

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.sp1

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



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 MCPEImpl.cpp:runDebugServices:710] - Started debug services...
30 Jun 2011 12:48:41.352 [3768] INFO spl_pe MCPEImpl.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 MCPProbePointServices.cpp:run:911] - 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



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`



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 MCEImpl.cpp:runDebugServices:711] - Started debug services...
06 Sep 2011 13:41:55.226 [13979] INFO spl_app M[ProbePointServices.cpp:run:91] - run - ProbePointServices starting
06 Sep 2011 13:41:55.227 [13974] INFO spl_pe MCEImpl.cpp:runDebugServices:712] - Notifying debug services of 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)
(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.



- ▶ 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.

```
11992, 11992
11993, 11993
11994, 11994
11995, 11995
11996, 11996
11997, 11997
11998, 11998
11999, 11999
90000, 12000
12001, 12001
12002, 12002
12003, 12003
12004, 12004
12005, 12005
12006, 12006
12007, 12007
12008, 12008
12009, 12009
12010, 12010
12011, 12011
12012, 12012
12013, 12013
```

Figure 5-15 Sample output file



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.



276 IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution





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.



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 quite 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*.



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



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



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++.



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 = getpid();
        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>>`.



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*.



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.



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 SPL source 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.



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
- ▶ pgallocc_high



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.sp1
namespace sample;
composite VMStat {
    type
        StatType = tuple<uint64 nr_page_table_pages,
                        uint64 nr_mapped,
                        uint64 nr_slab,
                        uint64 pgallocc_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.



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>
```



```

    <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.



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.



- ▶ `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`.



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()) {
```



```

        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`.



- ▶ 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.



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.



- ▶ `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>
```



```

        <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*.



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 is 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.



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) {
    IPort0Type const & ituple = static_cast<IPort0Type const&>(tuple);
    OPort0Type 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
```



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
```



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
```



```

        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.



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.



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 *SPI 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.



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 Transform 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
```



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>
```



```

        <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>
</cppOperatorMode>
</operatorMode>

```



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.



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() {}
}
```



```

    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)
{
    IPort0Type const & <%= $inTupleName %> = static_cast<IPort0Type
const&>(tuple);
    if (<%= $filterExpr %>) {
        <%SPL::CodeGen::emitSubmitOutputTuple($outputPort,
$inputPort);%>
    }
}

void MY_OPERATOR::process(Punctuation const & punct, uint32_t port)
{
    forwardWindowPunctuation(punct);
}

```



```
<%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);
    }
}
```



```
}  
}
```

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);
```



```

my $inTupleType = $inputPort->getCppType();
my $outTupleType = $outputPort->getCppType();
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);%>

