_missing_confidence == false)
    otuple.set_confidence(currentOutBuf->confidence);

   submit(otuple, 0);


   /* move to next output buffer in list */
  outBuffer* nextPtr = (outBuffer*)(currentOutBuf->next);
  delete currentOutBuf; // free the memory allocated in the output iterator
  currentOutBuf = nextPtr;
} // end of while loop
```

Operator prepareToShutdown method

In this section, we prepare for a shutdown.

Closing the image

When the operator is shut down, we close the image using clemrtl_closeImage(), allowing a graceful shutdown:

```
/* close the image */
  res = clemrtl_closeImage(image_handle);
  if (res != CLEMRTL_OK) {
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000365

```
    status = EXIT_FAILURE;
    SPLLOG(L_ERROR, "Close Image Failed", "MPK");
    displayError(res, image_handle);
  }
```

7.2.7 The Streams Application Developer

In this section, we describe some of the activities of the Streams Application Developer.

Using the Scoring Model operator

Integrating a scoring model into a Streams application is done typically by a Streams Application Developer. In a real Streams application, the input data might come from one or more continuously streaming sources of data. For this book, we simulate that flow by using a file containing rows to be scored and use the InfoSphere Streams FileSource operator to read in the information and produce a stream of tuples.

In a real streaming application, the output scores would be processed by further downstream application segments, written to external systems, or saved in historical data stores. Here we write the scored tuples to an output file using the InfoSphere Streams FileSink operator.

The inputs and outputs expected by the operator were built into the operator and would need to be communicated by the Component Developer to the Application Developer. In our case, they would need to communicate that this operator expects two input attributes, s_sex (an rstring value of T or F) and s_income (an int64 value), and produces two output attributes of prediction, an rstring T or F for whether this input indicates a preference of purchasing a combination of beer, beans, and pizza, and confidence, a float64 representing the confidence of that prediction.

Running the sample SPL application

In this section, we describe the requirements and setup:

1. To build and run the sample application, you need a functioning InfoSphere Streams V2.0 or greater environment.
2. You need the Solution Publisher Runtime and a hotfix installed, as described at the following address:

   http://www.ibm.com/support/docview.wss?uid=swg24030862
3. You need to ensure that LD_LIBRARY_PATH is set to contain the necessary Solution Publisher libraries on all systems on which the Streams operator will be deployed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000366

LD_LIBRARY_PATH requirement

Assuming the Solution Publisher is installed in `$INSTALL_PATH`, then the LD_LIBRARY_PATH needs to include the following paths:

- `$INSTALL_PATH`
- `$INSTALL_PATH/ext/bin/*`
- `$INSTALL_PATH/jre/bin/classic`
- `$INSTALL_PATH/jre/bin`

A script included in the compressed file named ldlibrarypath.sh is provided to set the path up correctly. If Solution Publisher is not installed in the default location, you need to change the first line of the script to point to your Solution Publisher install directory before using the script. For example, if Solution Publisher is installed in `/homes/hny1/koranda/IBM/SPSS/ModelerSolutionPublisher64P/14.2`, then set the first script line as follows:

```
CLEMRUNTIME=/homes/hny1/koranda/IBM/SPSS/ModelerSolutionPublisher64P/14
.2
```

Sample contents

The sample compressed file contains the `.pim`, `.par`, and `.xml` files from the Market Basket Analysis sample with a scoring branch added, a sample input and expected output files, and a complete Streams Programming Language application, including the custom operator that scores the basket analysis model.

We provide a simple SPL application in `baskrule/MpkSimpleSpss.spl`, which is shown in Figure 7-9.

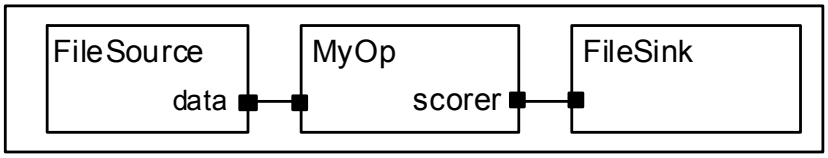


Figure 7-9 Sample SPL application

Adjusting and compiling the sample

To run the sample SPL application, extract the `baskrule.zip` file to your Linux system that has InfoSphere Streams and Solution Publisher installed.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000367

If the Solution Publisher Install location is different from the default value of `/opt/IBM/SPSS/ModelerSolutionPublisher/14.2`, you need to modify the operator xml file (`MyOp.xml`) in the MyOp directory. You need to change the libPath and includePath entries to match your Solution Publisher install location:

```
<cmn:libPath>/opt/IBM/SPSS/ModelerSolutionPublisher/14.2</cmn:libPath>
<cmn:includePath>/opt/IBM/SPSS/ModelerSolutionPublisher/14.2/clemrtl/in
clude</cmn:includePath>
```

You need to modify the `MpkSimpleSpss.spl` file by adding the SP_Install parameter on the operator's invocation, as shown in the following example:

```
stream<DataSchemaPlus> scorer = MyOp(data){
       param SP_Install:
"/homes/hny1/koranda/IBM/SPSS/ModelerSolutionPublisher64P/14.2";
  }
```

Compiling the sample

To compile the sample as a stand-alone Streams application, change the directory to where you extracted the sample project (baskrule) and run **`make`**:

```
bash-3.2$ cd STSPTest/baskrule/
bash-3.2$ make
/homes/hny1/koranda/InfoSphereStreams64/bin/sc -a -T -M Main
Creating types...
Creating functions...
Creating operators...
Creating PEs...
Creating standalone app...
Creating application model...
Building binaries...
 [CXX-type] tuple<rstring s_sex,int64 s_income>
 [CXX-operator] data
 [CXX-operator] scorer
 [CXX-type] tuple<rstring s_sex,int64 s_income,rstring
predLabel,float64 confidence>
 [CXX-operator] Writer
 [CXX-pe] pe0
 [CXX-standalone] standalone
 [LD-standalone] standalone
 [LN-standalone] standalone
 [LD-pe] pe0
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000368

Executing the sample

To execute the application, be sure you have set the LD_LIBRARY_PATH as follows:

```
bash-3.2$ source ldlibrarypath.sh
```

Next, change the directory to the data directory under the baskrule directory and then execute the stand-alone program:

```
bash-3.2$ cd data/
bash-3.2$ ../output/bin/standalone
```

Because the SPL program specified a verbosity of INFO, you should see a number of informational messages showing the processing:

```
09 Aug 2011 17:11:07.181 [21715] INFO spl_pe M[PEImpl.cpp:process:483]
- Start processing...
09 Aug 2011 17:11:07.379 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:167]  - About to clemrtl initialise using
SP_Install of:
"/homes/hny1/koranda/IBM/SPSS/ModelerSolutionPublisher64P/14.2"
09 Aug 2011 17:11:07.458 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:175]  - After clemrtl initialise
09 Aug 2011 17:11:07.459 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:180]  - Major=14 Minor=2 Release=0 build=0
09 Aug 2011 17:11:07.461 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:195]  - Image Handle Retrieved: 1
09 Aug 2011 17:11:07.461 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:203]  - About to get field count
09 Aug 2011 17:11:07.462 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:210]  - Field Count is: 2
09 Aug 2011 17:11:07.462 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:213]  - About to get Field Types
09 Aug 2011 17:11:07.463 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:127]  - Field Type 0 is: STRING
09 Aug 2011 17:11:07.463 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:133]  - Field Type 1 is: LONG
09 Aug 2011 17:11:07.464 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:229]  - About to get Output Field Count
09 Aug 2011 17:11:07.464 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:236]  - Output Field Count is: 4
09 Aug 2011 17:11:07.464 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:239]  - About to get output field types
09 Aug 2011 17:11:07.465 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:127]  - Field Type 0 is: STRING
09 Aug 2011 17:11:07.465 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:133]  - Field Type 1 is: LONG
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000369

```
09 Aug 2011 17:11:07.466 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:127]  - Field Type 2 is: STRING
09 Aug 2011 17:11:07.466 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:field_proc:136]  - Field Type 3 is: DOUBLE
09 Aug 2011 17:11:07.466 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:258]  - About to set alternative Input
09 Aug 2011 17:11:07.467 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:265]  - After Set Alternative input
09 Aug 2011 17:11:07.467 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:268]  - About to set alternative output
09 Aug 2011 17:11:07.467 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:275]  - After Set Alternative Output
09 Aug 2011 17:11:07.467 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:278]  - About to prepare
09 Aug 2011 17:11:07.471 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:285]  - After Prepare
09 Aug 2011 17:11:07.471 [21715] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:scorer:288]  - Leaving Constructor
09 Aug 2011 17:11:07.474 [21715] INFO spl_pe
M[PECleaners.cpp:initializeImpl:88]  - Opening ports...
09 Aug 2011 17:11:07.474 [21715] INFO spl_pe
M[PECleaners.cpp:initializeImpl:90]  - Opened all ports...
09 Aug 2011 17:11:07.475 [21715] INFO spl_operator
M[OperatorImpl.cpp:createThreads:257]  - Creating 1 operator threads
09 Aug 2011 17:11:07.475 [21715] INFO spl_operator
M[OperatorImpl.cpp:createThreads:263]  - Created 1 operator threads
09 Aug 2011 17:11:07.475 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:319]  - Processing tuple from input port 0
{s_sex="F",s_income=10000}
09 Aug 2011 17:11:07.476 [21715] INFO spl_pe
M[PEImpl.cpp:joinOperatorThreads:649]  - Joining operator threads...
09 Aug 2011 17:11:07.476 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:338]  - About to execute the image
09 Aug 2011 17:11:07.476 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:next_record:33]  - In next_record iterator
09 Aug 2011 17:11:07.477 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:next_record:33]  - In next_record iterator
09 Aug 2011 17:11:07.477 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:345]  - After Execute
09 Aug 2011 17:11:07.477 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:371]  - Sending tuple to output port 0
{s_sex="F",s_income=10000,predLabel="F",confidence=0.988327}
09 Aug 2011 17:11:07.477 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:319]  - Processing tuple from input port 0
{s_sex="F",s_income=20000}
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000370

```
09 Aug 2011 17:11:07.478 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:338]  - About to execute the image
09 Aug 2011 17:11:07.478 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:next_record:33]  - In next_record iterator
09 Aug 2011 17:11:07.478 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:next_record:33]  - In next_record iterator
09 Aug 2011 17:11:07.478 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:345]  - After Execute
09 Aug 2011 17:11:07.478 [21717] INFO #spllog,scorer,MPK
M[MyOp_cpp.cgt:process:371]  - Sending tuple to output port 0
{s_sex="F",s_income=20000,predLabel="F",confidence=0.989645}
…

…
09 Aug 2011 17:11:07.498 [21715] INFO spl_pe
M[PEImpl.cpp:joinOperatorThreads:653]  - Joined all operator threads...
09 Aug 2011 17:11:07.498 [21715] INFO spl_pe
M[PEImpl.cpp:joinWindowThreads:667]  - Joining window threads...
09 Aug 2011 17:11:07.498 [21715] INFO spl_pe
M[PEImpl.cpp:joinWindowThreads:671]  - Joined all window threads.
09 Aug 2011 17:11:07.499 [21715] INFO spl_pe
M[PEImpl.cpp:joinActiveQueues:658]  - Joining active queues...
09 Aug 2011 17:11:07.499 [21715] INFO spl_pe
M[PEImpl.cpp:joinActiveQueues:662]  - Joined active queues.
09 Aug 2011 17:11:07.500 [21715] INFO spl_pe
M[PECleaners.cpp:finalizeImpl:99]  - Closing ports...
09 Aug 2011 17:11:07.500 [21715] INFO spl_pe
M[PECleaners.cpp:finalizeImpl:101]  - Closed all ports...
09 Aug 2011 17:11:07.500 [21715] INFO spl_pe
M[PEImpl.cpp:prepareOperatorsForTermination:640]  - Notifying operators
of termination...
09 Aug 2011 17:11:07.501 [21715] INFO #spllog,Writer,MPK
M[MyOp_cpp.cgt:prepareToShutdown:302]  - About to Close Image with
handle1
09 Aug 2011 17:11:07.502 [21715] INFO #spllog,Writer,MPK
M[MyOp_cpp.cgt:prepareToShutdown:311]  - After Close Image
09 Aug 2011 17:11:07.505 [21715] INFO spl_pe
M[PEImpl.cpp:prepareOperatorsForTermination:644]  - Notified all
operators of termination...
09 Aug 2011 17:11:07.505 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:676]  - Flushing operator profile
metrics...
09 Aug 2011 17:11:07.505 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:680]  - Flushed all operator profile
metrics...
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000371

```
09 Aug 2011 17:11:07.505 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:682]  - Deleting active queues...
09 Aug 2011 17:11:07.506 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:686]  - Deleted active queues.
09 Aug 2011 17:11:07.506 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:688]  - Deleting input port tuple cache...
09 Aug 2011 17:11:07.506 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:692]  - Deleted input port tuple cache.
09 Aug 2011 17:11:07.506 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:697]  - Deleting all operators...
09 Aug 2011 17:11:07.507 [21715] INFO spl_pe
M[PEImpl.cpp:deleteOperators:701]  - Deleted all operators.
09 Aug 2011 17:11:07.507 [21715] INFO spl_pe M[PEImpl.cpp:process:534]
- Terminating...
09 Aug 2011 17:11:07.507 [21715] INFO spl_app
M[StandaloneApplicationImpl.cpp:run:201]  - Completed the standalone
application processing
09 Aug 2011 17:11:07.507 [21715] INFO spl_app
M[StandaloneApplicationImpl.cpp:run:203]  - Leaving
MyApplication::run()
09 Aug 2011 17:11:07.508 [21716] INFO spl_pe M[PEImpl.cpp:shutdown:740]
- Shutdown request received by PE...
09 Aug 2011 17:11:07.508 [21716] INFO spl_pe M[PEImpl.cpp:shutdown:744]
- shutdownRequested set to true...
```

To see the results, look in the `mpkoutput.csv` file produced by the FileSink operator:

```
bash-3.2$ cat mpkoutput.csv
"F",10000,"F",0.988326848249027
"F",20000,"F",0.989645351835357
"F",15000,"F",0.988326848249027
"F",25000,"F",0.989645351835357
"F",10000,"F",0.988326848249027
"F",20000,"F",0.989645351835357
"F",15000,"F",0.988326848249027
"F",25000,"F",0.989645351835357
"M",10000,"T",0.838323353293413
"M",20000,"F",0.990963855421687
"M",15000,"T",0.838323353293413
"M",25000,"F",0.990963855421687
"M",10000,"T",0.838323353293413
"M",20000,"F",0.990963855421687
"M",15000,"T",0.838323353293413
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000372

```
"M",25000,"F",0.990963855421687
bash-3.2$
```

You have written an operator to invoke SPSS models from a Streams application. The next logical step would be for you to modify the sample adapter to work for your own specific model and data. In the following sections is the basic information of what would need to be modified.

Adjusting for a different model

To adjust this sample operator to work for a different input tuple/model combination you need to modify it in the following places:

- `_h.cgt file`: Adjust the output structure.
- `_cpp.cgt` file.
  - `next_record_back`
    - Adjust the output structure.
  - `constructor`
    - Adjust the `.pim` and `.par` file names and locations.
    - Adjust the input and output file tags.
    - Adjust the input field pointer array size.
  - `process`
    - Adjust the load input structure code.
    - Adjust the load output tuple code.

The sample code provided has annotated those sections that might need to be adjusted:

```
/************************************************************
* the following needs to be adjusted to match the pim and par *
* file locations.  Unqualified gets from data directory.      *
************************************************************/
res = clemrtl_openImage("baskrule.pim","baskrule.par", &image_handle);
```

Type matching

An important aspect to support successful integration is ensuring the tuple attributes are compatible with the storage required by the model fields.

In defining models in SPSS Modeler, the Solution Publisher documentation describes the mapping from Modeler types to their typical C Declarator. For example, the modeler type STRING is interpreted as a UTF-8 null-terminated character string and declared as a const char*, while the modeler type LONG is interpreted as a 64-bit signed integer and declared as a long long.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000373

The InfoSphere Streams documentation provides a table that describes how to map from SPL types to their C++ equivalent. For example, an SPL rstring maps to a SPL::rstring that has a C++ base implementation of std::string and an SPL int64 maps to a SPL::int64 that has a C++ base implementation of int64_t.

Table 7-1 describes the total mapping from Modeler types to Streams SPL types:

Table 7-1 Mapping Modeler types to Streams SPL types

Modeler type	XML metadata tag	SP field type (returned from PIM)	C declaration (from SP documentation)	SPL type
String	string	STRING	const char *	rstring
Integer	integer	LONG	long long	int64
Real	real	DOUBLE	double	float64
Time	time	TIME	long long	int64
Date	date	DATE	long long	int64
Timestamp	timestamp	TIMESTAMP	long long	int64

Results

In this section, we show how you can wrap the execution of an SPSS Modeler predictive analytic. We provide insight as to how the sample operator provided could be modified to handle different models.

Note that there are other ways to execute scoring models in InfoSphere Streams through PMML and the Streams Mining Toolkit. The direct wrapping technique provided here opens the scoring up to a much larger set of models than what are supported through the PMML integrations of the Mining Toolkit.

Future possibilities

Some possibilities for extending this work in the future include:

- Producing a generic operator with mapping code from parameters and the xml metadata, allowing for models to be incorporated without the need to modify the operator C++ template code.
- Providing customizable error behavior when execution fails or data is malformed.
- Providing better support for dynamically updating the model. You could just stop and restart the operator causing it to reload the `.pim` and `.par` files, but the modeling toolkit has an optional input port to feed new models and the operator can manage the replacement of the model.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000374

7.3 Streams integration with data stores

In this section, we discuss the capability of Streams to connect and interface with other data stores. We do not attempt to document how to set up the different databases themselves; for that task, you must consult the database documentation.

However, we do describe the steps to configure connectivity to the following database servers:

- DB/2 (9.5 and 9.7)
- Informix
- Oracle
- solidDB
- SQLServer
- MySQL
- Netezza

We discuss the basic concepts, describe some database specific configuration steps, and present examples. In this section, when we refer to a database adapter, we mean an SPL operator that interacts with a database.

Note that there may also be other ways of establishing these connections. Refer to Appendix B, “Toolkits and samples” on page 397, which deals with toolkits, and includes an overview of the Database Toolkit, whose operators are used in this section.

7.3.1 Concepts

In this section, we discuss some of the basic concepts that you need to understand before connecting a Streams application to a database.

Environment variables

For a Streams application to be able to access a database, it needs to know which database product to connect to and where the client libraries for that database product are installed so the environment variables can be set. In particular, some of these environment variables need to be set in the process running the database adapter's code, that is, in the corresponding Processing Element (PE). Furthermore, each of the different database configurations requires different environment variables to be set.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000375

Setting these environment variables can be done in one of two ways:

- Set environment variables in the instance owner's `.bashrc` file, which is the preferred method.
- Specify the values of the environment variables in the DNA.backDoorEvs instance property. An example of how to set these values is as follows:

```
streamtool setproperty -i myInstance
DNA.backDoorEvs=OBCDINI=/mydir/odbc.ini,LD_LIBRARY_PATH=/mydir/libpa
th
```

In addition to the environment variables that are set in the runtime process, you need to set an environment variable (named STREAMS_ADAPTERS_ODBC_*) at compile time that corresponds to the database being accessed. The operators in the Streams Database Toolkit use ODBC, a standard software interface for accessing databases. The only exception to this is the solidDBEnrich operator, which connects directly using the solidDB API.

Furthermore, at compile time, you also need to set the environment variables STREAMS_ADAPTERS_ODBC_[INC,LIB]PATH, based on where the database client library is installed. Note that because these environment variables need to be set at compile time, a Streams application can connect to only one database at a time. However, if you require that your application connects to multiple databases, here are two possible solutions:

- Segment the application into multiple smaller applications, each of which is compiled with a different set of environment variables. The various segments can talk to each other using dynamic connections (import/export streams that are set up at run time).
- If all the databases being considered are supported by the unixODBC driver manager, then the same application could talk to all of them using this interface. For more information about unixODBC, see “unixODBC: A common driver” on page 358.

7.3.2 Configuration files and connection documents

The operators in the Streams Database Toolkit must be configured to connect to a database. This configuration information is specified in an XML document, which is known as the connection specification document. This document (named `connections.xml`) is often complex, detailed, and specific to a particular database, and is therefore kept separate from the SPL application code. We show an example of this later in this section, where the same application code can connect to different databases using different environment variables and a connection specification document.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000376

The connection specification document contains two kinds of specifications:

- Connection specification: This specification refers to the API used, and provides the information needed to establish a connection to the database.
  - For ODBC, this is the information needed for the ODBC SQLConnect() function, and includes three attributes (database name, user ID, and password).
  - For solidDB, this is the information needed for the solidDB SaConnect() function, and includes five attributes (communications protocol, host name, port number, user ID, and password).
- Access specification: This specification refers to the data resources being accessed, and depends on the type of operator, as listed below. For operators that have parameters, the access specification may also include a mapping between the elements of the database table and parameters in the operator. Each access specification must also identify the connection specification to be used. Finally, the access specification must specify the schema of the data received from (or sent to) the database.
  - ODBCEnrich and ODBCSource operators run queries, and therefore must have a query element.
  - ODBCAppend operator add records to a database table, and therefore must have a table element.
  - solidDBEnrich operator run queries against a database table, and therefore must have a tablequery element.

At the time of the writing of this book, the latest fixpack of Streams V2.0 also includes a tool to help you test your database connections (if you are using UnixODBC) outside of a Streams application. This utility, called odbchelper, is part of the Database Toolkit shipped with the product. The source code is provided for this utility, along with the makefile needed to build it. The utility can be invoked using any of the following action flags:

- help: Displays the options and parameters available.
- testconnection: Tests the connection to an external data source instance with a user ID and password.
- runsqlstmt: Runs an SQL statement, either passed in on the command invocation, or in a specified file.
- runsqlquery: Runs an SQL query, either passed in on the command invocation, or in a specified file. The results of the query are returned to STDOUT.
- loaddelimitedfile: Allows you to pass in a comma-delimited file, used to create and populate a database table.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000377

Using the testconnection flag, you can check whether the information in your `connections.xml` file is correct for your external data source. As you may have guessed, the other flags (loaddelimitedfile, runsqlquery, and runsqlstmt) are also useful, allowing you to create database tables and run SQL queries / statements off it to help find and debug setup, connection, and configuration issues.

unixODBC: A common driver

Among the many ODBC drivers, unixODBC deserves special attention because it provides support for databases from multiple vendors. In these cases, the database adapter's code would link to the unixODBC libraries instead of directly to the database client libraries. In fact, if you were trying a database not listed in this section, the first thing to check would be whether the unixODBC libraries support it. If it does, then the database adapters can connect to it using the unixODBC libraries by setting the environment variable STREAMS_ADAPTERS_ODBC_UNIX_OTHER when compiling the application.

For the purposes of this section, we refer to Version 2.3 of unixODBC, which can be downloaded from the following address:

`http://www.unixodbc.org`

The details may be slightly different for older versions.

One thing to understand regarding of using unixODBC is that regardless of what database to which you are connecting, is the different unixODBC configuration files. unixODBC comes with a graphical interface to manipulate these configuration files. The configuration files are designed so that the configuration can be split up across multiple files: the `odbcinst.ini`, `odbc.ini`, and `.odbc.ini` files. In this section, we present some examples. Furthermore, to keep things simple in this section, the examples show entire database configurations in one file (which is perfectly valid). More detailed information about the different unixODBC configuration files can be found at the following address:

`http://www.unixodbc.org`

In the examples presented here, the entire configuration is in a nonstandard location, and the ODBCINI environment variable is set to point to that nonstandard location.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000378

7.3.3 Database specifics

Next, we present more details (including the specific environment variables) for each of the databases discussed. In particular, we specify the variables that need to be set at compile time, and those that need to be set in the processes running the database adapter code (PE run time). Besides these environment variables, you also need to set the environment variables STREAMS_ADAPTERS_ODBC_[INC,LIB]PATH at compile time, based on where the database client library is installed. The environment variables STREAMS_ADAPTERS_ODBC_*, which are set at compile time, and tell the compiler which database product to connect to, do not need to be set to any particular value; they just need to be set.

It is important to note that this book does not necessarily describe the only way, or even the best way, to configure these databases. The following examples are provided as a guide to you, and as a useful place to begin. As an example, in the section that deals with SQLServer, we mention that the FreeTDS driver can be used on the back end. However, this is not the only driver available, and you can as easily use other drivers (such as EasySoft, for example). There may be other client options available for other databases too.

DB2

DB2 connections are accomplished by directly linking to the database client libraries. DB2 has a concept of an instance, and the environment variables dictate what DB2 instance should be used. The following environment variable needs to be set in the runtime processes:

Env Var for running database adapters with DB/2. The environment variable Name is Sample Value. The DB2INSTANCE is myDB2InstanceName.

The ODBC database name used in the `connections.xml` document should match the name of the database instance (myDB2InstanceName). The following environment variable needs to be set prior to compiling SPL applications accessing DB2:

Env Var to be set for compiling database adapters with DB2:
STREAMS_ADAPTERS_ODBC_DB2

Informix

Informix connections are accomplished by directly linking to the database client libraries. The following environment variable needs to be set prior to compiling SPL applications accessing Informix:

Env Vars to be set for compiling database adapters with Informix:
STREAMS_ADAPTERS_ODBC_IDS

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000379

In addition, you need an `sqlhosts` file (pointed to by the INFORMIXSQLHOSTS environment variable), which will contain the following information:

- dbservername: Database server name
- nettype: Connection type
- hostname: Host computer for the database server
- servicename: Alias for port number
- options: Options for connection (optional field)

The following is an example `sqlhosts` file:

```
inf115  onsoctcp        myHost     55000
```

Informix also relies on an `odbc.ini` file that is similar to the ones used by unixODBC. This `odbc.ini` file (pointed to by ODBCINI environment variable) looks something like the following lines:

```
[itest]
Description = Informix
Driver =<installed Informix directory>/lib/cli/iclit09b.so
APILevel=1
ConnectFunctions=YYY
DriverODBCVer=03.00
FileUsage=0
SQLLevel=1
smProcessPerConnect=Y
ServerName=inf115
Database=adaptersTest
Port=55000
CLIENT_LOCALE=en_us.8859-1
DB_LOCALE=en_us.8859-1
TRANSLATIONDLL=<installed Informix directory>/lib/esql/igo4a304.so
```

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (itest). Note that the ServerName field in the `/mydir/odbc.ini` file (inf115) should match the name of the server name specified in the `/mydir/sqlhosts` file. To summarize, the environment variables need to be set in the runtime processes, as shown in Table 7-2.

Table 7-2 Env Vars for running database adapters with Informix

Environment variable name	Sample value
INFORMIXDIR	<installed Informix directory>
INFORMIXSQLHOSTS	/mydir/sqlhosts
ODBCINI	/mydir//odbc.ini

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000380

Oracle

Oracle connections are accomplished by linking to the database client through unixODBC with the Oracle driver on the back end. The following environment variable needs to be set prior to compiling SPL applications accessing Oracle:

Env Var to be set for compiling database adapters with Oracle:
STREAMS_ADAPTERS_ODBC_ORACLE

You also need an `odbc.ini` file (pointed to by the ODBCINI environment variable), which will look something like the following lines:

```
[twins]
Driver                       = <installed Oracle
directory>/lib/libsqora.so.11.1
description                  = test
ServerName                   = 10.1.1.92:1521/twins.mydomain.com
server                       = 10.1.1.92
port                         = 1521
Longs                      = F
```

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (twins). Furthermore, with Oracle, you need to know whether its libraries are dynamically or statically linked, which is set by LD_LIBRARY_PATH. To summarize, the environment variables in Table 7-3 need to be set in the runtime processes.

Table 7-3 Env Vars for running database adapters with Oracle

Environment variable name	Sample value
ODBCINI	/mydir/odbc.ini
ORACLE_HOME	<installed Oracle directory>
LD_LIBRARY_PATH	<installed Oracle directory>/lib (if using static linking) OR <installed Oracle directory>/lib:/mydir/unixodbc/lib (if NOT using static linking)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000381

SQLServer

SQLServer connections are also accomplished by linking to the database client through unixODBC. The FreeTDS driver can be used on the back end, and can be downloaded from the following address:

http://www.freetds.org

The following environment variable needs to be set prior to compiling SPL applications accessing SQLServer:

Env Var to be set for compiling database adapters with SQLServer:
STREAMS_ADAPTERS_ODBC_SQLSERVER

You also need an `odbc.ini` file (pointed to by the ODBCINI environment variable), which will look something like the following lines:

```
[mstest]
Driver              = /myTDSDir/freetds/lib/libtdsodbc.so
Description         = SQL Server
Trace               = no
TDS_Version         = 7.0
Server              = myHost.foo.com
Port                = 1136
Database            = msdb
```

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (mstest). The environment variable shown in Table 7-4 needs to be set in the runtime processes.

Table 7-4 Env Var for running database adapters with SQLServer

Environment variable name	Sample value
ODBCINI	/mydir/odbc.ini

MySQL

SQLServer connections are also accomplished by linking to the database client through unixODBC with the MySQL driver on the back end. The following environment variable needs to be set prior to compiling SPL applications accessing MySQL:

Env Var to be set for compiling database adapters with MySQL:
STREAMS_ADAPTERS_ODBC_MYSQL

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000382

You need an `odbc.ini` file (pointed to by the ODBCINI environment variable), which will look something like the following lines:

```
[mysqltest]
Driver              =
/myMySQLDir/mysql-connector-odbc-5.1.7-linux-glibc2.3-x86-64bit/lib/lib
myodbc5.so
Description  = Connector/ODBC 3.51 Driver DSN
Server              = myHost
PORT         = 3306
Database     = UserDB
OPTION       = 3
SOCKET       =
```

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (mysqltest). The environment variable shown in Table 7-5 needs to be set in the runtime processes.

Table 7-5 Env Var for running database adapters with MySQL

Environment variable name	Sample value
ODBCINI	/mydir/odbc.ini

Netezza

Netezza connections are accomplished by linking to the database client through unixODBC with the Netezza driver on the back end. The following environment variable needs to be set prior to compiling SPL applications accessing Netezza:

Env Var to be set for compiling database adapters with Netezza:
STREAMS_ADAPTERS_ODBC_NETEZZA

You also need an `odbc.ini` file (pointed to by the ODBCINI environment variable), which will look something like the following lines:

```
[netezza]
Driver           = /myNetezzaDir/lib64/libnzodbc.so
Setup            = /myNetezzaDir/lib64/libnzodbc.so
APILevel         = 1
ConnectFunctions = YYN
Description      = Netezza ODBC driver
DriverODBCVer    = 03.51
DebugLogging     = false
LogPath          = /tmp
UnicodeTranslationOption = utf8
CharacterTranslationOption = all
PreFetch         = 256
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000383

```
Socket          = 16384
Servername           = 127.0.1.10
Port                 = 5480
Database             = streamstest
ReadOnly             = false
```

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (netezza). The environment variables in Table 7-6 need to be set in the runtime processes.

Table 7-6 Env Vars for running database adapters with Netezza

Environment variable name	Sample value
ODBCINI	/mydir/odbc.ini
NZ_ODBC_INI_PATH	/mydir (directory containing the `odbc.ini` file)

solidDB with ODBC operators

Streams applications using ODBCSource, ODBCEnrich, and ODBCAppend operators can connect to solidDB through unixODBC using the solidDB client driver. The following environment variable needs to be set prior to compiling SPL applications accessing solidDB with unixODBC:

Env Var to be set for compiling database adapters with solidDB (with unixODBC):
STREAMS_ADAPTERS_ODBC_SOLID

In addition, you need a `solid.ini` file (whose parent directory is pointed to by the SOLIDDIR environment variable), which will contain something like the following lines:

```
[Data Sources]
solidtest=tcp myHost 1964
```

You also need an `odbc.ini` file (pointed to by the ODBCINI environment variable), which looks something like the following lines:

```
[solidtest]
Description     = solidDB
Driver          = <installed solidDB directory>/bin/sacl2x6465.so
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000384

The data source name used in the `connections.xml` document should match the data source name in `odbc.ini` (solidtest). The environment variables shown in Table 7-7 need to be set in the runtime processes.

Table 7-7 Env Vars for running database adapters with solidDB (with unixODBC)

Environment variable name	Sample value
ODBCINI	/mydir/odbc.ini
SOLIDDIR	/mydir (directory containing the `solid.ini` file)

solidDB with solidDBEnrich operator

solidDB connections using the solidDBEnrichOperator are done with direct linking. No extra environment variables (other than the [INC/LIB]PATH) need to be set when compiling the database adapter's code or during run time.

7.3.4 Examples

In this section, we present some examples to illustrate the two main ways of setting environment variables. At the same time, we describe how to connect to two of the databases (DB2 and Informix). In both examples, the application is the same (the Main Composite is named singleinsert). The SPL code in the application uses an ODBCAppend operator that is coded as follows:

```
    () as writeTable = ODBCAppend(singleinsert) {
      param
        connection         : "DBPerson";
        access             : "PersonSinkSingleInsert";
        connectionDocument : "connections.xml";
}
```

Environment variables set in connecting to DB2

In this example, we show how a DB2 connection can be configured by placing the runtime environment variables in the instance owner's `.bashrc` file.

The `connections.xml` file specified by ODBCAppend has a connection specification that looks like the following lines:

```
<connection_specification name="DBPerson" > <ODBC database="spc97"
user="mike" password="myPassword" /></connection_specification>
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000385

Recall that DB2 connections are accomplished by directly linking to the database client libraries. The code is compiled by the following commands, first setting up the environment variables using the command line:

```
export STREAMS_ADAPTERS_ODBC_DB2
export STREAMS_ADAPTERS_ODBC_INCPATH=<installed DB2 directory>/include
export STREAMS_ADAPTERS_ODBC_LIBPATH=<installed DB2 directory>/lib64
sc -s -v -t /mydir/InfoSphereStreams/toolkits/com.ibm.streams.db   -a
-M singleinsert
```

The following lines are added to the `.bashrc` file so that the environment variables are picked up by the PEs:

```
export DB2INSTANCE=spc97
```

The Streams instance is created and started with the following default values:

```
streamtool mkinstance -i myInstance -hosts myHost
streamtool startinstance -i myInstance
```

When submitted, the application starts, connects successfully, and data is written to the database by using the following command:

```
streamtool submitjob -i myInstance output/singleinsert.adl
```

Environment variables set in connecting to Informix

In this example, we show how an Informix connection could be configured by placing the runtime environment variables in the instance DNA.backDoorEvs instance property.

The `connections.xml` file specified by ODBCAppend has a connection specification that might look like the following lines:

```
<connection_specification name="DBPerson" > <ODBC database="mike"
user="myPassword" password="db2expr1" /></connection_specification>
```

Recall that Informix connections are accomplished by directly linking to the database client libraries. The code is compiled by running the following commands:

```
export STREAMS_ADAPTERS_ODBC_IDS
export STREAMS_ADAPTERS_ODBC_INCPATH=/<installed Informix
directory>/incl/cli
export STREAMS_ADAPTERS_ODBC_LIBPATH=/<installed Informix
directory>/lib
sc -s -v -t /mydir/InfoSphereStreams/toolkits/com.ibm.streams.db   -a
-M singleinsert
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000386

For Informix, we set up an `sqlhost` file and an `odbc.ini` file, as is previously described in this chapter.

The Streams instance is created by suing the DNA.backDoorEvs property and then started using the following commands:

```
streamtool mkinstance -i myInstance -property
DNA.backDoorEvs=INFORMIXDIR=<installed Informix
directory>,INFORMIXSQLHOSTS=/mydir/sqlhosts,ODBCINI=/mydir/odbc.ini
streamtool startinstance -i myInstance
```

When submitted, the application starts, connects successfully, and data is written to the database by running the following command:

```
streamtool submitjob -i myInstance output/singleinsert.adl
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000387

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000388

A

InfoSphere Streams installation and configuration

In this appendix, we detail the installation and configuration of the IBM InfoSphere Streams (Streams) software products, both the runtime system and Streams Studio. Step-by-step instructions provide an easy to follow procedure to create either a single-host or a multi-host Streams environment sufficient for replicating the SPL samples that come with the Streams installation media and those included in this book.