```



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;  
}
```



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
```



```

    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*.





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®.



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.



- 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 shown in Figure 7-1 and explained in greater detail in the subsequent sections.

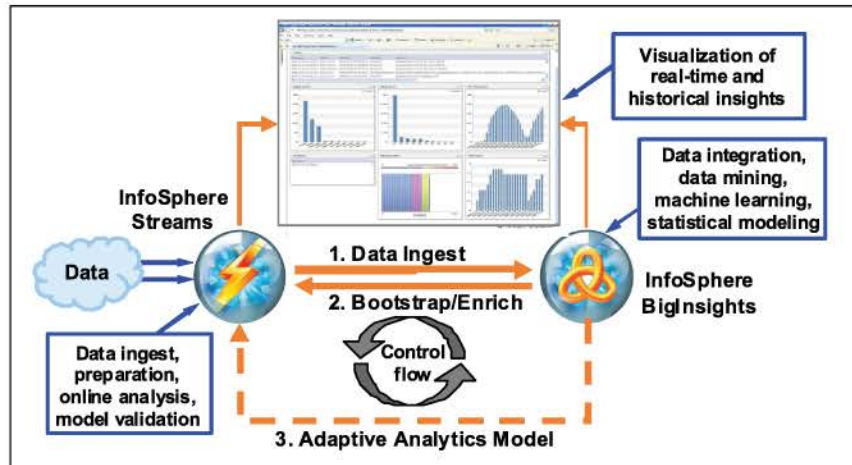


Figure 7-1 Big Data application scenarios

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.



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.



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.



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.

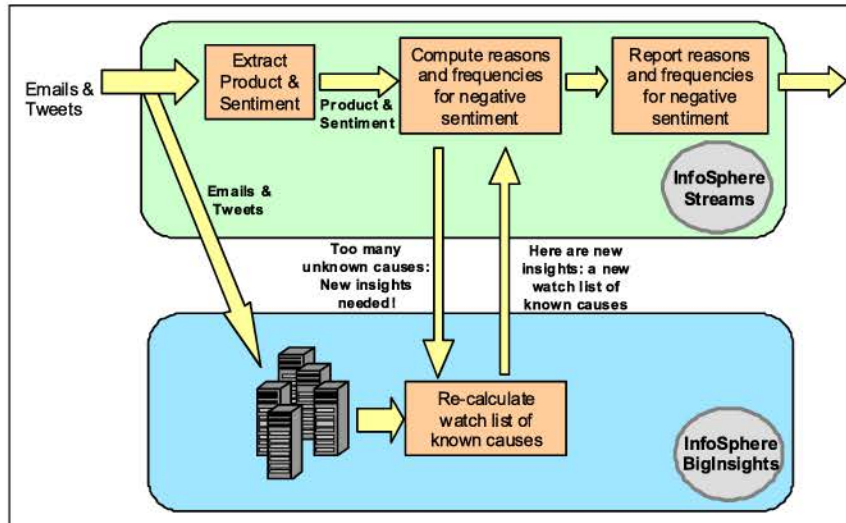


Figure 7-2 Streams and BigInsights application example

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



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.



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.



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 HDFSSink 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.



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.

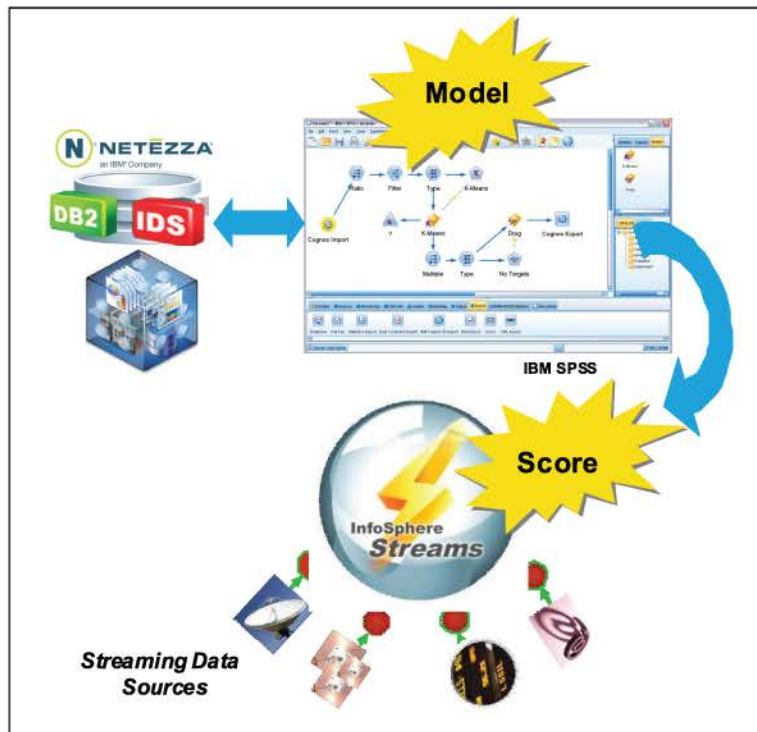


Figure 7-3 Streams and data mining

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.



- **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.



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.



In this section, we start with a working modeler session, as shown in Figure 7-4.

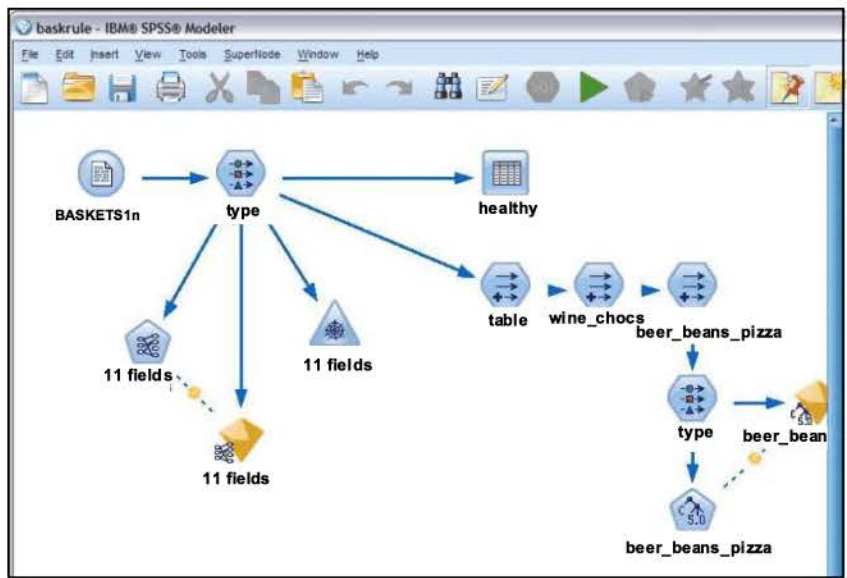


Figure 7-4 Working modeler session

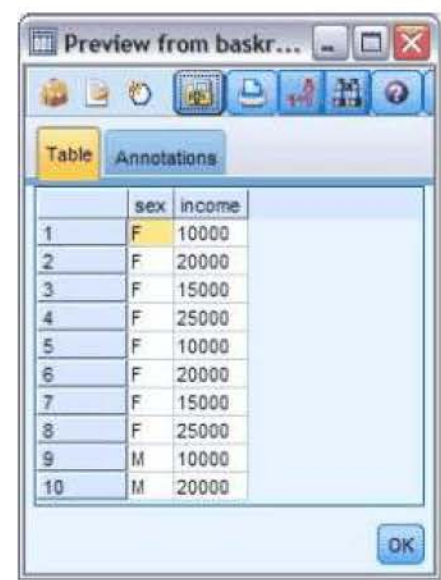
to use the predictive model built in this session, you need to create a scoring branch by connecting an input node to the scoring nugget and then connecting the scoring nugget to an output flat file, as shown in Figure 7-5. Note that this branch in the example is quite trivial and not representative 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



We have also created a sample input file to test the execution of the model, as shown in the preview in Figure 7-6.



	sex	income
1	F	10000
2	F	20000
3	F	15000
4	F	25000
5	F	10000
6	F	20000
7	F	15000
8	F	25000
9	M	10000
10	M	20000

Figure 7-6 Model preview



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	E
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



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.



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">
```



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.



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
```



```
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 clemrml 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.



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 * libclemrtl;
clemrtl_initialise_ext_t clemrtl_initialise_ext;
```



Then in the operator's constructor in the `_cpp.cgt` file the libraries are loaded using:

```
libclemrtl = dlopen("libclemrtl.so", RTLD_NOW | RTLD_DEEPBIND);
if(!libclemrtl)
    throw SPLRuntimeOperatorException(getThisLine(), dlerror());
/* get the routine addresses */
clemrtl_initialize_ext = (clemrtl_initialize_ext_t)
dlsym(libclemrtl,"clemrtl_initialize_ext");
if(!clemrtl_initialize_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_initialize_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_initialize_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:



```

/* 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);

```



```

/* 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;

```



```
...
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;
SPLOG(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;
```



```

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;

```



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_setAlternativeOutput` 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;
SPLOG(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;
SPLOG(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



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);
}
```



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) {
```



```

status = EXIT_FAILURE;
SPLOG(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.



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.

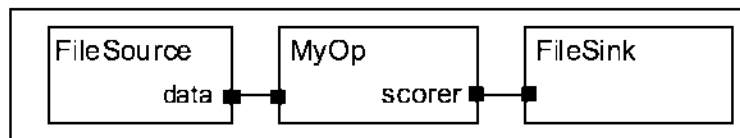


Figure 7-9 Sample SPL application

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.



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.sp1 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 STSPTTest/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
```



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
```



```

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}

```



```

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...

```



```

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

```



```
"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`.



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.



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.



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.



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.
- ▶ **load delimitedfile:** Allows you to pass in a comma-delimited file, used to create and populate a database table.



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 (loadlimitedfile, 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.



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`



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  onsocket      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



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)

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



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
```



```

Socket          = 16384
Servername      = 127.0.1.10
Port            = 5480
Database        = streamtest
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/sac12x6465.so

```



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
SOLDDIR	/mydir (directory containing the <code>solid.ini</code> 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>
```



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
```



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 using 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
```



368 IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution





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>



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.



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).



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
```



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



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.



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



Select **Disabled** and click **Apply**, as shown in Figure A-3.

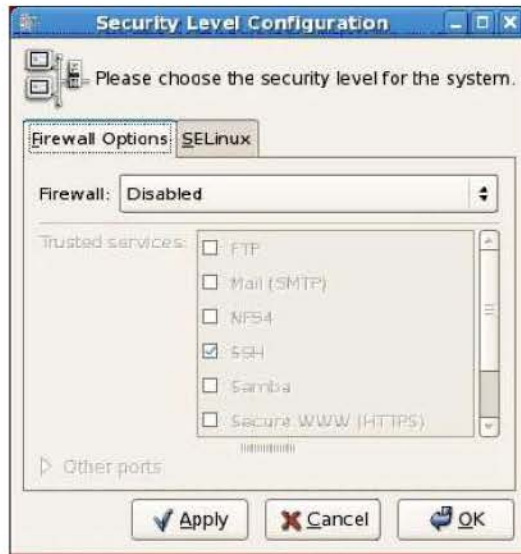


Figure A-3 Disable the firewall

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`.



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**.



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



11. Review dependencies.

Review the results of the software dependency check and click **Next**, as shown in Figure A-6.

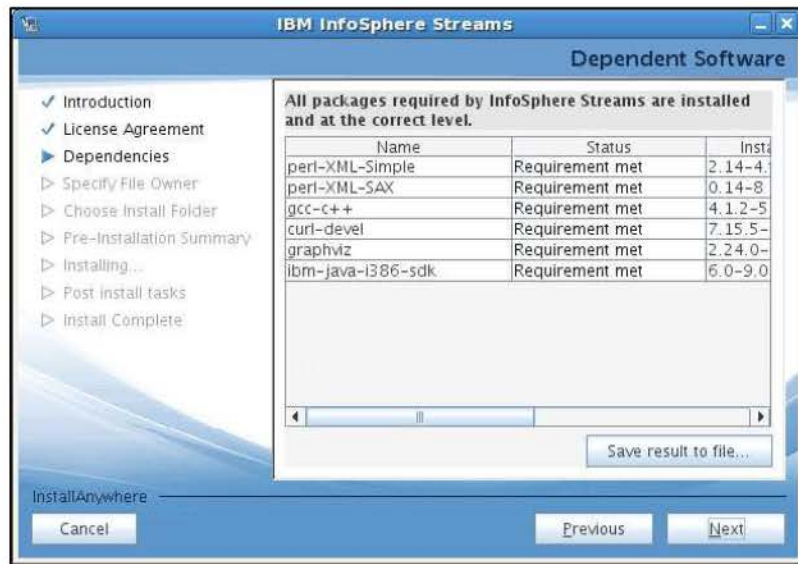


Figure A-6 Dependent Software window



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.

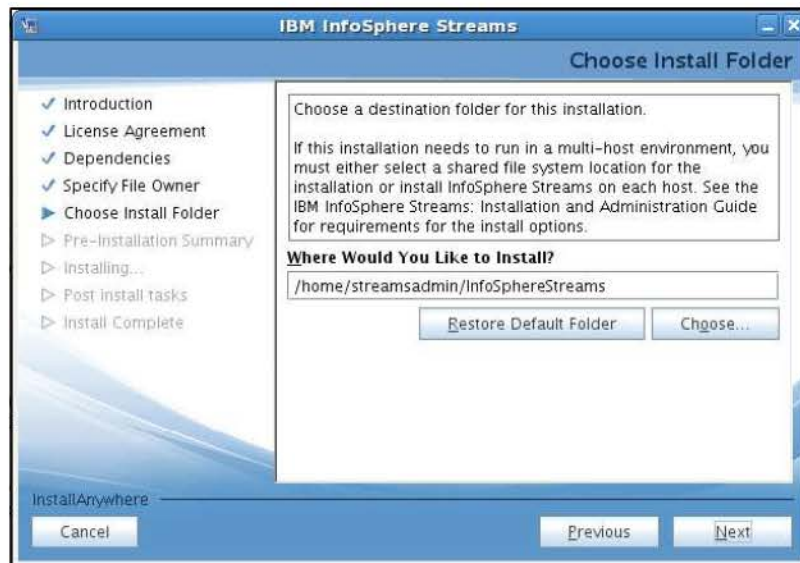


Figure A-7 Install folder



13. Review the pre-installation summary.

Review the information shown in Figure A-8, and click **Install**.



Figure A-8 Pre-installation Summary window



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.



- 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 &
```



Choose the workspace directory and click **OK**, as shown in Figure A-10.