For more advanced topics, such as configuring multi-user environments, enabling the SELinux feature, recovery support, LDAP integration, High Availability, and others that are relevant primarily for production installations, refer to the Streams documentation available at the Streams Information Center website found at the following address:

http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000389

Installation considerations and requirements for Streams

To install Streams, you need the following components:

- Streams installation media

  You must download the correct version of the Streams software distribution file for either the 32-bit or 64-bit Red Hat Enterprise Linux (RHEL) operating system. For supported RHEL versions, refer to the Streams documentation.

- License certificate file

  To install Streams (other than the trial version), you need to obtain a valid license certificate file, which can be downloaded from the IBM Passport Advantage® website found at the following address:

  `http://www.ibm.com/software/howtobuy/passportadvantage`

- Red Hat RPMs

  There is a set of RPMs that need to be applied to each host before installing Streams. The Streams install package contains the dependency checker script that should be run on the target computer to get the list of the required RPMs. An RPM that belongs to this list is either provided on the RHEL installation media or included with the Streams install package.

- Eclipse SDK

  To take advantage of the benefits offered by Streams Studio, you need Eclipse SDK Version 3.6.2 or higher. The Streams Studio install components are provided in the form of Eclipse plug-ins. They are copied to the target computer by the Streams install utility. To install the Streams plug-ins, use the Eclipse interface. A supported version of the Eclipse SDK can be downloaded from the following address:

  `http://download.eclipse.org/eclipse/downloads`

There are several options to consider before installing Streams:

- Interactive GUI install versus interactive console install

  There is no significant difference between these two types of installation methods. Functionally, both are identical; however, the GUI installation method requires the X Window System interface, while the console method does not. If an X Window System interface is present on the machine where you install Streams, the installation by default will be performed in the GUI mode. Otherwise, the only option is the console install.

Note: An X Window System interface is required for Streams Studio.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000390

Streams also could be installed silently using a response file. This method is useful for repeatable installations on multiple hosts. For more information about the silent install method, refer to the Streams documentation.

- Shared directory install versus local drive install

  With two or more hosts, a Streams runtime system could be either installed locally on each host, or it could be placed in a shared directory accessible from each member of the Streams cluster. Shared installations are easier to deploy and maintain, but if you are looking for maximum performance, you should consider installing Streams on each host locally. You also have to use this install option if you want to enable the SELinux security-enhanced feature of Streams, which is not supported on shared file systems. Regardless of the cluster installation method, the home directory of the install owner and the home directory of the instance owner must be in a shared file system.

  Note: Each cluster host must be at the same operating system level, have the same RPMs, and have the same CPU architecture.

- Root install versus non-root user install

  Streams can be installed either by the root user or a non-root user. A non-root user who is installing the product becomes the owner of the Streams installation files. If you decide to install Streams as root, you will be prompted by the install utility for a non-root user ID and group who will own the installed files. This user must be created prior launching the install utility. A common name for an owner of the Streams installation is streamsadmin.

  Note: To enable the SELinux support, Streams must be installed by the root user.

Installing Streams on a single host

In this section, we provide details about how to install Streams runtime system and Streams Studio on a single computer. For the purpose of this demonstration, we use a 32-bit RHEL 5.5 VMWare image configured with 2 GB of RAM and hosted by a PC running Microsoft Windows XP.

Complete the following steps:

1. Create a user that will own the Streams installation (we refer to this user as streamsadmin).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000391

2. Prepare the Streams install package.

   Download and unpack the Streams installation media into a temporary location using the following command:

   ```
   tar -zxvf streams-install-package-name.tar.gz
   ```

   As a result, the StreamsInstallFiles directory will be created with the self-extracting install binary file `InfoSphereStreamsSetup.bin` and other install files in it. Copy the license certificate file to the `StreamsInstallFiles` directory.

3. Run the Streams dependency checker.

   The Streams install package contains the `dependency_checker.sh` script that validates the pre-install requirements, including the operating system version, installed RPMs, network configuration, and so on. To run the dependency checker, enter the following commands:

   ```
   cd streams-install-package-directory/StreamsInstallFiles
   ./dependency_checker.sh
   ```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000392

In Figure A-1, we show a sample output of this script. Review the dependency checker script output for errors and warnings before proceeding to the next step.

```
IBM InfoSphere Streams 2.0.0.0 Dependency Checker
Date:  Tue Jul 12 20:05:01 EDT 2011

=== System Information ===
* Hostname:  streamsvm.myvm
* IP address:  192.168.145.131
* Operating system:  Red Hat Enterprise Linux Server release 5.5 (Tikanga)
* System architecture:  i386
* Security-Enhanced Linux setting:  Disabled
* Java vendor:  IBM Corporation
* Java version:  1.6.0
* Java VM version:  2.4
* Java runtime version:  pxi3260sr9ifx-20110211_02 (SR9)
* Java full version:  JRE 1.6.0 IBM J9 2.4 Linux x86-32 jvmxi3260sr9-20101124_69295
J9VM - 20101124_069295
JIT  - r9_20101028_17488ifx2
GC   - 20101027_AA
* Java IBM system encoding:  UTF-8
* Encoding:  UTF-8

=== System Configuration Check ===
* Status:  PASS - Check:  License certificate file check
* Status:  PASS - Check:  Hostname and IP address check
* Status:  PASS - Check:  Operating system version check
* Status:  PASS - Check:  Architecture check
* Status:  PASS - Check:  Java check
* Status:  PASS - Check:  Encoding check

=== Software Dependency Package Check ===
* Status:  CORRECT LEVEL - Package:  perl-XML-Simple, System Version:  2.14-4.fc6
* Status:  CORRECT LEVEL - Package:  perl-XML-SAX, System Version:  0.14-8
* Status:  CORRECT LEVEL - Package:  gcc-c++, System Version:  4.1.2-50.el5
* Status:  CORRECT LEVEL - Package:  curl-devel, System Version:  7.15.5-9.el5_6.2
* Status:  CORRECT LEVEL - Package:  graphviz, System Version:  2.24.0-1.el5
* Status:  CORRECT LEVEL - Package:  ibm-java-i386-sdk, System Version:  6.0-9.0

=== Summary of Errors and Warnings ===

The dependency checker completed with no errors or warnings.
```

Figure A-1 Dependency checker

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000393

4. Install required RPMs.

   Examine the output of the dependency checker script and identify the RPMs that have to be installed (or upgraded) on your system. Install RPMs using the **yum install command**. To install an RPM shipped with the Streams install package, log in as root and enter the following commands:

   ```
   cd streams-install-package-directory/StreamsInstallFiles/rpm
   yum install --nogpgcheck rpm-file-name
   ```

   For instruction about how to install the RPMs that are provided with RHEL, refer to the RHEL documentation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000394

5. Disable the firewall.

   As root, select **System** → **Administration** → **Security Level and Firewall**, as shown in Figure A-2.



Figure A-2 Security Level and Firewall menu

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000395

Select **Disabled** and click **Apply**, as shown in Figure A-3.

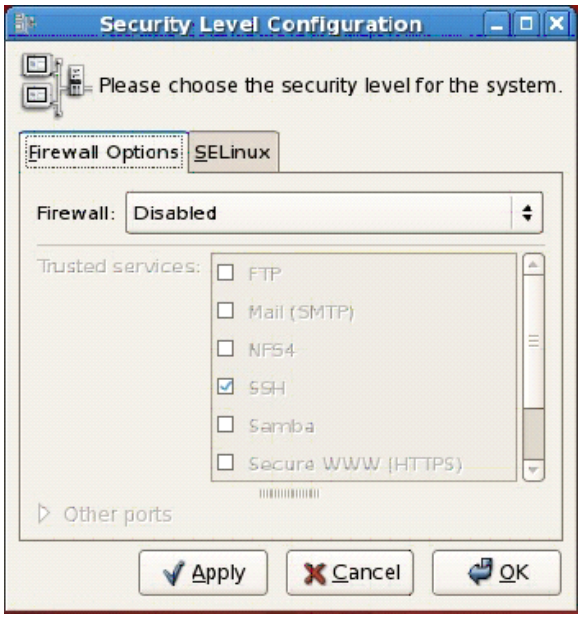


Figure A-3 Disable the firewall

6. Configure the network.

   Ensure that the name of the target machine returned by the **`hostname`** command matches an entry in the `/etc/hosts` file. For example, if the **`hostname`** command returns `myhost.mydomain` and the IP address of the target host is 192.168.145.9, the `/etc/hosts` file must contain the following line:

   ```
   192.168.145.9 myhost.mydomain myhost
   ```

   Also make sure that the `/etc/hosts` file contains the following line:

   ```
   127.0.0.1 localhost.localdomain localhost
   ```

7. Set the character encoding of your locale to UTF-8.

   Refer to your Red Hat documentation for details about how to set the character encoding.

8. Ensure that the streamsadmin user is able to run applications in the GUI mode using the X Window System interface.

   Refer to your Red Hat documentation for details about how to enable the X Window System interface. You must copy the `LicenseCert*.txt` file into `streams-install-package-directory`.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000396

9. Run the install utility.

   Log in as streamsadmin and enter the following commands:

   ```
   cd streams-install-package-directory/StreamsInstallFiles
   ./InfoSphereStreamsSetup.bin
   ```

   The Welcome window opens (Figure A-4).

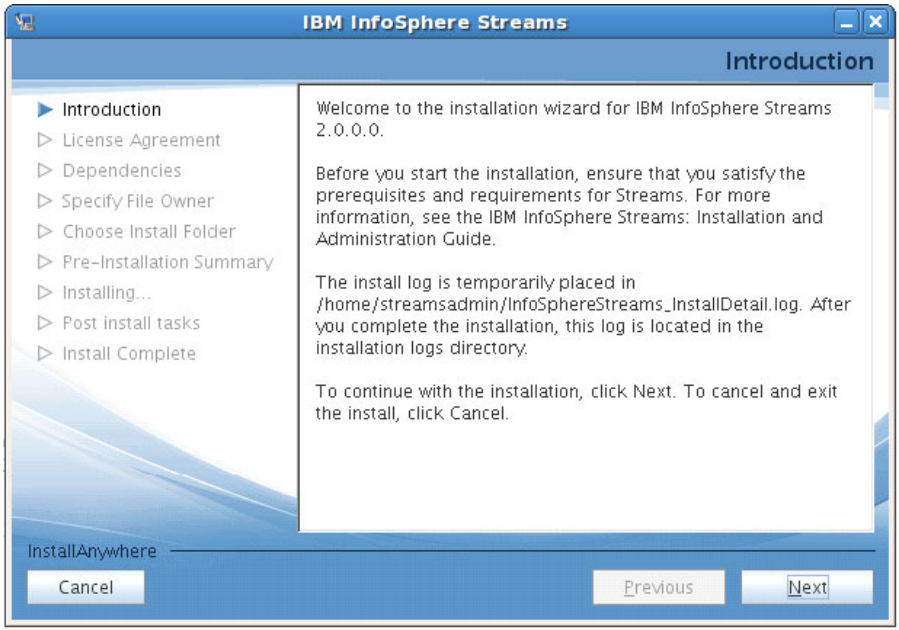


Figure A-4 Welcome window

To continue, click **Next**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000397

10. Accept the license agreement.

    Accept the terms and license agreement and click **Next**, as shown in Figure A-5.

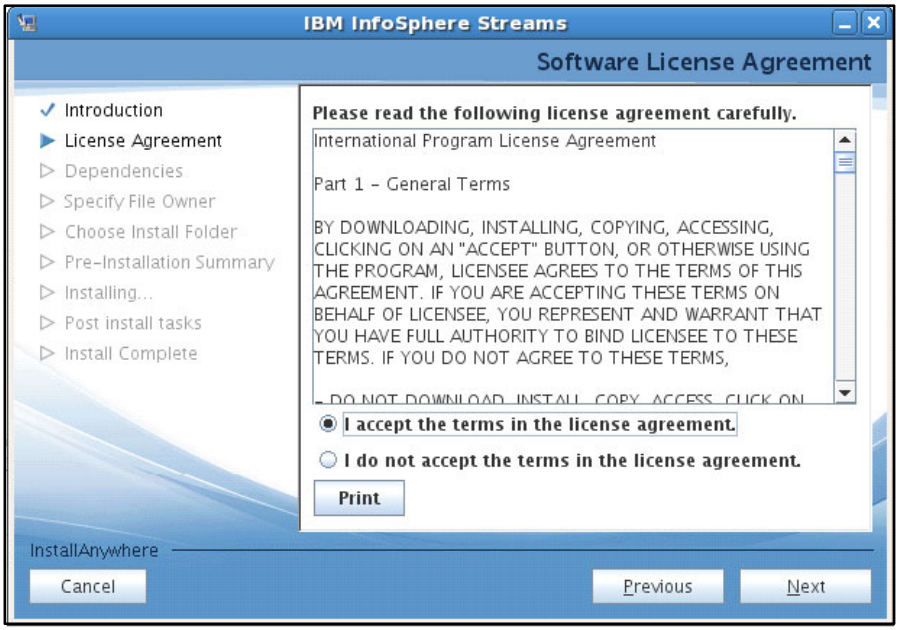


Figure A-5 Software License Agreement window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000398

11. Review dependencies.

Review the results of the software dependency check and click **Next**, as shown in Figure A-6.

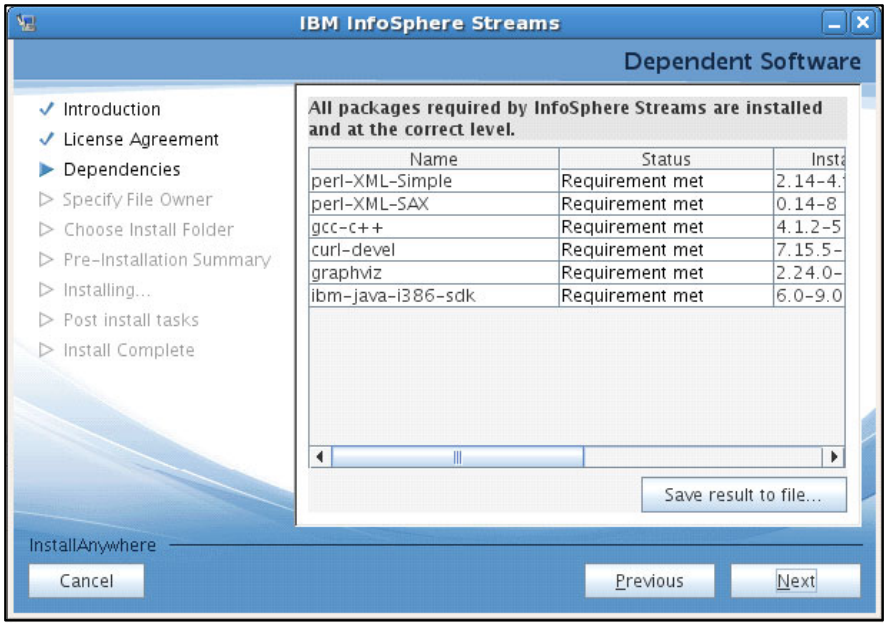


Figure A-6 Dependent Software window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000399

12. Choose the install folder.

    By default, the installed files are placed in the home directory of the non-root user who is installing Streams. Use the default path of `/home/streamsadmin/InfoSphereStreams`. The window shown in Figure A-7 opens.

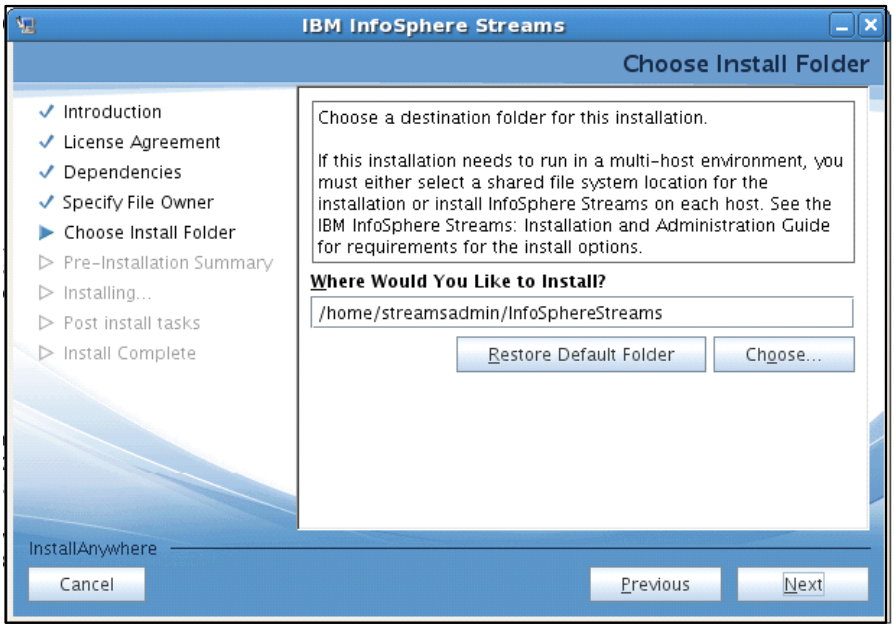


Figure A-7 Install folder

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000400

13. Review the pre-installation summary.

Review the information shown in Figure A-8, and click **Install**.