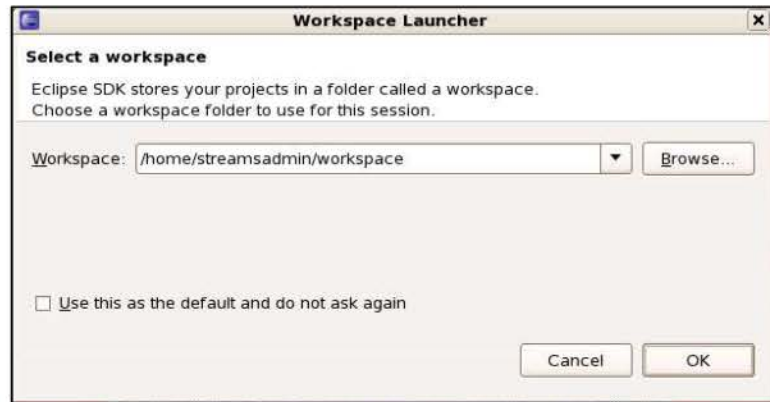


Figure A-10 Workspace Launcher window



- 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>.

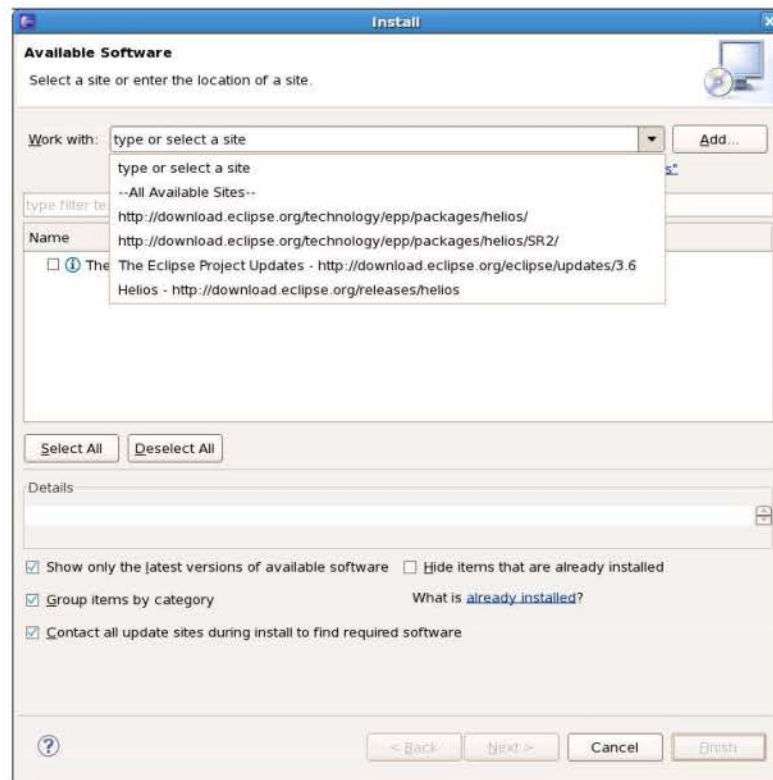


Figure A-11 Available Software window



Expand the **General Purpose Tools** option and select **XText SDK**, as shown in Figure A-12. Click **Next**.

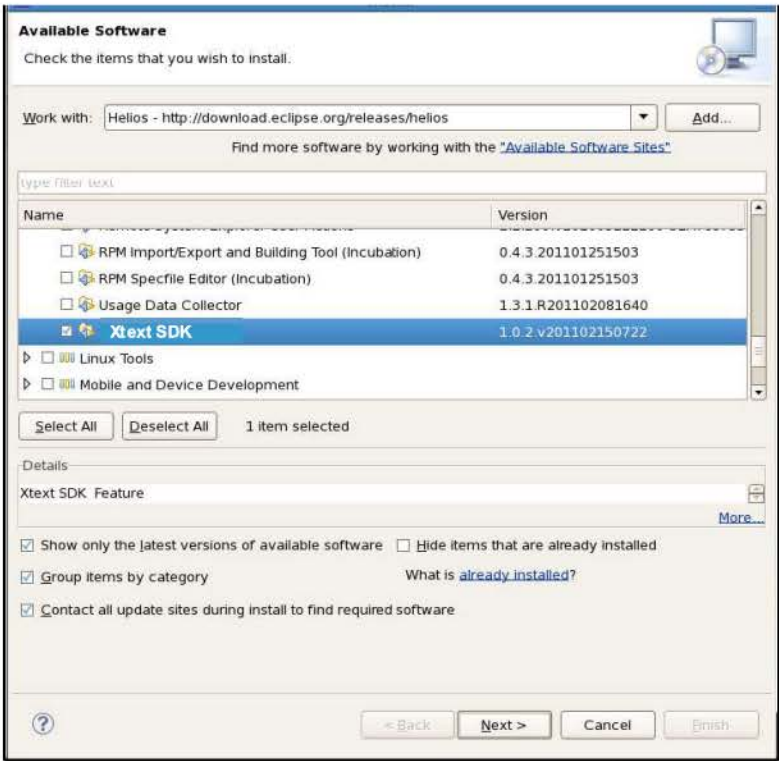


Figure A-12 XText SDK window

Complete the XText installation and restart the Eclipse IDE when prompted.



- 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.

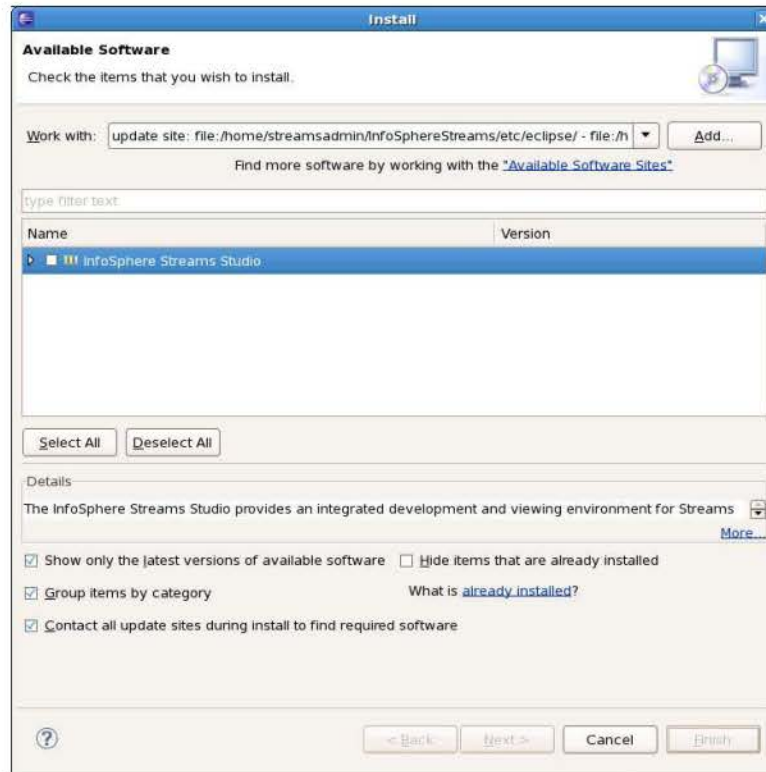


Figure A-13 Streams Studio Installation window

Click **Select All** and follow the prompts to complete the Streams Studio installation.