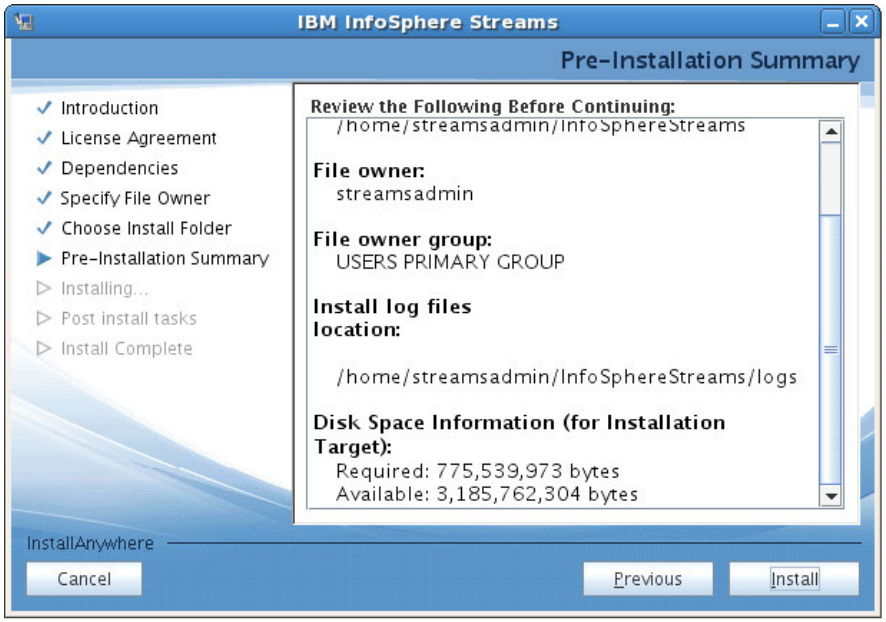


Figure A-8 Pre-installation Summary window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000401

14. Complete the install.

    To exit the Streams install utility, click **Done** on the Install Complete window, as shown in Figure A-9.

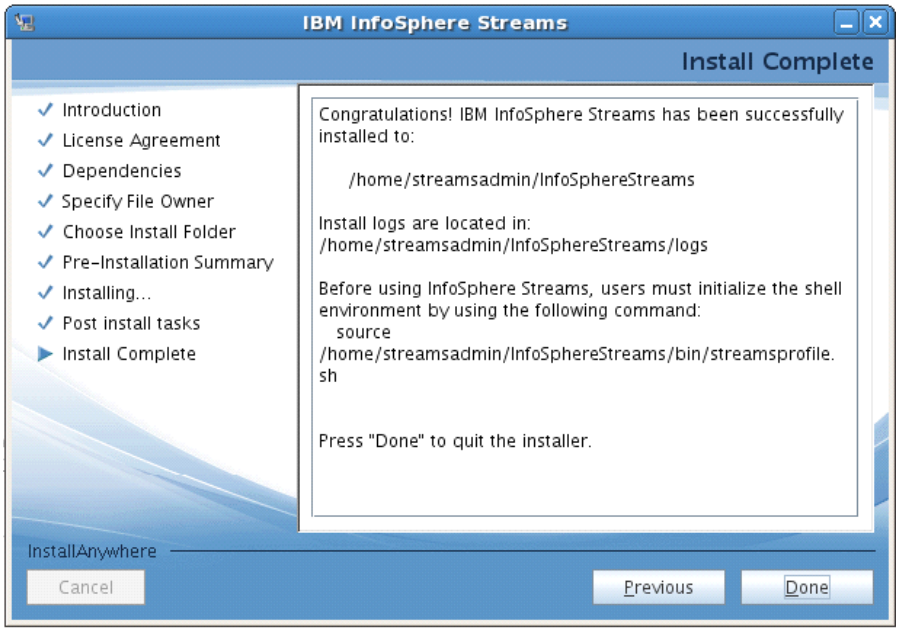


Figure A-9 Install Complete window

15. Configure the Streams environment.

    Add the following command to the streamsadmin's ~/.bashrc file:

    ```
    source streams-install-directory/bin/streamsprofile.sh
    ```

    Log out before executing the next steps.

16. Configure the secure shell environment.

    Complete the following steps to configure secure shell (ssh) environment for Streams:

    a. Log in as streamsadmin and enter the following command:

       ```
       ssh-keygen -t dsa
       ```

       The command prompts you to provide the file name for saving the key. Press Enter to accept the default name. Next, the command prompts you for a passphrase. Press Enter twice, providing an empty passphrase. If the streamsadmin's home directory does not contain the `.ssh` subdirectory, it will be created by the **`ssh-keygen`** command.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000402

b. Change to the .ssh directory:

```
cd .ssh
```

c. Append the public key to the `authorized_keys` file:

```
cat id_dsa.pub >> authorized_keys
```

d. Change the permissions on the files in the `.ssh` directory:

```
chmod 600 *
```

e. Ensure that only the owner has write access to their home directory by entering the following command:

```
chmod go-w ~
```

f. To test the ssh setting, use the **`streamtool checkhost`** command:

```
streamtool checkhost --phase1-only --xinstance
```

17. Configure RSA authentication.

Log in as streamsadmin and enter the following command:

```
streamtool genkey
```

18. Install Streams Studio.

Streams Studio is an integrated development environment (IDE) based on Eclipse SDK. To install Streams Studio, complete the following steps:

a. Download the appropriate version of the Eclipse SDK (Version 3.6.2 and higher) compatible with your operating system (32-bit or 64-bit). Log in as streamsadmin and install the Eclipse SDK by unpacking the downloaded file into the streamsadmin's home directory:

```
tar -zxvf eclipse-SDK-<platform-version-details>.tar.gz
```

b. As a result of the previous step, the eclipse directory will be created in the streamsadmin's home directory. Launch the Eclipse SDK by executing the following command:

```
~/eclipse/eclipse &
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000403

Choose the workspace directory and click **OK**, as shown in Figure A-10.

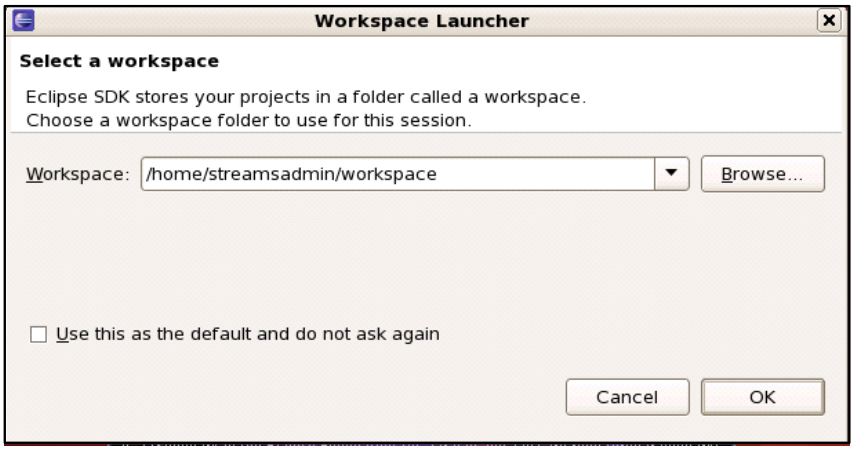


Figure A-10 Workspace Launcher window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000404

c. Close the Welcome window and select **Help** → **Install New Software**. You will see the Available Software window, as shown in Figure A-11. From the Work with drop-down box, select **Helios - http://download.eclipse.org/releases/helios**.

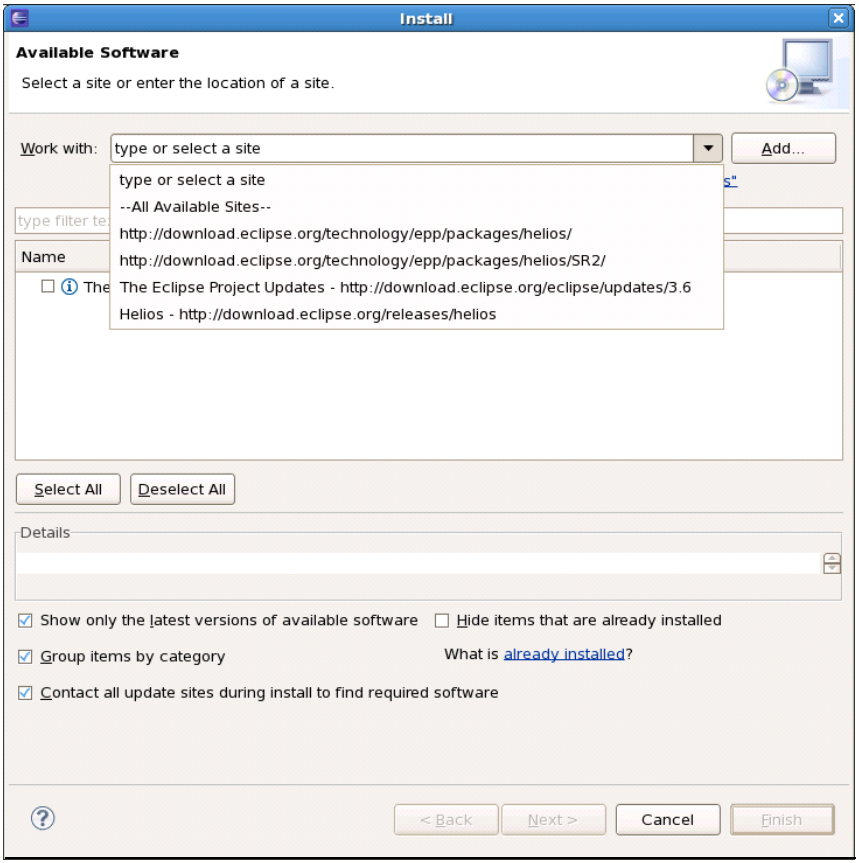


Figure A-11 Available Software window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000405

Expand the **General Purpose Tools** option and select **XText SDK**, as shown in Figure A-12. Click **Next**.

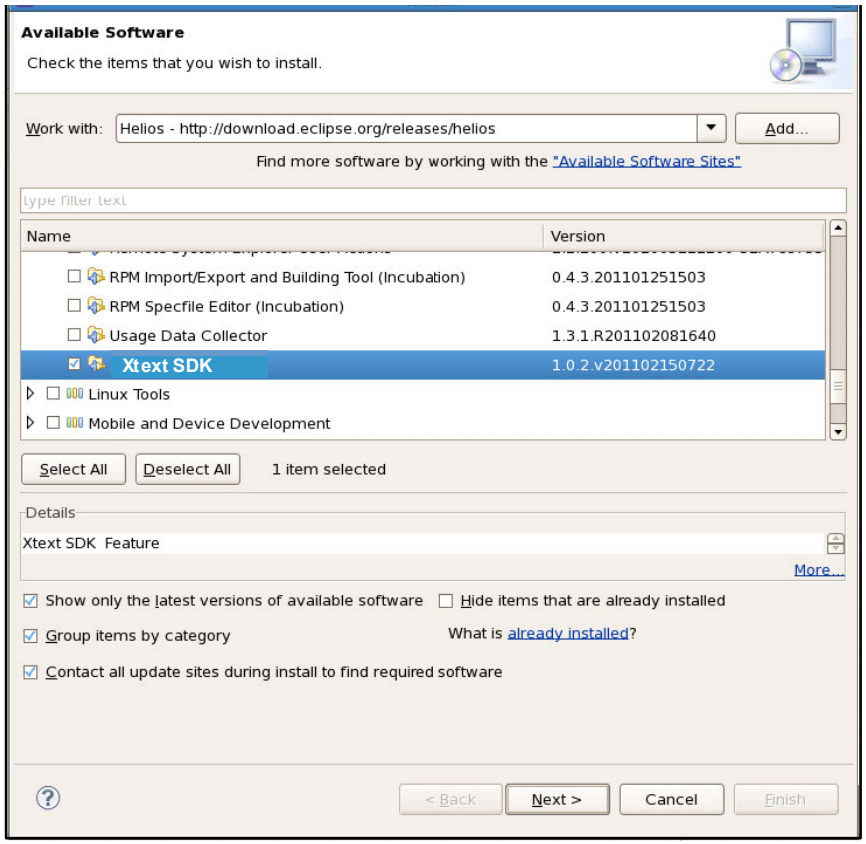


Figure A-12 XText SDK window

Complete the XText installation and restart the Eclipse IDE when prompted.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000406

d. To install the Streams Studio plug-ins, select **Help** → **Install New Software** and click **Add**. On the Add Repository window, click **Local,** select the `<streams-install-directory>/etc/eclipse` directory, and close the Add Repository window by clicking **OK**. The window shown in Figure A-13 opens.

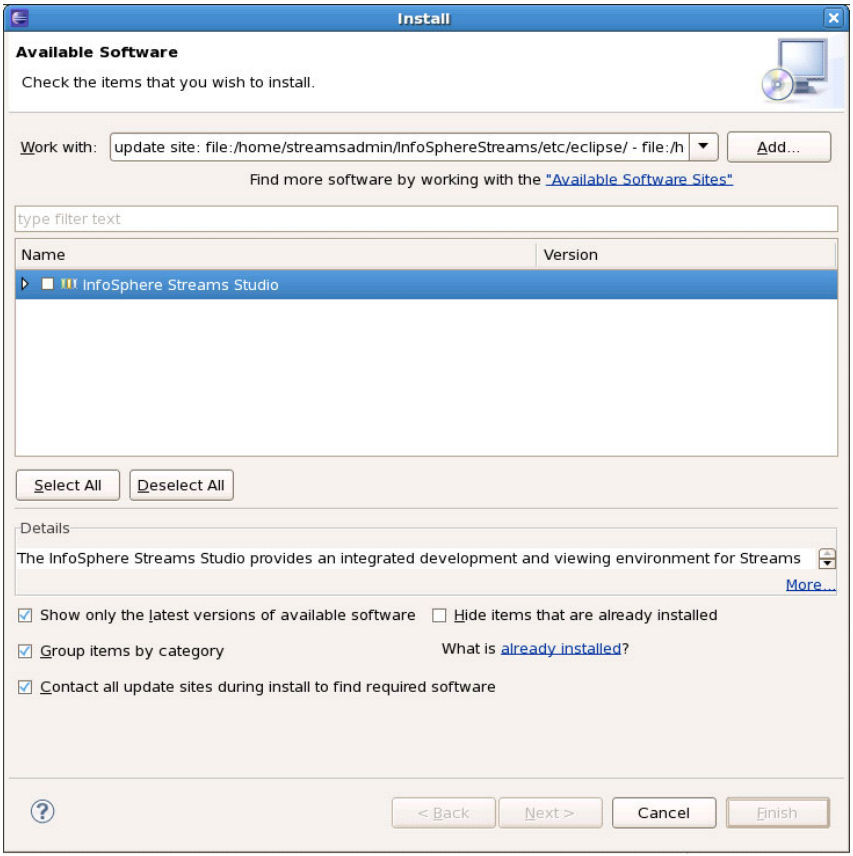


Figure A-13 Streams Studio Installation window

Click **Select All** and follow the prompts to complete the Streams Studio installation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000407

19. Verify your Streams installation.