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).

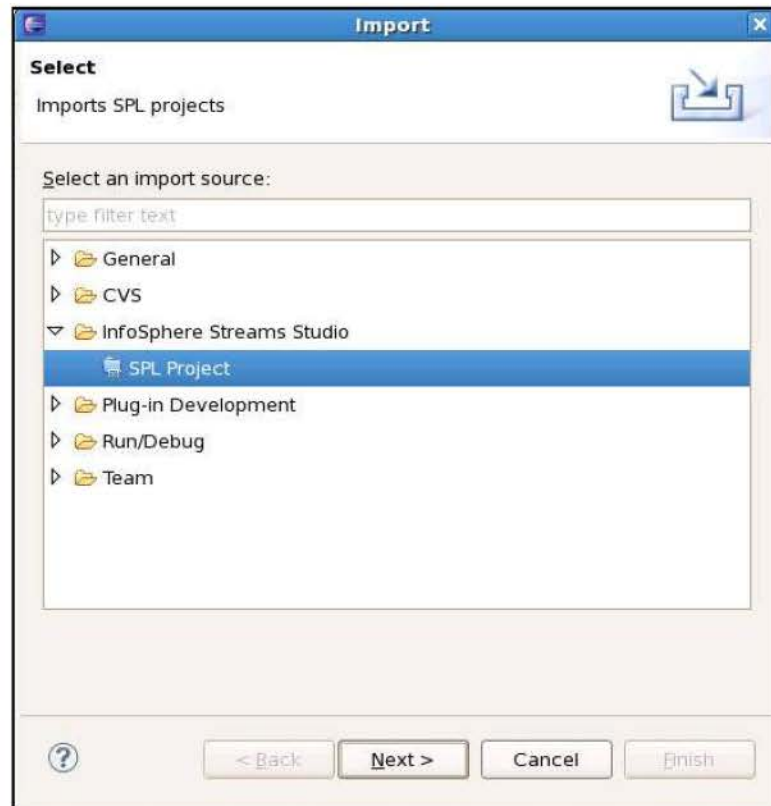


Figure A-14 Select an Import Source



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.

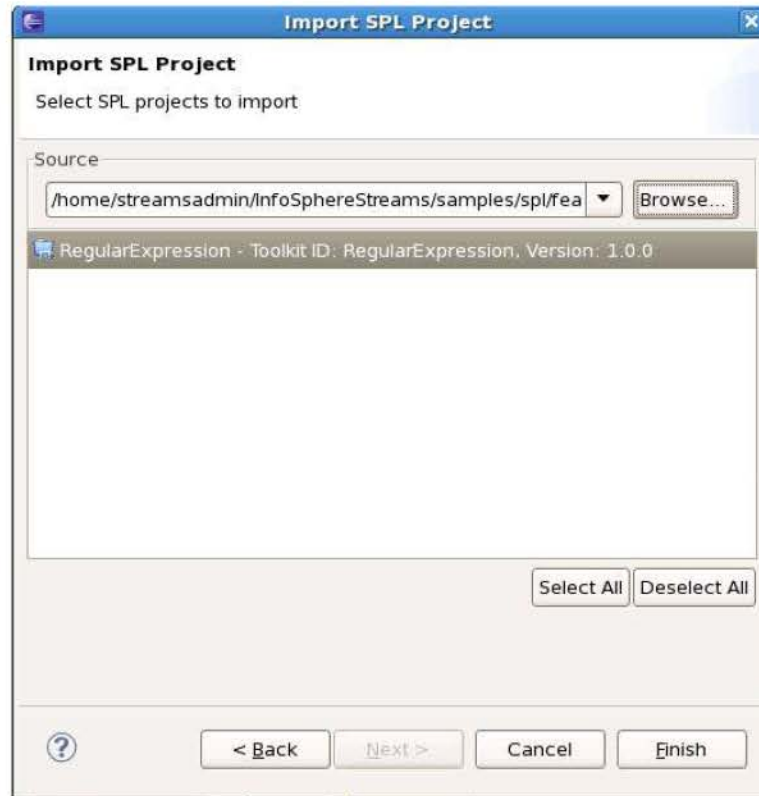


Figure A-15 Select an SPL Project to Import

Click **Select All** and then **Finish**.



- 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.

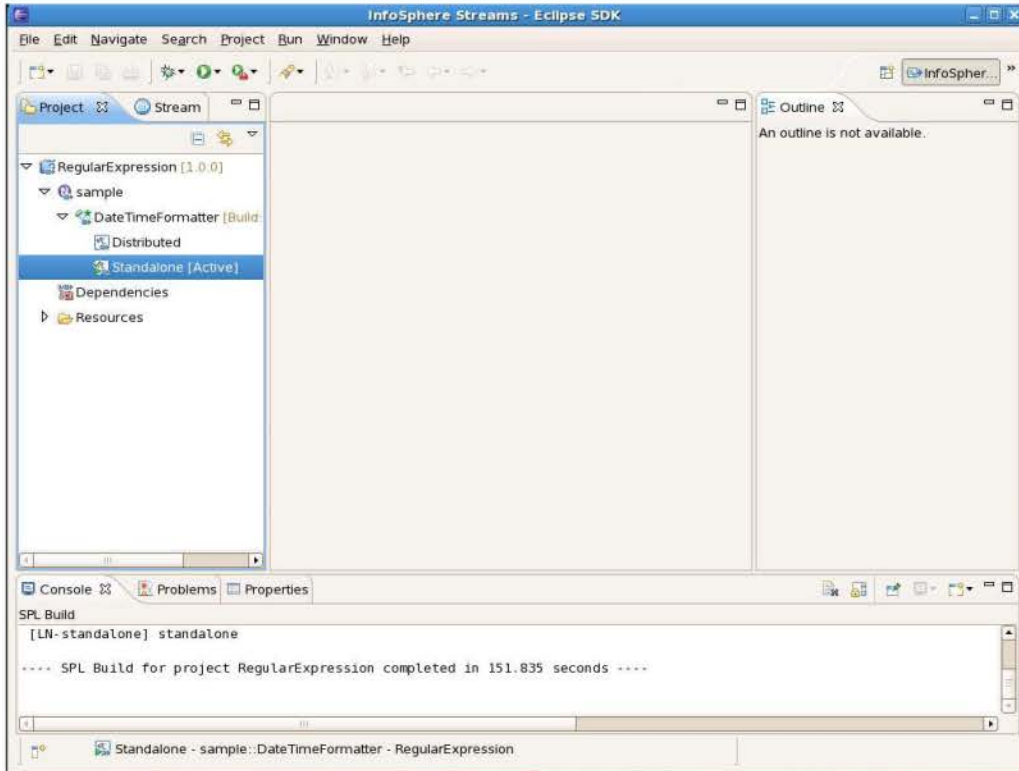


Figure A-16 Streams Studio Console

- 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
```



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.

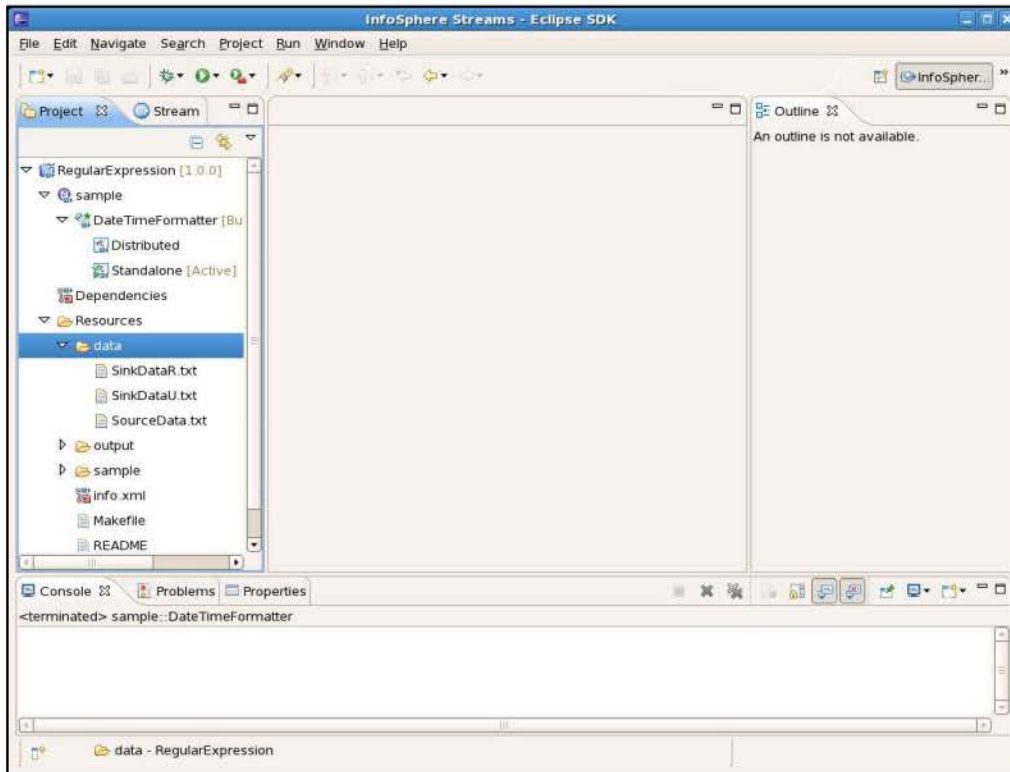


Figure A-17 SPL Project data directory

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.



## 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
```



**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.

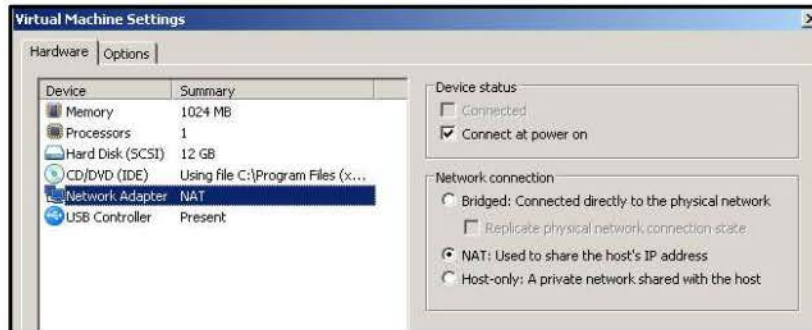


Figure A-18 Virtual Machine Settings window

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.



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.