    You can verify your Streams installation by building and running one of the SPL samples included with Streams:

    a. Log in as streamsadmin and launch the Eclipse IDE. Select **File** → **Import**. On the Import window, expand the InfoSphere Streams Studio entry, select **SPL Project**, and click **Next** (Figure A-14).

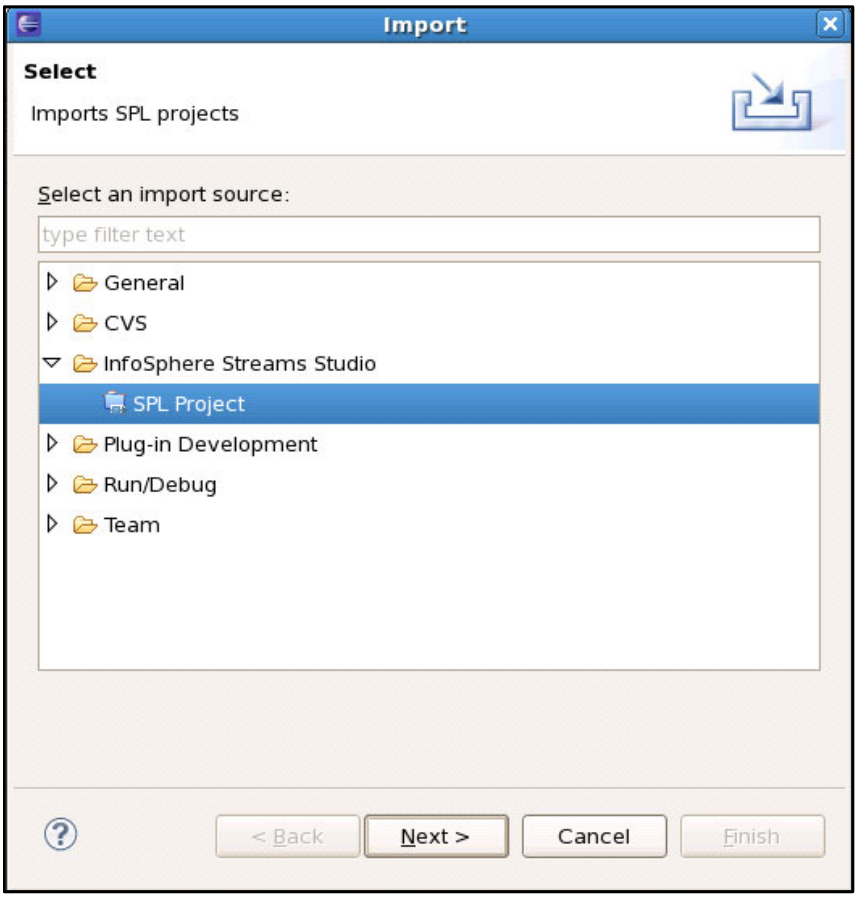


Figure A-14 Select an Import Source

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000408

On the Import SPL Project window, click **Browse** and then navigate to the `/home/streamsadmin/InfoSphereStreams/samples/spl/feature/RegularExpression` directory and click **OK**. The window shown in Figure A-15 opens.

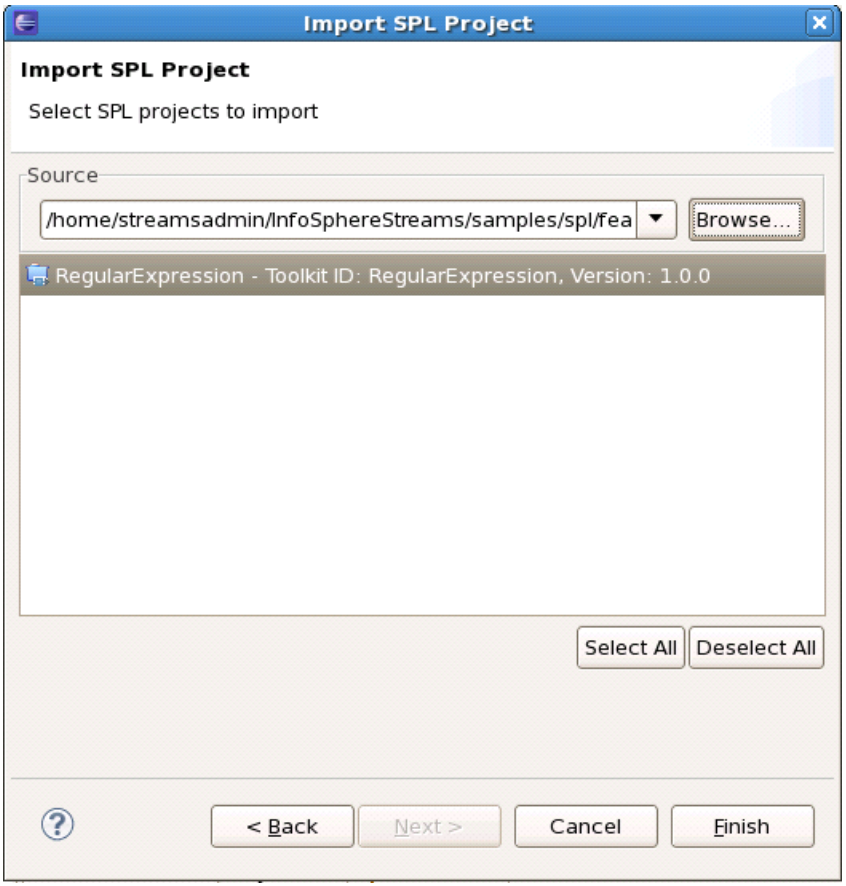


Figure A-15 Select an SPL Project to Import

Click **Select All** and then **Finish**.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000409

b. Select **Window** → **Open Perspective** → **Other menu**, select **InfoSphere Streams**, and click **OK**. In the Project pane, drill down to **RegularExpression** → **sample** → **DateTimeFormatter**. Right-click the **DateTimeFormatter** entry, select **New** → **Standalone Build**, and click **OK**. Expand **DateTimeFormatter** in the Project pane, right-click the **Standalone** entry, and select **Set active** from the menu. The SPL application build should start automatically. You can monitor the build process in the Console pane of the Streams Studio IDE, as shown in Figure A-16.

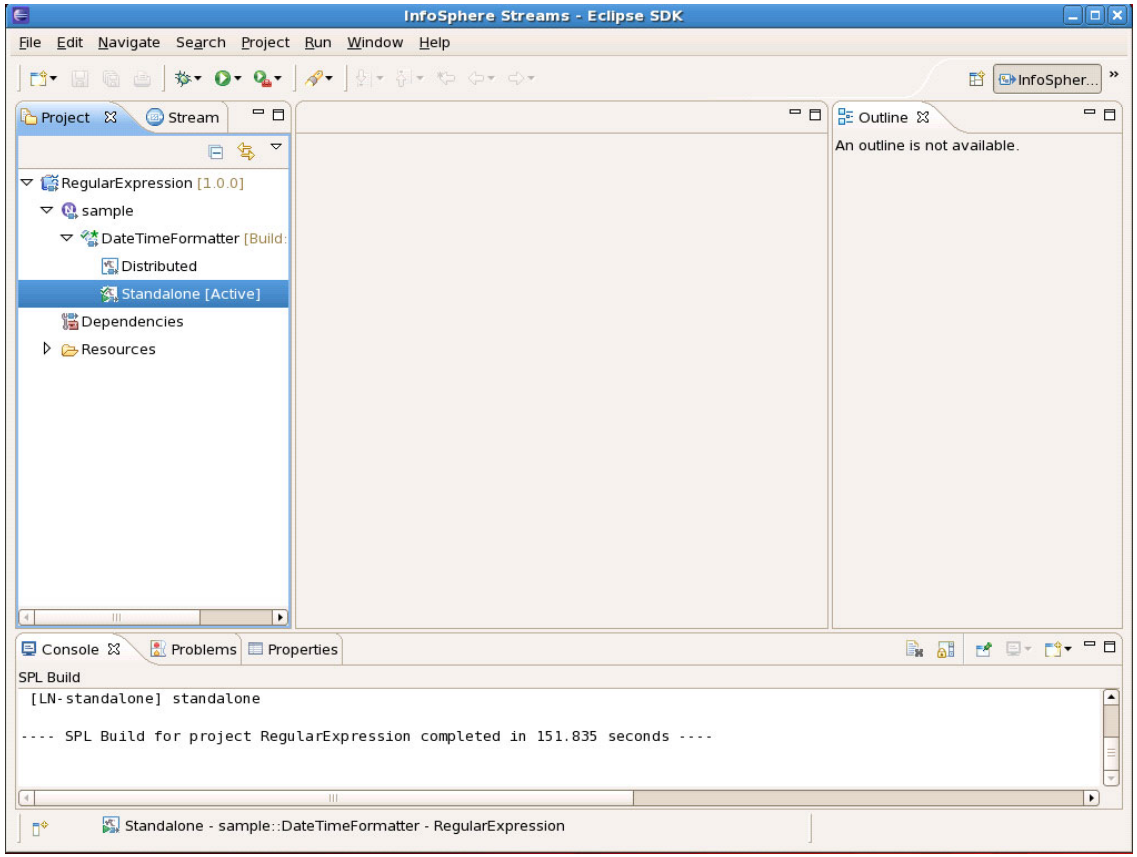


Figure A-16 Streams Studio Console

c. After the build has finished, right-click the **Standalone** entry and select **Launch**. The following message should appear shortly in the Console window:

<terminated> sample::DateTimeFormatter

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000410

Expand the **Resources** entry in the Project pane, right-click **data,** and choose **Refresh**. Expand the data entry. It should contain three items (SinkDataR.txt, SinkDataU.txt, and SourceData.txt), as shown in Figure A-17.

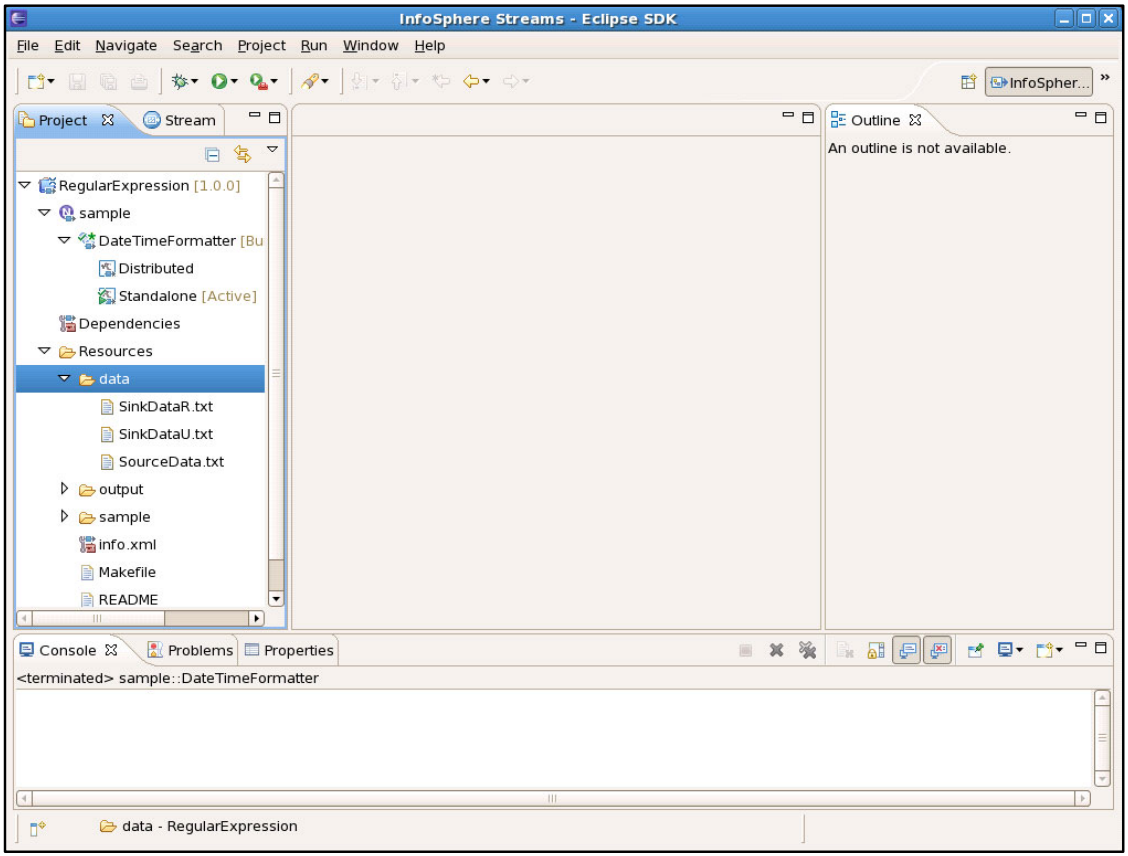


Figure A-17 SPL Project data directory

Refer to the Streams documentation for details about how to build distributed SPL applications, how to create Streams instances, and how to launch distributed applications using Streams Studio.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000411

Installing Streams on multiple hosts

In this section, we describe how to install Streams on a cluster of hosts. We provide instructions about how to configure the Network File System (NFS) and how to install Streams in a shared directory by root user. Two identical 32-bit RHEL 5.5 VMWare images, both configured with 1 GB of RAM and hosted by a single physical computer with 3 GB of RAM running Microsoft Windows XP, provide a sufficient environment for demonstrating this type of the Streams installation. We refer to the target computers as host1 and host2. Table A-1 contains the network settings used in our installation:

Table A-1 Network settings

Component	host1	host2
Host name	streamsnode1	streamsnode2
Domain	mydomain	mydomain
IP Address	192.168.160.131	192.168.160.132

The following steps will guide you through the installation procedure:

1. Create the streamsadmin user.

   On both host1 and host2, create the user that will own the Streams installation files. Name this user streamsadmin. Make sure that the streamsadmin's password, group name, user ID, and group ID on host1 are equal to the corresponding values on host2.

2. Prepare the Streams install package on host1 (see step 2 on page 372).

3. Run dependency checker on host1 and host2 (see step 3 on page 372).

4. Install required RPMs on host1 and host2 (see step 4 on page 374).

5. Disable the firewall on host1 and host2.

6. Configure the network.

   To satisfy the network requirements for the Streams hosts (refer to the network resolution requirements in the Streams documentation), you can alter the `/etc/hosts` file on each host to provide the required name service. Here are the lines that have been added to the `/etc/hosts` file on both host1 and host2:

```
192.168.160.131    streamsnode1. mydomain    streamsnode1
192.168.160.132    streamsnode2. mydomain    streamsnode2
localhost must resolve to 127.0.0.1:
127.0.0.1  localhost.localdomain  localhost
```


Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000412

Note: The RHEL virtual machines that we use for the Streams installation are configured with the network connection property set to Network Address Translation (NAT), as shown in Figure A-18.

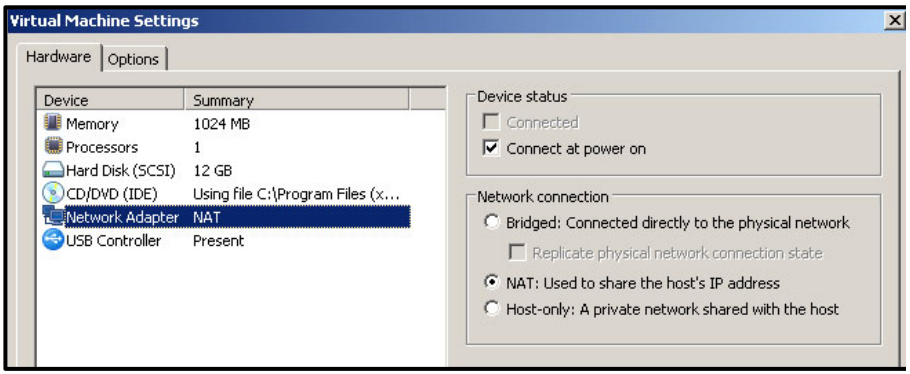


Figure A-18 Virtual Machine Settings window

For more information regarding NAT and other virtual machine settings, refer to the VMWare documentation.

7. Configure NFS.

   In our example, we configure NFS to share (in read and write mode) the `/opt` and `/home` directories of host1 to host2:

   a. Log in to host1 as root and modify the `/etc/exports` file to include the following lines:

      ```
      /home*(rw,no_root_squash)
      /opt*(rw,no_root_squash)
      ```

   b. Restart host1.

   c. Log in to host2 as root and modify the `/etc/fstab` file by adding these two lines at the end:

      ```
      streamsnode1.mydomain:/home  /home  nfs  rw,hard,_netdev,intr  0 0
      streamsnode1.mydomain:/opt      /opt      nfs  rw,hard,_netdev,intr  0 0
      ```

   d. Edit the `/etc/rc.local` file on host2 and add the following two lines at the end:

      ```
      sleep 10
      mount -a -t nfs
      ```

   e. Restart host2.

      For troubleshooting NFS issues, refer to the RHEL documentation.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000413

8. Set the character encoding of your locale to UTF-8 on host1 and host2.
9. Run the install utility.

   Log in to host1 as root and run the install utility (InfoSphereStreamsSetup.bin). Follow the prompts. When the install utility asks for the file owner, provide the streamsadmin user name and group, as shown in Figure A-19.

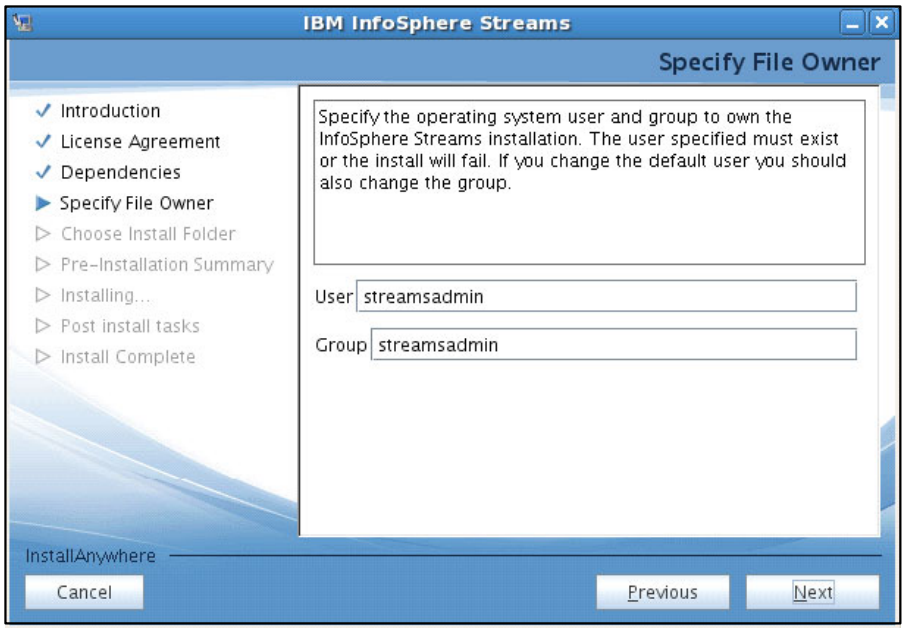


Figure A-19 Specify File Owner window

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000414

Next, you will be prompted for the install location, as shown in Figure A-20.

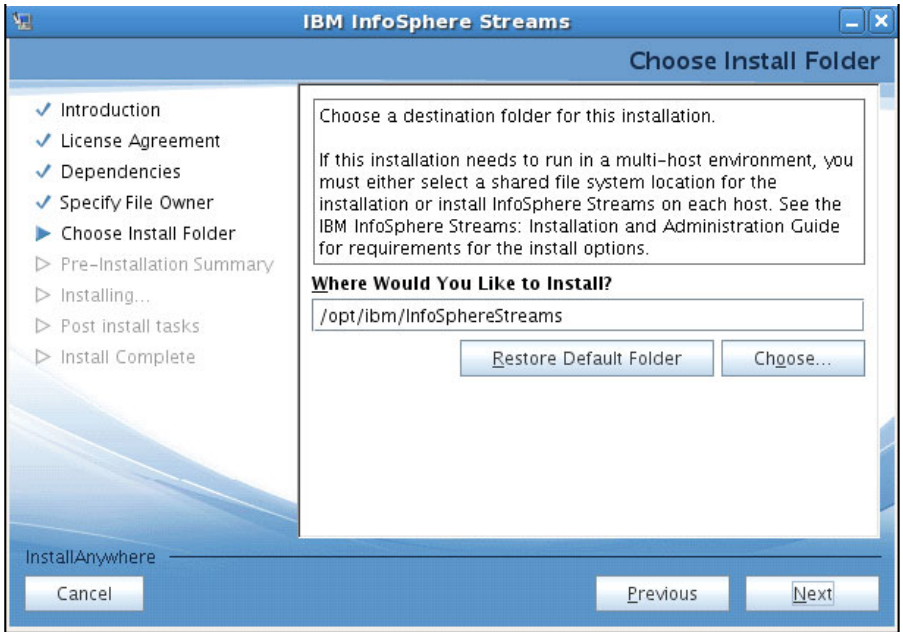


Figure A-20 Install location

Keep the default value as /opt/ibm/InfoSphereStreams and complete the installation.

10. Complete the post-install configuration.

    Configure the Streams environment, the secure shell environment, and RSA authentication by completing steps 15 through 17 of “Installing Streams on a single host” on page 371 on host1 only.

11. Install Streams Studio

    Follow the instructions in steps 18 and 19 of “Installing Streams on a single host” on page 371 to install and run Streams Studio on either host1 or host2.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000415

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000416

B

Toolkits and samples

In this appendix, we introduce one of the strengths of the IBM InfoSphere Streams platform, that is, toolkits and samples. Although these components are main components of the base product, we chose to cover these important assets here in an appendix because of their evolutionary nature. We present a representation and description of the toolkits and samples available with InfoSphere Streams Version 2.0, which is the current release of InfoSphere Streams at the time of the publication of this book. We also describe how you can find the information for the toolkits and samples that are included in that particular release of the software.

In the previous chapters of this book, we discussed the valuable role the toolkits and samples play in enabling the delivery of high quality solutions using data in motion. We touch on that topic here, but also cover the following key questions and concepts:

- What is a toolkit or sample?
- Why do we provide toolkits and samples?
- Where can the toolkits and samples can be found?
- What toolkits and samples are currently available and how do you use them?
- What are the other sources of publicly available assets?

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000417

Overview

In this section, we discuss some of the basic information relative to the Streams toolkits and samples.

What is a toolkit or sample

Simply put, in the context of the InfoSphere Streams platform, toolkits and samples are a collection of assets that facilitate the development of a solution for a particular industry or functionality.

Derived from our experience in delivering streaming applications, these assets may be as simple as common operators and adapters, or as complete as one or more sample applications, or anywhere in between (such as composite operators). In general, samples tend to encompass the less complex or complete of these assets, are composed typically of simple operators/adapters or a simple application flow. Conversely, toolkits tend to have a more complex nature, with one or more complete sample applications. Although this may typically be true, it is by no means a certainty. Development will deploy these helpful assets in the manner that seems most consistent with the way customers can relate to using them, based on our experience.

Why do we provide toolkits or samples

The book has covered at length the newness of this type of analysis for data in motion and analytical programming. Although you may be able to easily relate conceptually to how to apply this new technology to current challenges, or use it to forge into compelling new areas of interest, when it comes time to move from the whiteboard to the keyboard, it may leave you feeling a bit overwhelmed. The primary goal of InfoSphere Streams is not to get you to think about new ways of doing things that can make a difference; it is about enabling you to do those things. This component of the Streams platform is focused on enabling you to achieve that goal.

These assets are provided to give the new InfoSphere Streams developer working examples of what the language can do and how to program a working application to accomplish it. As a developer, for example, you can use these assets for the following:

- As a template to begin developing an application.
- To understand how to program an application in Streams.
- To augment or include functionality into an existing Streams application.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000418

- As an example of what type of applications can be developed in Streams and how they are structured.

These assets do not come with any implicit warranties. They are not guaranteed to provide optimal performance for any specific environment, and performance tuning will likely still be required whether you use these assets alone or in conjunction with an application that is already developed.

Where the toolkits and samples can be found

The samples are packaged with the product and placed in subdirectories under the sample directory of the Streams install directory. (A user typically sets up the $STREAMS_INSTALL environment variable when installing the product.) Each sub-directory will contain multiple samples and their supporting files (such as a makefile). Examples of the subdirectory listings for the samples is shown in the following lists:

- `ls $STREAMS_INSTALL/samples/spl/application`
  - `Compress`
  - `Ping`
  - `Sudoku`
  - `TrendCalculator`
  - `VMStat`
  - `Vwap`
  - `WordCount`
- `ls $STREAMS_INSTALL/samples/spl/feature`
  - `CompositeParameter`
  - `LinuxPipe`
  - `OperatorAndLibrary`
  - `TypedArguments`
  - `Composites`
  - `LoopBack`
  - `RegularExpression`
  - `WindowLibrary`
  - `FanInFanOut`
  - `SampleToolkit`
  - `HostConstraint`
  - `MixedMode`
  - `SchemaSharing`
  - `JavaOperators`
  - `NativeFunction`
  - `TaskParallel`
- `ls $STREAMS_INSTALL/samples/spl/demo`
  - `CommodityPurchasing`

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000419

Most often, there is documentation in the form of comments in the sample files, but it might not be consistent between the samples.

Note: These samples are examples of code to help you understand more about Streams and get started with the software. You should feel free to make a copy of any of them and modify them as desired.

In addition to the SPL Standard Toolkit, in the release of the Streams software available at the time of the writing of this book, there are the following four toolkits:

- Mining Toolkit
- Financial Toolkit
- Database Toolkit
- Internet Toolkit

One of these toolkits is focused on a vertical industry (Financial Markets), while the other three are aligned with a functionality that could be used for many different industries (Data Mining, Internet, and Database). The components of these toolkits are provided to facilitate application development in or around each of these focus areas. However, it is possible to use some of the components in applications for other industries or even to build other functionality. Because of this situation, it is a good idea to review the toolkits and their documentation to see if there is any similarity to the applications you may be developing and use them as an example.

The Financial and Mining toolkits that are available with the product must be separately downloaded from the same site used to download the software. Each is downloaded separately and the Installation Instructions for each is provided with them. There are also instructions in the product documentation. On the other hand, the Internet and Database toolkits are provided as a part of the Streams Product Installation as follows:

- `ls $STREAMS_INSTALL/toolkits`
  - `com.ibm.streams.db`
  - `com.ibm.streams.inet`

As future toolkits become available, they will be available on the software download site where the software is available, for example, Passport Advantage, and should appear in the release notes.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000420

Other publicly available assets

In addition to the samples and toolkits, there is another major source of sample applications and operators that is available to a Streams user called Streams Exchange, which can be found at IBM developerWorks at the following address:

https://www.ibm.com/developerworks/

The Streams Exchange community at IBM developerWorks is a wonderful resource for Streams documentation, discussion forums, sample applications, operator toolkits, and so on. It can be accessed by searching for Streams Exchange at the developerWorks website. Many toolkits and code samples are being added to this resource, often on a weekly basis. At the time of the writing of this book, more than 200 files, sample operators, and toolkits have been added. Most of the content on this community is source code, but you can find items such as useful bookmarks, design patterns, experience reports, and rules of thumb. Use the file search feature in developerWorks to look for content relevant to Streams.

The following is a small sample of the content recently added to the Streams Exchange repository and the list is ever growing:

- HDFS Adapter toolkit
- System T (text analytics) Adapter toolkit
- UIMA Annotate operator
- SPL Examples for Beginners (and tutorial)

Database Toolkit

In this section, we discuss and describe the Database Toolkit. We describe the main mechanism for Streams to connect to other data stores, including but not restricted to solidDB, Netezza, DB2, and Oracle.

Adapters to other technologies will be made available in other resources as and when they are developed.

Streams and databases

Streams applications typically process streams of data flowing from external data sources, and may convert processed streams to external formats to be used by components that are not part of Streams. Much of the data is stored in high-level data-at-rest stores (traditional databases), with higher-lever interfaces than those provided by the standard toolkit (which cover file and network interfaces).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000421

The Database Toolkit is provided to facilitate the development of applications that need to interact with traditional databases.

Streams application may need to interact with such traditional data sources for a variety of reasons. The most common use case is to write the streaming data (after processing) into databases for future use, but there may be other reasons too, such as check-pointing the state of the application. In this scenario, a Streams application may also need to read data from such databases. In other cases, Streams applications may merge data from external repositories with internal streams, enriching their content.

In Chapter 7, “Streams integration approaches” on page 317, there is a section devoted to integration with data stores. The focus of that section is user-centric, and walks a user through the steps needed to set up a database so that a Streams application can interact with it.

Database Toolkit V2.0

The Database Toolkit provides a set of SPL operators that allows easy integration with external data stores. At a higher level, there are three kinds of operators:

- Source
- Append
- Enrich

The Source operators can be thought of as input adapters. They generate an input stream from a database, with a tuple generated for each row in the result of a sequence of SQL SELECT queries. The Append operators can be thought of as output adapters. They store an output stream in a database table, using an SQL INSERT statement. The Enrich operators are not as straightforward as the former two. In some sense, they can be thought of as input adapters, because they do read from a database connection. However, unlike the Source adapters, the Enrich operators do have an input stream, and operate on a tuple-by-tuple basis. For each input tuple received, an Enrich operator executes a sequence of SQL SELECT queries against a database, and generates an output tuple for each row in the result of the queries.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000422

What is included in the Database Toolkit

The toolkit includes the following adapters:

- ODBCSource
- ODBCAppend
- ODBCEnrich
- SolidDBEnrich

ODBCAppend, ODBCSource, and ODBCEnrich implement the preceding operations against a variety of databases (using the Open DataBase Connectivity (ODBC) interface), depending on the environment variable set. In addition, a specialized SolidDBEnrich operator is also provided that achieves much higher performance for the Enrich operation by using a specialized API for SolidDB.

Because the operators in the Database Toolkit need to interact with external software technologies, they need to be configured precisely. This configuration information is specified in an XML document (Connection Specifications Document) that is separate from the SPL application. There are three main reasons for the separation:

- Complexity: The configuration information is complex, and is often specific to a product.
- Interoperability: The same SPL application can work against a variety of databases without having to modify the SPL file significantly. At the same time, many operators from many different applications may want to connect to the same database table.
- Accessibility: The developers who understand the databases are often not the ones who are developing the application, and can access these two components independently.

The operators require a set of environment variables to be defined or set at compile time, which indicate which database is used, and where the header files and libraries for the database product are installed. For the ODBC operators, one of the following environment variables (STREAMS_ADAPTERS_ODBC_*) must be defined (to indicate which database), and the environment variables (STREAMS_ADAPTERS_ODBC_[LIB,INC]PATH) must be set to indicate where the database product is installed. Similar environment variables are needed for the SolidDB operator.

The operators in the Database Toolkit can also be configured with an additional output stream, which produces a tuple for each runtime error it encounters. This tuple contains the error code, the message, and the state returned by the SQL query.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000423

Finally, the Database Toolkit comes with a set of simple sample applications, one for each operator in the toolkit. Each sample includes a template Connections Document that needs to be edited to provide specific information about the database. It also includes two SQL queries (Setup, which is used to create and populate the database table, and Cleanup, which is used to remove the database table). These applications are quite simple in the sense that they just illustrate the operators provided in the toolkit, but getting comfortable with them will go a long way towards building a larger application where interaction with a database is just one part of the whole picture.

Mining Toolkit

In this section, we discuss and describe the Mining Toolkit.

Streams and data mining

Data mining has been a valuable analytical technique for decades. The process involves extracting relevant information or intelligence from large data sets based on such things as patterns of behavior or relationships between data or events to predict, react, or prevent future actions or behavior. Accumulating the amount of data needed to perform meaningful data mining has meant that most data mining has traditionally been performed on stored historical data.

For a certain class of problems, doing data mining analysis on historical data has limited value in being able to take certain specific actions, such as:

- Cyber security (requiring sub-millisecond reaction to potential network threats)
- Fraud detection
- Market trade monitoring
- Law enforcement

The challenge for these types of problems is to enable the techniques and algorithms used in data mining to be applied to real-time data.

The two types of analysis are not mutually exclusive. A combined approach of applying the algorithms employed in traditional mining of data at rest to streaming analysis of data in motion enables both proactive and reactive business intelligence. The Mining Toolkit facilitates the development of applications to use existing data mining analytics to be applied to real-time data streams.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000424

Mining Toolkit V2.0

This toolkit enables scoring of real-time data in a Streams application. The goal is to be able to use the data mining models that you have been using against data at rest to do the same analysis against real-time data. The scoring in the Streams Mining Toolkit assumes, and uses, a predefined model. A variety of model types and scoring algorithms are supported. Models are represented using the Predictive Model Markup Language (PMML) (a standard XML representation) for statistical and data mining models.

The toolkit provides four Streams Processing Language operators to enable scoring, such as:

- Classification
- Regression
- Clustering
- Associations

The toolkit also supports dynamic replacement of the PMML model used by an operator to allow the application to be developed in a way to easily evolve to what is determined by the analysis.

How the Mining Toolkit works

A user starts by building, testing, and training a model of a stored data set using modeling software (such as SPSS, InfoSphere Warehouse, SAS, and so on). After one or more models are created, the user may export models as PMML statements. The user can then incorporate the scoring of real-time data streams against this model into any Streams application through the scoring operators in the Mining Toolkit. Specific operators are compatible with specific types of models. Some of the models supported include:

- For Classification models
  - Decision Tree
  - Logistic Regression
  - Naive Bayes
- For Regression models
  - Linear Regression
  - Polynomial Regression
  - Transform Regression
- For Clustering models
  - Demographic Clustering
  - Kohonen Clustering
- For Associations models
  - Association Rules

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000425

At run time, real-time data is scored against the PMML model. The data streams flow into the scoring operators and the results of the scoring flow out of the operator as a stream. This process is shown in Figure B-1.

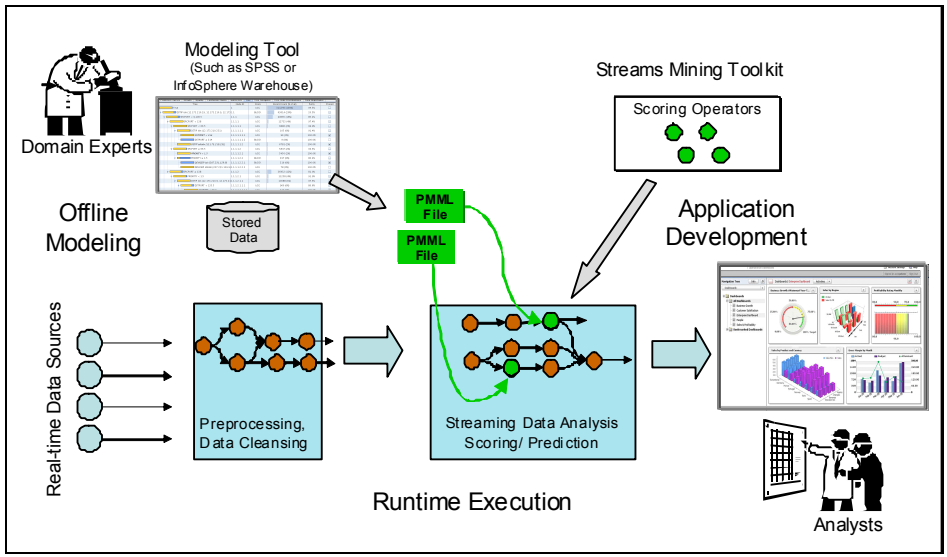


Figure B-1 Process flow of basic use case for Streams Mining Toolkit

By integrating the scoring of models to process data in motion in real time, you can make decisions and take actions in time to make a significant difference. The actions as a result of scoring real-time data will in turn change the data that is stored and used to build future models. This integration allows the analysis to evolve with improved business practices.

Financial Toolkit

In this section, we discuss and describe the Financial Toolkit.

Why use Streams for financial markets

Financial Markets have long struggled with vast volumes of data and the need to make key decisions quickly. To address this situation, automated trading solutions have been discussed and designed in several forms. The challenge is to be able to use information from widely different sources (both from a speed and format perspective) that may hold the definitive keys to making better and profitable trades.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000426

InfoSphere Streams offers a platform that can accommodate the wide variety of source information and deliver decisions at the low latency that automated trading requires.