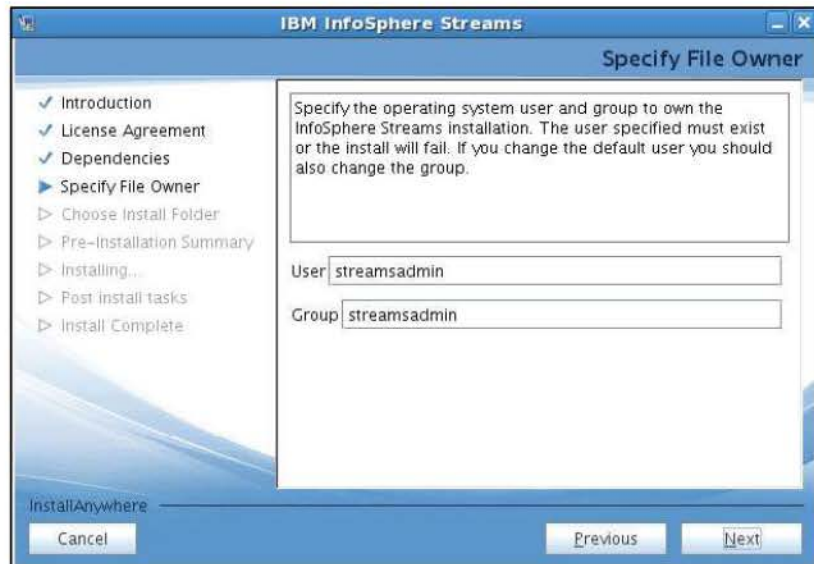


Figure A-19 Specify File Owner window



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.



**396** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution





## 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?



## 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.



- ▶ 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/sp1/application`
  - Compress
  - Ping
  - Sudoku
  - TrendCalculator
  - VMStat
  - Vwap
  - WordCount
- ▶ `ls $STREAMS_INSTALL/samples/sp1/feature`
  - CompositeParameter
  - LinuxPipe
  - OperatorAndLibrary
  - TypedArguments
  - Composites
  - LoopBack
  - RegularExpression
  - WindowLibrary
  - FanInFanOut
  - SampleToolkit
  - HostConstraint
  - MixedMode
  - SchemaSharing
  - JavaOperators
  - NativeFunction
  - TaskParallel
- ▶ `ls $STREAMS_INSTALL/samples/sp1/demo`
  - CommodityPurchasing



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.



## 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-level interfaces than those provided by the standard toolkit (which cover file and network interfaces).



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.



## 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.



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.



## 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



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.

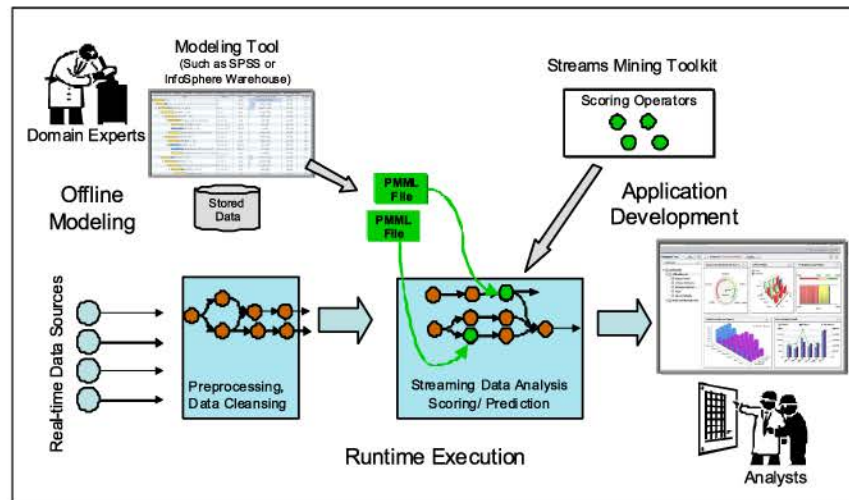


Figure B-1 Process flow of basic use case for Streams Mining Toolkit

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.



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.

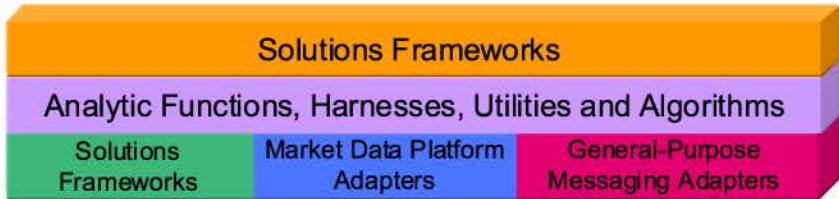


Figure B-2 Financial Toolkit organization

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.



## 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.)



- 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.



## 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.



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.



**412** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution





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](http://ibm.com/redbooks)

Select the **Additional materials** and open the directory that corresponds with the IBM Redbooks form number, SG247970.



## Using the web material

The additional web material that accompanies this book includes the following files:

| <i>File name</i>    | <i>Description</i>                                                                                                                    |
|---------------------|---------------------------------------------------------------------------------------------------------------------------------------|
| <b>baskrule.zip</b> | 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.



# 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 execution groups, the message flows and sets that are assigned to them, 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.



**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



**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.



**418** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution



# Abbreviations and acronyms

|              |                                              |                 |                                       |
|--------------|----------------------------------------------|-----------------|---------------------------------------|
| <b>AAS</b>   | Authentication and Authorization Service     | <b>JDBC</b>     | Java DataBase Connectivity            |
| <b>CFG</b>   | configuration                                | <b>JDK</b>      | Java Development Kit                  |
| <b>CORBA</b> | Common Object Request Broker Architecture    | <b>JDL</b>      | Job Description Language              |
| <b>CPU</b>   | central processing unit                      | <b>JE</b>       | Java Edition                          |
| <b>DDY</b>   | Distributed Distillery                       | <b>JMC</b>      | job manager controller                |
| <b>DFE</b>   | Distributed Front End                        | <b>JMN</b>      | job manager                           |
| <b>DFS</b>   | data fabric server                           | <b>LAS</b>      | logging and auditing subsystem        |
| <b>DGM</b>   | dataflow graph manager                       | <b>LDAP</b>     | Lightweight Directory Access Protocol |
| <b>DIG</b>   | dense information grinding                   | <b>Mb</b>       | megabits                              |
| <b>DNA</b>   | Distillery instance Node Agent               | <b>MB</b>       | megabytes                             |
| <b>DPFS</b>  | General Parallel File System                 | <b>MNC</b>      | master node controller                |
| <b>DPS</b>   | distributed processing system                | <b>NAM</b>      | naming                                |
| <b>DSF</b>   | Distillery Services Framework                | <b>NAN</b>      | nanoscheduler                         |
| <b>DSM</b>   | data source manager                          | <b>NC</b>       | node controller                       |
| <b>DSS</b>   | Distillery Semantic Services                 | <b>NDA</b>      | network data analysis                 |
| <b>DST</b>   | Distillery Base API                          | <b>ODBC</b>     | open database connectivity            |
| <b>EVT</b>   | event service                                | <b>OPT</b>      | optimizer                             |
| <b>Gb</b>    | gigabits                                     | <b>ORA</b>      | Ontology and Reference Application    |
| <b>GB</b>    | gigabytes                                    | <b>OS</b>       | Operating System                      |
| <b>GUI</b>   | graphical user interface                     | <b>PE</b>       | Processing Element                    |
| <b>I/O</b>   | input/output                                 | <b>PEC</b>      | Processing Element Container          |
| <b>IBM</b>   | International Business Machines Corporation  | <b>PHI</b>      | physical infrastructure               |
| <b>IDE</b>   | Integrated Development Environment           | <b>PRV</b>      | privacy                               |
| <b>IIDE</b>  | Distillery instance Node Agent               | <b>IBM RAA®</b> | Resource Adaptive Analysis            |
| <b>IKM</b>   | Information Knowledge Management             | <b>REF</b>      | reference application                 |
| <b>INQ</b>   | inquiry services                             | <b>REX</b>      | results/evidence explorer             |
| <b>ITSO</b>  | International Technical Support Organization | <b>RFL</b>      | resource function learner             |
|              |                                              | <b>RMN</b>      | resource manager                      |
|              |                                              | <b>RPS</b>      | repository                            |



|                |                                                         |
|----------------|---------------------------------------------------------|
| <b>RTAP</b>    | real-time analytic processing                           |
| <b>SAGG</b>    | statistics aggregator                                   |
| <b>SAM</b>     | Streams Application Manager                             |
| <b>SCH</b>     | Scheduler                                               |
| <b>SDO</b>     | stream data object                                      |
| <b>SEC</b>     | security                                                |
| <b>SMP</b>     | symmetric multiprocessing                               |
| <b>SOA</b>     | service-oriented architecture                           |
| <b>SODA</b>    | Scheduler                                               |
| <b>SPADE</b>   | Streams Processing<br>Application Declarative<br>Engine |
| <b>SPC</b>     | Streams Processing Core                                 |
| <b>SRM</b>     | Streams Resource Manager                                |
| <b>STG</b>     | storage                                                 |
| <b>Streams</b> | IBM InfoSphere Streams                                  |
| <b>SWS</b>     | Streams Web Service                                     |
| <b>SYS</b>     | system management                                       |
| <b>TAF</b>     | text analysis framework                                 |
| <b>TEG</b>     | traffic engineering                                     |
| <b>TCP</b>     | transmission control program                            |
| <b>TCP/IP</b>  | transmission control program<br>/internet protocol      |
| <b>TCP/UDP</b> | transmission control program<br>/user datagram protocol |
| <b>TRC</b>     | tracing (debug)                                         |
| <b>TSM</b>     | type system & metadata                                  |
| <b>UBOP</b>    | user-defined built-in operator                          |
| <b>UDF</b>     | user-defined function                                   |
| <b>UDOP</b>    | user-defined operator                                   |
| <b>UE</b>      | user experience                                         |
| <b>UTL</b>     | utilities                                               |
| <b>WLG</b>     | workload generator                                      |
| <b>URI</b>     | uniform resource identifier                             |
| <b>URL</b>     | universal record locator                                |
| <b>WWW</b>     | World Wide Web                                          |
| <b>XML</b>     | eXtensible Markup Language                              |

**420** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution



# 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](http://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>



- ▶ 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.svg.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](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](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.svg.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](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.

**422** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution



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 in Table 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](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).

Related publications **423**



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:<br><a href="http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/services/InfoSphere_Streams_Delivery_Capabilities.pdf">http://download.boulder.ibm.com/ibmdl/pub/software/data/sw-library/services/InfoSphere_Streams_Delivery_Capabilities.pdf</a> | 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: <ul style="list-style-type: none"> <li>► Installation and configuration support</li> <li>► Getting acquainted with IBM InfoSphere Streams</li> <li>► Mentored pilot design</li> <li>► Streams programming language education</li> <li>► Supplemental Streams programming support</li> </ul> |

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



- ▶ 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](http://ibm.com/redbooks)

Related publications **425**



## Help from IBM

IBM Support and downloads

[ibm.com/support](http://ibm.com/support)

IBM Global Services

[ibm.com/services](http://ibm.com/services)

**426** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution



# 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\_setAlternativeOutput 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



- 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

**428** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution



## 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

Index 429



## 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



- 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



**432** IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution





IBM InfoSphere Streams: Assembling Continuous Insight in the Information Revolution

(0.5" spine)  
0.475" x 0.875"  
250 x 459 pages









# 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](http://ibm.com/redbooks)

SG24-7970-00

ISBN 0738436151



## JURAT ATTACHMENT

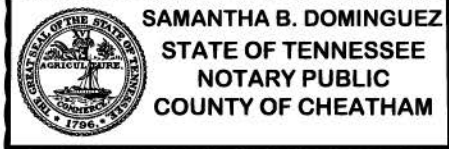
A notary public or other officer completing this certificate verifies only the identity of the individual who signed the document to which this certificate is attached, and not the truthfulness, accuracy, or validity of that document.

STATE OF Tennessee }

COUNTY OF Cheatham }

The foregoing instrument was subscribed and sworn before me this date of  
05/09/2024 by Nathaniel Frank-White

~~This notarial act was an online notarization.~~



ONLINE NOTARY PUBLIC  
MY COMMISSION EXPIRES: JULY 26 2025

(Notary Seal)

Notary's Signature *Samantha B. D.*

Registration No.: 0

Commission Expiration Date: July 26, 2025