This combination of being able to use the variety of sources and deliver results in real time is not only attractive in financial markets, but may also provide insight into how to design an application for other industries that have similar needs.

Financial Markets Toolkit V2.0

This toolkit is focused on delivering examples and operators that facilitate the development of applications to provide a competitive advantage to financial industry firms by using InfoSphere Streams. The examples provided demonstrate functionality that should be easy to integrate into their existing environment and reduce the time and effort in developing Streams-based financial domain applications. Our goal is to make it easy to use the unique strengths of InfoSphere Streams (real-time, complex analysis combined with low latency).

Because one of the core needs of Financial Markets is the wide variety of sources, one of the key foci of this toolkit is to provide source adapters for the more common market data feeds. This toolkit also provides adapters for some of the typical market data platforms and general messaging. These adapters make up the base layer of this toolkit's three-layer organization, as shown in Figure B-2.

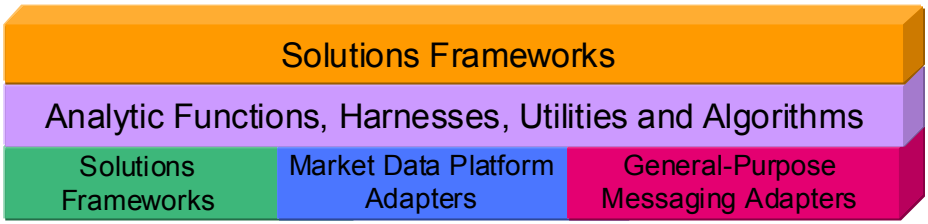


Figure B-2 Financial Toolkit organization

The components in the adapters layer are used by top two layers of the organization and can also be used by applications directly. The functions layer components are used by the top layer and can also be used for applications directly. The components of the top layer represent the Solution Frameworks, starter applications that target a particular use case within the financial markets sector. These are typically modified or extended by developers for a variety of specific needs.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000427

What is included in the Financial Toolkit

The toolkit includes operators to support adapters for Financial Information Exchange (FIX), such as:

- FIXInitiator operator
- FIXAcceptor operator
- FIXMessageToStream operator
- StreamToFixMessage operator

It also supports IBM WebSphere Front Office for Financial Markets (WFO) adapters with the following operators:

- WFOSource operator
- WFOSink operator

For messaging, the toolkit includes an operator to support the IBM WebSphere MQ Low-Latency Messaging (LLM) adapter: MQRMMSink operator.

The toolkit also provides adapters for Simulated Market Feeds (Equity Trades and Quotes, Option Trades) with the following operators:

- EquityMarketTradeFeed operator
- EquityMarketQuoteFeed operator
- OrderExecutor operator

In the Function layer, this toolkit includes analytic functions and operators such as the following:

- Analytical Functions:
  - Coefficient of Correlation
  - The Greeks (Delta, Theta, Rho, Charm, DualDelta, and more)
    - For Put and Call values
    - For general values
- Operators:
  - Wrapping QuantLib financial analytics open source package
  - Provides operators for equity pricing and rating:
    - TrailingPriceStatsCalculator operator (Computes the volume-weighted average price of equities, over a range of the equity's three most-recent trading prices.)
    - OpportunityRater operator (Identifies opportunities.)

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000428

    - VWAPDeltaAgressive operator (This and the following operator examines opportunities and determines whether to generate an order, for different criteria.)
    - VWAPDeltaConservative operator
  - Provides operators to compute theoretical value of an option:
    - EuropeanOptionValue operator (Provides access to 11 different analytic pricing engines, such as Black Scholes, Integral, Finite Differences, Binomial, Monte Carlo, and so on.)
    - AmericanOptionValue operator (Provides access to 11 different analytic pricing engines, such as Barone Adesi Whaley, Bjerksund Stensland, Additive Equiprobabilities, and so on.)

Finally, the Solution Framework layer provides two extensive example applications:

- Equities Trading
- Options Trading

These examples are typically called *white box applications*, because customers can, and typically will, modify and extend them. These example applications are modular in design with pluggable and replaceable components that you can extend or replace. This modular design allows these applications to demonstrate how trading strategies may be swapped out at run time, without stopping the rest of the application.

The Equities Trading starter application includes:

- TradingStrategy module: Looks for opportunities that have specific quality values and trends.
- OpportunityFinder module: Looks for opportunities and computes quality metrics.
- SimpleVWAPCalculator module: Computes a running volume-weighted average price metric.

The Options Trading starter application includes:

- DataSources module: Consumes incoming data and formats and maps for later use.
- Pricing module: Computes theoretical put and call values.
- Decision module: Matches theoretical values against incoming market values to identify buying opportunities.

These starter applications give the application developer good examples and a great foundation to start building their own applications.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000429

Internet Toolkit

In this section, we discuss and describe the Internet Toolkit. We present the main mechanism for Streams applications to connect to the Internet using the standard HTTP request and response connections. This mechanism allows Streams applications to connect to HTTP(S), FTP(S), RSS, and file sources. Certain feeds (such as Twitter, for example) do not use this mechanism, and adapters to these feeds will be made available as and when they are developed.

Streams and Internet sources

Streams applications may need to process streams of data flowing from external data sources that reside on the Internet, and are made available using the standard HTTP/FTP mechanism. For RSS data feeds, the generated stream is the RSS XML data. The Internet toolkit facilitates the interaction of Streams applications with Internet sources.

Furthermore, the application developer may be interested not just in reading the source once, but continually obtain updates from the source. In other cases, the source may be an RSS feed to which the application subscribes. Note that this mechanism could be used to read from files, especially if you are interested in updates to the file.

Internet Toolkit V2.0

The Internet toolkit provides the InetSource operator that receives text-based data from remote sources using HTTP, and generates an input stream from this content. InetSource generalizes the functionality of the File/TCP Source operators provided in the SPL Standard toolkit in two main ways:

- Multiple input sources: The InetSource operator can take, as input, a URI list, which allows it to access all the sources listed in sequential order (as opposed to File/TCP Source that can read only from one source).
- Incremental updates: The InetSource operator can be configured to periodically check for, and fetch updates from, the sources in its URI list, and stream either the whole content or just the updates. Not only that, it can be configured to not stream the initial contents. It can also be configured to stream the whole content periodically even if there are no updates.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000430

Furthermore, the InetSource operator is significantly more configurable than the TCP/File Source operators. For example, the operator can be configured to treat “k” lines of input as a given record, where “k” is a parameter. It can be configured to emit tuples per record, per URI, or per fetch. Not only does this make it a natural interface to connect to remote sources, but in fact can be used to read from local files if the additional capabilities are required.

What is included in the Internet Toolkit

The Internet toolkit contains the InetSource operator described above, and a sample application that uses this operator. Although this sample application is simple (just an InetSource operator followed by a Custom operator that prints the stream), it show some of the parameters to the InetSource operator. Also, the included Makefile shows how to compile the Internet Toolkit along with an SPL application.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000431

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000432

C

Additional material

This book refers to additional material that can be downloaded from the Internet as described in the following sections.

Locating the web material

The web material associated with this book is available in softcopy on the Internet from the IBM Redbooks web server. Point your web browser at:

`ftp://www.redbooks.ibm.com/redbooks/SG247970`

Alternatively, you can go to the IBM Redbooks website at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247970.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000433

Using the web material

The additional web material that accompanies this book includes the following files:

File name	*Description*
baskrule.zip	Compressed code sample files. It includes the `baskrule/MyOp/MyOp.xml`, `baskrule/MyOp/MyOp_h.cgt`, and `baskrule/MyOp/MyOp_cpp.cgt` files.

Downloading and extracting the web material

Create a subdirectory (folder) on your workstation, and extract the contents of the web material .zip file into this folder.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000434

Glossary

access control list (ACL). The list of principals that have explicit permission (to publish, to subscribe to, and to request persistent delivery of a publication message) against a topic in the topic tree. The ACLs define the implementation of topic-based security.

analytic. An application or capability that performs some analysis on a set of data.

application programming interface. An interface provided by a software product that enables programs to request services.

asynchronous messaging. A method of communication between programs in which a program places a message on a message queue, then proceeds with its own processing without waiting for a reply to its message.

computer. A device that accepts information (in the form of digitalized data) and manipulates it for some result based on a program or sequence of instructions about how the data is to be processed.

configuration. The collection of brokers, their execu**tion groups, the message flows and sets that are assigned to the**m, and the topics and associated access control specifications.

data mining. A mode of data analysis that focuses on the discovery of new information, such as unknown facts, data relationships, or data patterns.

deploy. Make operational the configuration and topology of the broker domain.

engine. A program that performs a core or essential function for other programs. A database engine performs database functions on behalf of the database user programs.

instance. A particular realization of a computer process. Relative to database, the realization of a complete database environment.

metadata. Typically called data (or information) about data. It describes or defines data elements.

multitasking. Operating system capability that allows multiple tasks to run concurrently, taking turns using the resources of the computer.

multithreading. Operating system capability that enables multiple concurrent users to use the same program. This reduces the impact of initiating the program multiple times.

optimization. The capability to enable a process to execute and perform in such a way as to maximize performance, minimize resource utilization, and minimize the process execution response time delivered to the user.

pool. Set of hosts used on colocation and exlocation constraints.

process. An instance of a program running in a computer.

program. A specific set of ordered operations for a computer to perform.

roll-up. Iterative analysis, exploring facts at a higher level of summarization.

sample. A sample is an SPL application that is made available with the product. It may be part of the product, or made available with a toolkit. A sample is usually simple, and meant to illustrate a single feature or functionality.

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000435

server. A computer program that provides services to other computer programs (and their users) in the same or other computers. However, the computer that a server program runs in is also frequently referred to as a server.

SPL application. The term SPL application refers to an SPL Main composite operator. When compiled, a Main composite operator may be executed as a distributed or stand-alone application. An SPL source file may have zero or more Main composites.

SPL application set project. The term SPL application set refers to a project that references one or more SPL applications or SPL mixed-mode applications that are developed or executed together.

SPL mixed mode source file. An SPL mixed-mode source file contains a mix of Perl and SPL code. When pre-processed, the SPL mixed-mode source file yields an SPL source file. SPL mixed-mode source files always has a file extension of .splmm.

SPL project. The term SPL project refers to an SPL project. An SPL project can contain an SPL application, which has an .spl file extension, an SPL mixed-mode application, which has an .splmm file extension, SPL native functions, SPL primitive operators, and more.

To use Streams Studio, you need to create a new SPL project or import an existing SPL toolkit or application. When you import an existing SPL toolkit or application, Streams Studio creates an SPL project for you.

SPL source file. An SPL source file contains SPL code. The code in a source file implements SPL functions and composite operators. SPL source files always have a file extension of .spl.

stand-alone application. A stand-alone application refers to a stand-alone executable file that has been compiled so that it can be run without a Streams instance.

A stand-alone application runs as an executable and does not use the Streams runtime or job management facilities. This can be helpful when you are testing or debugging applications. The SPL compiler generates this type of executable file when you set the Executable type option on the SPL Compiler Options preference page to Standalone application. Applications that run on a Streams instance are called distributed applications.

stream. A stream is an infinite sequence of tuples, and each time an operator invocation receives a tuple on one of its input streams, it fires, producing some number of tuples on its output streams.

task. The basic unit of programming that an operating system controls. Also see multitasking.

thread. The placeholder information associated with a single use of a program that can handle multiple concurrent users. Also see multithreading.

toolkit. A toolkit is a set of SPL artifacts, organized into a package. The main toolkit purpose of a toolkit is to make functions (SPL or native) and operators (primitive or composite) reusable across different applications. A toolkit provides one or more namespaces, which contain the functions and operators that are packaged as part of the toolkit, and makes them available to use in applications that have access to the toolkit.

toolkit model editor. The term toolkit model editor refers to one of the following editors of toolkit model documents:

- Info model editor
- Operator model editor
- Function model editor

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000436

white box applications. Sample applications that are modular in design with easily pluggable and replaceable components so that they can easily be modified and extended. Sample applications made available with the Streams toolkits are written as white box applications, for easy extensibility.

zettabyte. A trillion gigabytes.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000437

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000438

Abbreviations and acronyms

AAS	Authentication and Authorization Service
CFG	configuration
CORBA	Common Object Request Broker Architecture
CPU	central processing unit
DDY	Distributed Distillery
DFE	Distributed Front End
DFS	data fabric server
DGM	dataflow graph manager
DIG	dense information grinding
DNA	Distillery instance Node Agent
DPFS	General Parallel File System
DPS	distributed processing system
DSF	Distillery Services Framework
DSM	data source manager
DSS	Distillery Semantic Services
DST	Distillery Base API
EVT	event service
Gb	gigabits
GB	gigabytes
GUI	graphical user interface
I/O	input/output
IBM	International Business Machines Corporation
IDE	Integrated Development Environment
IIDE	Distillery instance Node Agent
IKM	Information Knowledge Management
INQ	inquiry services
ITSO	International Technical Support Organization
JDBC	Java DataBase Connectivity
JDK	Java Development Kit
JDL	Job Description Language
JE	Java Edition
JMC	job manager controller
JMN	job manager
LAS	logging and auditing subsystem
LDAP	Lightweight Directory Access Protocol
Mb	megabits
MB	megabytes
MNC	master node controller
NAM	naming
NAN	nanoscheduler
NC	node controller
NDA	network data analysis
ODBC	open database connectivity
OPT	optimizer
ORA	Ontology and Reference Application
OS	Operating System
PE	Processing Element
PEC	Processing Element Container
PHI	physical infrastructure
PRV	privacy
IBM RAA®	Resource Adaptive Analysis
REF	reference application
REX	results/evidence explorer
RFL	resource function learner
RMN	resource manager
RPS	repository

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000439

RTAP	real-time analytic processing
SAGG	statistics aggregator
SAM	Streams Application Manager
SCH	Scheduler
SDO	stream data object
SEC	security
SMP	symmetric multiprocessing
SOA	service-oriented architecture
SODA	Scheduler
SPADE	Streams Processing Application Declarative Engine
SPC	Streams Processing Core
SRM	Streams Resource Manager
STG	storage
Streams	IBM InfoSphere Streams
SWS	Streams Web Service
SYS	system management
TAF	text analysis framework
TEG	traffic engineering
TCP	transmission control program
TCP/IP	transmission control program /internet protocol
TCP/UDP	transmission control program /user datagram protocol
TRC	tracing (debug)
TSM	type system & metadata
UBOP	user-defined built-in operator
UDF	user-defined function
UDOP	user-defined operator
UE	user experience
UTL	utilities
WLG	workload generator
URI	uniform resource identifier
URL	universal record locator
WWW	World Wide Web
XML	eXtensible Markup Language

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000440

Related publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this book.

IBM Redbooks

The following IBM Redbooks publication provides additional information about the topic in this document. Note that some publications referenced in this list might be available in softcopy only.

- *IBM InfoSphere Streams Harnessing Data in Motion*, SG24-7865

You can search for, view, download or order these documents and other Redbooks, Redpapers, Web Docs, draft and additional materials, at the following website:

ibm.com/redbooks

Other publications

This publication is also relevant as a further information source:

- Kernighan, et al., *The C Programming Language* , Prentice Hall, 1978, ISBN 0131103628

Online resources

These websites are also relevant as further information sources:

- *IBM InfoSphere Streams Version 2.0.0.2 Database Toolkit*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/topic/com.ibm.swg.im.infosphere.streams.product.doc/doc/IBMInfoSphereStreams-DatabaseToolkit.pdf

- *IBM SPSS Modeler 14.2 Solution Publisher*

  ftp://ftp.software.ibm.com/software/analytics/spss/documentation/modeler/14.2/en/SolutionPublisher.pdf

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000441

- LOFAR Outrigger in Scandinavia project

  http://www.lois-space.net/index.html

- The Smarter Computing Blog

  http://www.smartercomputingblog.com/2011/08/22/big-data-the-next-phase-of-the-information-revolution/

- *SPL Code Generation Perl APIs*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.doxygen.codegen.doc%2Fdoc%2Findex.html

- *SPL Language Specifications*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_5

- *SPL Operator Model Reference*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_4

- *SPL Runtime C++ APIs*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?topic=%2Fcom.ibm.swg.im.infosphere.streams.doxygen.runtime.doc%2Fdoc%2Findex.html

- *SPL Toolkit Development Reference*

  http://publib.boulder.ibm.com/infocenter/streams/v2r0/index.jsp?nav=%2F3_1_8

- University of Ontario Institute of Technology Neonatal research project

  http://research.uoit.ca/assets/Default/Publications/Research%20Viewbook.pdf

IBM education support

In this section, we discuss IBM education support for InfoSphere Streams.

IBM training

Available from IBM training are the newest offerings to support your training needs, enhance your skills, and boost your success with IBM software. IBM offers a complete portfolio of training options, including traditional classroom, private onsite, and eLearning courses.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000442

Many of our classroom courses are part of the IBM “Guaranteed to run program”, ensuring that your course will never be canceled. We have a robust eLearning portfolio, including Instructor-Led Online (ILO) courses, Self Paced Virtual courses (SPVC), and traditional Web-Based Training (WBT) courses. A perfect complement to classroom training, our eLearning portfolio offers something for every need and every budget; simply select the style that suits you.

Be sure to take advantage of our custom training plans to map your path to acquiring skills. Enjoy further savings when you purchase training at a discount with an IBM Education Pack (online account), which is a flexible and convenient way to pay, track, and manage your education expenses online.

The key education resources listed inTable 8 have been updated to reflect InfoSphere Streams. Check your local Information Management Training website or chat with your training representative for the most recent training schedule.

Table 8 Education resources

Course title	Classroom	Instructor-Led	Self Paced Virtual Classroom	Web-Based Training
Programming for InfoSphere Streams	DW722	3W722	2W722	1W722

Descriptions of courses for IT professionals and managers are available at the following address:

`http://www.ibm.com/services/learning/ites.wss/tp/en?pageType=tp_search`

Visit `http://www.ibm.com/training` or call IBM training at 800-IBM-TEACH (426-8322) for scheduling and enrollment.

Information Management Software Services

When implementing an Information Management solution, it is critical to have an experienced team involved to ensure that you achieve the results you want through a proven, low risk delivery approach. The Information Management Software Services team has the capabilities to meet your needs, and is ready to deliver your Information Management solution in an efficient and cost-effective manner to accelerate your return on investment (ROI).

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000443

The Information Management Software Services team offers a broad range of planning, custom education, design engineering, implementation, and solution support services. Our consultants have deep technical knowledge, industry skills, and delivery experience from thousands of engagements worldwide. With each engagement, our objective is to provide you with a reduced risk and expedient means of achieving your project goals. Through repeatable services offerings, capabilities, and best practices leveraging our proven methodologies for delivery, our team has been able to achieve these objectives and has demonstrated repeated success on a global basis.

The key services resources listed in Table 9 are available for InfoSphere Streams.

Table 9 Services resources

Information Management Software Services offering	Short description
InfoSphere Streams Delivery Capability: http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/services/InfoSphere_Streams_Delivery_Capabilities.pdf	Our Information Management (IM) Software Services team has a number of capabilities that can support you with your deployment of our IBM InfoSphere Streams Solution, including: ► Installation and configuration support ► Getting acquainted with IBM InfoSphere Streams ► Mentored pilot design ► Streams programming language education ► Supplemental Streams programming support

For more information, visit our website at the following address:

http://www.ibm.com/software/data/services

IBM Software Accelerated Value Program

The IBM Software Accelerated Value program provides support assistance for issues that fall outside normal “break-fix” parameters addressed by the standard IBM support contract, offering customers a proactive approach to support management and issue resolution assistance through assigned senior IBM support experts who know your software and understand your business needs. Benefits of the Software Accelerated Value Program include:

- Priority access to assistance and information
- Assigned support resources
- Fewer issues and faster issue resolution times
- Improved availability of mission-critical systems

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000444

- Problem avoidance through managed planning
- Quicker deployments
- Optimized use of in-house support staff

To learn more about IBM Software Accelerated Value Program, visit our website at the following address:

http://www.ibm.com/software/data/support/acceleratedvalue/

To talk to an expert, contact your local Accelerated Value Sales Representative at the following address:

http://www.ibm.com/software/support/acceleratedvalue/contactus.html

Protect your software investment

To protect your software investment, ensure you renew your Software Subscription and Support. Complementing your software purchases, Software Subscription and Support gets you access to our world-class support community and product upgrades with every new license. Extend the value of your software solutions and transform your business to be smarter, more innovative, and cost-effective when you renew your Software Subscription and Support. Staying on top of on-time renewals ensures that you maintain uninterrupted access to innovative solutions that can make a real difference to your company's bottom line.

To learn more, visit our website at the following address:

http://www.ibm.com/software/data/support/subscriptionandsupport

How to get Redbooks

You can search for, view, or download Redbooks, Redpapers, Technotes, draft publications and Additional materials, as well as order hardcopy Redbooks publications, at this website:

ibm.com/redbooks

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000445

Help from IBM

IBM Support and downloads

ibm.com/support

IBM Global Services

ibm.com/services

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000446

Index

Symbols
_h.cgt file 337, 340, 353

A
Absolute host location 187
Access Control Lists (ACLs) 145
action flags 357
Active Diagnostics 25
Adapter operators 221
Admin Group 155, 157
Aggregate operator 92, 102–104, 119, 234–235, 315
AllData stream 102–104
allowAny element 289, 297, 308
AmericanOptionValue operator 409
Apache Hadoop 318
API calls 78
Application Graph 45–46, 62, 64
application host services 148–149
Application parameterization 248, 251
applications, DYNAMIC INSIGHT ASSEMBLY LINE 68
applications, MORE CONTEXT 68
applications, RIGHT NOW 68
Associations models 405
Authentication and Authorization Service 42, 142, 199

B
Barrier operator 241–242
baskrule.xml file 333
Beacon operator 95, 100, 133, 135, 259, 279, 295
BEP 25
Big Data xi, xiii, 7, 17, 317–318, 325
BigSchema stream 88
botnets 30
build host 167
Business Activity Monitoring 25
Business Event Processing 25
business intelligence xiii, 404
Business Logic Derived Events 25
Business Service Management 25

C
C/C++ language debuggers 269
C++ Primitive operators 277, 286, 304
Call Detail Record 28–29, 68, 320
capturestate command 134
CDR 28, 68, 320
CEP 25–26
CEP tools 26
Classification models 405
clemrtl_setAlternativeOuput call 343
Clustering models 405
Command and Control (C&C) machines 30
Complex Event Processing 25–26
Composite operator 83, 86, 88, 93, 97–98, 100, 102–104, 109, 111, 113, 117–119, 121, 125, 127, 132–133
Composite types 209–210
ComputeRevenueSnapshot operator 92, 95
connection.xml 107–108
Custom operator 80, 111, 114, 117, 125, 133, 135, 137, 279, 287, 295, 411

D
Data Analyst 326–328, 333–336, 338–339, 343–344
Data Parallel pattern 80
data types 27, 46, 49, 53–54, 69, 76, 208–210, 325
data warehousing xiii, xvi, 15, 19
database xi, xiii, xv, 19, 21, 35–40, 44, 69, 72–74, 76, 80, 107–108, 110, 140, 142, 149, 157, 194, 199–200, 210, 317, 328, 355–367, 402–404
Database Toolkit 72, 76, 80, 107, 355–357, 400–404
debugging a Streams application 260, 269
declarative commands 53
DirectoryScan operator 230
DYNAMIC INSIGHT ASSEMBLY LINE applications 68
DynamicExporter 128, 130, 132–133
DynamicImporter 121, 123–126

E
Eclipse IDE framework 154

© Copyright IBM Corp. 2011. All rights reserved.

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000447

Eclipse installation location 153–154
Eclipse SDK 370, 383
Enrich Stream pattern 80
environment checklist 152–153
environment variables 155, 164–165, 176, 355–356, 358–366, 399, 403
EquityMarketQuoteFeed operator 408
EquityMarketTradeFeed operator 408
EuropeanOptionValue operator 409
execution host 167
Export operator 115, 127, 130, 132, 182–183, 255, 258
ExportController Composite operator 133
Expression mode 314

F

FileSink operator 75, 110, 125, 346, 352
FileSource operator 53, 55, 85, 119, 227, 230, 253, 268, 271, 314, 346
Filter operator 83, 312–315
FilterCommon.pm 313
Final punctuations 303
FIXAcceptor operator 408
Functor 305
Functor operator 51, 54–55, 86–88, 98–100, 104–105, 122, 125, 133, 183, 189–190, 221–222, 232–234, 239–240, 249, 269
fusing 180, 209

G

g (go) command 273
Gate operator 111, 113–115
Generic and Non-generic Primitive operators 191
genresWritten attribute 238
getCustomMetricValue 135
getInputPortMetricValue 135–136
getOutputPortMetricValue 135
GlobalCalls 205–207, 211–212
GPS data 31

H

Hadoop Distributed File Systems 318
hasDelayField parameter 227, 229
HDFS sink 76
HDFSDirectoryScan operator 325
HDFSSource operator 325
High Volume Stream Analytics 25
Home Directory 153–156
Host Controller 43, 141–143, 149, 181, 197
host pools 187, 209
host tags 167, 172, 184–185
hostColocation placement 188
hostExlocation placement 188
hostIsolation placement 188

I

IBM General Parallel File System 143, 158, 168
IBM InfoSphere BigInsights 17, 318
IBM InfoSphere Streams xi, xiii, xv–xvi, 1, 17, 33–34, 36–39, 44–45, 47–48, 56, 58–59, 61, 63–65, 69, 107, 134, 160, 173, 269, 317–318, 369, 397, 420
IBM InfoSphere Streams Studio 20, 22, 45, 58–59, 61–65, 206, 388
IBM SPSS Modeler 317, 325–326, 328, 335
IBM Streams Processing Language 20
IBM WebSphere Front Office 21
IBM WebSphere Front Office for Financial Markets (WFO) adapters 408
IDE xiii, 22, 154, 162, 164, 383, 386, 388, 390
Import operator 108, 110, 120–121, 124, 182–183, 256, 258
ImportController 121, 123, 126
ImportControllerComposite operator 125
Importer Composite operator 125, 133
STREAMS_ADAPTERS_ODBC_ 356, 359
Information-Derived Events 25
InfoSphere Streams operator 325
InfrastructureMonitoring 205, 207
initDelay= parameter 230
initialization timing delay 230
Input ports 82, 86, 91, 97, 102, 108, 111
input source node key name 333
InputStream stream 113
instance ID 144, 156, 176
Instance Shared Space 156–157
Integrated Development Environment 22
Integrated Development Environment (IDE) 22
Integration and Test instances 146

J

J2EE compliant application servers 39
JAVA_HOME environment variable 164
Join operator 56–57, 73, 102–104, 192, 241–242, 244, 285
JSONToTuple operator 324

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000448

K
key-mapping file 236, 238

L
LD_LIBRARY_PATH 346–347, 349, 356, 361
STREAMS_ADAPTERS_ODBC_ 403
Lightweight Directory Access Protocol 158
Load shedding 135
log files 147
loops 248, 252, 302

M
Main.spl editor 60
Make Instance menu 178
Marketing xiii
medical analysis 31
Metric View 64
MORE CONTEXT applications 68
MultiFileWriter 117–119
Multi-Host Reserved Topology 148–149
Multi-Host Unreserved Topology 149–150

N
Name Service 43, 156, 158
NameServiceUrl property 200
Network File System 143, 392

O
ODBC connection 76
ODBCAppend operator 76, 357, 365
ODBCEnrich operator 107–108, 110
ODBCINI 358, 360–365, 367
online analytic processing (OLAP) 15, 19
online transaction processing (OLTP) 14–15
Open Source Eclipse 22
Operator Custom Logic 191
Operator prepareToShutdown method 345
operator windows 191
OpportunityRater operator 408
OPRA data feeds 30
optimizing compiler 23
OrderExecutor operator 408
Outlier Detection pattern 80
OutlierIdentifier operator 102
Outliers stream 104
output adapters 21
Output ports 82, 86, 92, 97, 102, 108, 111

P
parallel pipelines 193
PATH 356, 359, 403
PE 97, 111, 142–143, 181, 189, 199, 209, 262, 267–268, 352, 355, 359
PEC 43, 47, 143, 181
PEs 43, 47, 97, 111, 134, 141, 143, 148, 180–181, 184, 189, 193–198, 348, 366
pinToCpu function 279–282
pipelining 26, 193
Pluggable Authentication Modules 158
Predictive model 327
Predictive Processing 25
preprocessors 248
Primary (Management) Host / Server 157
Primitive operator 80, 121, 125–126, 128, 132, 134, 277, 284, 305
Primitive operators 78, 191
Primitive types 209
Private instances 145
Processing Element Container 43, 47, 143
Processing Elements 43, 47, 267–268
Processing Language files 206
Processing Multiplexed Stream 81
Production instances 146
Project Explorer 60–61
Punctor operator 221, 234, 240
punctuation markers 212, 231
punctuation modes 302

R
Reactive-analytics 135
real-time analytic processing 1, 15
real-time predictions 31
Recovery database 142
Red Hat Enterprise Linux 59, 160–161, 370
Redbooks website 421, 425
  Contact us xvii
Regression models 405
Relational operators 221
relative operator placement 188
RevenueSnapshots 95–96
RevenueTot 92–95
RIGHT NOW applications 68
RSA authentication 159, 169–170, 383, 395
RSA public keys 159
RTAP 1, 15–16, 19
runtime configuration 23

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000449

S

SAM 42, 141–143, 148, 180, 194, 197–201
SCH 42, 142, 148, 194, 200
Scheduler 42–43, 69, 141–142, 180, 184
schema definitions 53
Scoring Model operator 346
Secondary (Application) Hosts / Servers 157
Secure Shell 169
Security Public Key Directory 156, 159
SecurityPublicKeyDirectory 158–159, 170, 173–174
SELECT statement 108, 110
SelectTransactions 82–83
setAlternativeInput call 341
Shared instances 145
Shared Space 156–157
SimpleSchemaReducer 86–87, 89
Single Host Topology 147–148
Sink operator 46, 58, 76, 204, 231, 255, 261, 271, 279, 323
Sink SPL statement 231
sliding window 57, 217–220, 235, 243, 246
SmallSchema stream 88
SmallSchemaType 86–87, 89
Smarter Planet 10, 12–14, 17, 19, 27
Solid Accelerator API 21
solidDB with solidDBEnrich operator 365
SolidDBEnrich operator 403
solidDBEnrich operator 356–357, 365
SOLIDDIR 364–365
Solution Publisher API interface 336
Sort operator 245–246
source and output adapters 21
Source Editor View 45
Source operator 45–46, 54, 134, 204, 227, 229, 231, 255, 257, 279, 286, 290, 294, 298
SourceGen Composite operator 125, 132
SPL xiv, 39–40, 45, 47, 53–56, 59–62, 78, 80–83, 86–88, 96, 100, 107, 115–116, 121, 125, 127, 134–135, 138, 143–144, 167, 184, 187, 189, 194, 206, 208–211, 219–220, 224, 227, 229, 231, 247–248, 250–253, 255, 259, 261–262, 269, 275, 277–286, 288–293, 295, 297–298, 301–302, 304–305, 308–313, 315–316, 325, 327, 346–347, 349, 354–356, 359, 361–365, 369, 388–391, 400–403, 410–411
SPL functions 278
Split operator 92, 95, 236, 239, 253
SQL SELECT statements 40
SRM 42, 142–143, 148, 194, 197, 199–200
Streams xi–xvi, 1, 17–34, 36–48, 50, 53–72, 74–82, 86, 88, 96, 101, 107, 110–111, 115, 121, 127, 134, 137, 139–147, 149, 151–173, 175–177, 179–182, 184, 186–187, 191, 194–200, 203–207, 209–213, 215–216, 220–221, 227, 229–230, 234, 243, 247–248, 250–253, 255, 257–262, 264–275, 277, 281, 284, 292, 299, 302, 317–328, 331–333, 335–337, 346–348, 353–357, 364, 366–367, 369–372, 374, 380, 382–383, 387–388, 390–393, 395, 397–402, 404–407, 410
Streams Admin Home Directory 154, 156
Streams application 23, 33, 38–40, 42–48, 50, 53, 55–56, 59–64, 67, 69, 75–78, 139, 147, 149, 179, 204, 252, 255, 260–263, 266–267, 269–274, 317, 325, 328, 346, 355–357, 398, 402, 405
Streams Application Developer 326–328, 346
Streams Application Manager 42, 141, 199
Streams Available Hosts 153, 155
Streams Component Developer 327–328, 333–334, 336
Streams data flow 204
Streams Debugger 22, 269–275
Streams Installation Hosts 153, 155
Streams installation location 153–154
Streams instance configuration checklist 152, 155
Streams instance configuration files 146, 154, 156
Streams instance ID 144, 156
Streams instance log files 147
Streams instance user authentication files 147
Streams instances 40–42, 58, 139–140, 143–144, 151, 159, 166, 168, 267, 391
Streams Job 47, 267–268
Streams Live Graph 20, 22, 223, 263–266
Streams Metric View 64
Streams Name Service URL 156, 158
Streams operators 55, 57, 70–71, 191, 212
  Sink 46, 58
Streams Processing Language 20, 39, 45, 53, 55, 59, 143, 203, 205–206, 212, 220, 259, 275, 277, 327
Streams Processing Language operators 220, 405
Streams Processing Language statements 60
Streams Resource Manager 42, 180, 199
Streams RSA key 169
Streams Runtime 20, 22–23, 40, 42, 47
Streams Scheduler 69
Streams Software Group 153–154, 163
Streams Software Owner 153, 156, 163, 170
Streams Studio IDE (Eclipse) command 164

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000450

Streams User Group 155, 157
Streams Web Service 43, 142
Streams Web Service (SWS) HTTPS port 175
STREAMS_ADAPTERS_ODBC_MYSQL 362
STREAMS_ADAPTERS_ODBC_NETEZZA 363
STREAMS_ADAPTERS_ODBC_ORACLE 361
STREAMS_ADAPTERS_ODBC_SOLID 364
STREAMS_ADAPTERS_ODBC_SQLSERVER 362
STREAMS_ADAPTERS_ODBC_UNIX_OTHER 358
STREAMS_DEFAULT_IID environment 176
streamsbasketrule.str file 333
streamsprofile.sh command 164–165
streamtool 58, 64–65, 134, 137, 149, 158, 168–169, 174, 176, 185, 188, 197–198, 200–201, 223, 266–268, 356, 366–367, 383
streamtool addservice command 201
streamtool canceljob command 147
streamtool checkhost command 161
streamtool chhost command 185
streamtool genkey command 158, 170, 173
streamtool getresourcestate command 174, 197–198, 200
streamtool lshosts command 177
streamtool lsinstance command 268
streamtool man command 143
streamtool man mkinstance command 173
streamtool mkhosttag command 185
streamtool mkinstance command 166, 171, 173, 177
streamtool restartpe command 198
streamtool restartservice command 201
streamtool stopPE command 195
streamtool stoppe command 198
streamtool submitjob command 252
submit() API call 294, 300
SWS 43, 65, 142, 148, 175, 199–200
SWS service 65
Sys.Wait parameter 98

T

tagged pool 186
TCPSource operator 195, 228
terminal node key name 333
Throttle operator 80, 115, 137, 261, 271
TickerFilter operator 83–85
tickerSymbol 82–85
TrailingPriceStatsCalculator operator 408
TransactionDataSchemaMapper 88, 90
TransactionRecordOutput 87, 89
TransactionRecordOutput type 88
Transactions 82–83
Transform operator 54, 305–306, 308–310, 315
Transportation 31
tumbling window 57, 217, 240, 243, 245
TupldPacer 113
Tuple History 191
tuple inter-arrival timing 229
Tuple Pacer pattern 111
TuplePacer 111–113, 115
TuplePacer operator 115
TuplePacerMain 115
TupleToJSON operator 324

U

U.S. National Oceanic & Atmospheric Administration (NOAA) 21
UDPSink 76, 221, 231
Union operator 92, 95
unixODBC driver 356
User Group 155, 157
User Shared Space 156–157
Utility operators 221

V

VMStatReader operator 286, 288–289, 291–292
volume weighted average price 116
VWAPDeltaAgressive operator 409
VWAPDeltaConservative operator 409

W

WFOSink operator 408
WFOSource operator 408
windows 56–58, 192, 212–218, 223, 234–235, 241, 243–247, 297, 308
  eviction policy 216, 218
  sliding 57–58, 213, 215, 219
  trigger policy 214, 218, 243
  tumbling 57, 213, 217, 245

Y

yum install command 374

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000451

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000452

IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution

(0.5" spine)
0.475"<->0.875"
250 <-> 459 pages

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000453

IBM InfoSphere Streams

Assembling Continuous Insight in the Information Revolution



Supporting scalability and dynamic adaptability

Performing real-time analytics on Big Data

Enabling continuous analysis of data

In this IBM Redbooks publication, we discuss and describe the positioning, functions, capabilities, and advanced programming techniques for IBM InfoSphere Streams, a new paradigm and key component of IBM Big Data platform. Data has traditionally been stored in files or databases, and then analyzed by queries and applications. With stream computing, analysis is performed moment by moment as the data is in motion. In fact, the data might never be stored (perhaps only the analytic results). The ability to analyze data in motion is called real-time analytic processing (RTAP).

IBM InfoSphere Streams takes a fundamentally different approach to Big Data analytics and differentiates itself with its distributed runtime platform, programming model, and tools for developing and debugging analytic applications that have a high volume and variety of data types. Using in-memory techniques and analyzing record by record enables high velocity. Volume, variety and velocity are the key attributes of Big Data. The data streams that are consumable by IBM InfoSphere Streams can originate from sensors, cameras, news feeds, stock tickers, and a variety of other sources, including traditional databases. It provides an execution platform and services for applications that ingest, filter, analyze, and correlate potentially massive volumes of continuous data streams.

This book is intended for professionals that require an understanding of how to process high volumes of streaming data or need information about how to implement systems to satisfy those requirements.

INTERNATIONAL TECHNICAL SUPPORT ORGANIZATION

BUILDING TECHNICAL INFORMATION BASED ON PRACTICAL EXPERIENCE

IBM Redbooks are developed by the IBM International Technical Support Organization. Experts from IBM, Customers and Partners from around the world create timely technical information based on realistic scenarios. Specific recommendations are provided to help you implement IT solutions more effectively in your environment.

For more information:
ibm.com/redbooks

SG24-7970-00 ISBN 0738436151

Amazon.com, Inc., Amazon.com Services LLC - Ex. 1004, Page 000456