

UNITED STATES PATENT AND TRADEMARK OFFICE

BEFORE THE PATENT TRIAL AND APPEAL BOARD

PALO ALTO NETWORKS, INC.
Petitioners,

v.

QOMPLX LLC,
Patent Owner.

Case No. IPR2026-00371

U.S. Patent No. 12,143,425

**DECLARATION OF PAUL S. MIN, PH.D.
IN SUPPORT OF PETITION FOR *INTER PARTES* REVIEW OF
U.S. PATENT NO. 12,143,425**

TABLE OF CONTENTS

I.	Introduction	1
II.	Qualifications and Background	1
III.	Legal Standards	5
A.	Claim Construction	6
B.	Anticipation Under 35 U.S.C. § 102	7
C.	Obviousness Under 35 U.S.C. § 103	7
IV.	The '425 patent	10
A.	Background of Technology	10
1.	Distributed, Highly Scalable, Stream Processing Engines Were Commonly Used For Real-Time Data Processing	10
2.	Stream Processing Engines Typically Used Directed Graphs To Organize And Manage Processing Data Streams	13
B.	Overview of the '425 patent	15
1.	The Claims' Earliest Effective Filing Date	15
2.	The '425 Patent's Specification	15
C.	Prosecution History of the '425 patent	16
D.	Person of Ordinary Skill in the Art	17
V.	Claim Construction of Terms of the '425 patent	18
VI.	Overview of Prior Art	19
A.	Barsness (EX1004)	19
B.	Chakradhar (EX1005)	24
1.	Chakradhar Is Supported By The Chakradhar Provisional	26

2.	The Chakradhar Provisional Supports Claim 1 Of Chakradhar	26
3.	The Chakradhar Provisional Supports Relied-Upon Subject Matter Of Chakradhar	27
C.	Siripurapu (EX1007).....	28
VII.	Summary of Opinions on Unpatentability	30
A.	Ground 1: Claims 1-6 are Rendered Obvious by Barsness	31
1.	Claim 1	31
2.	Claim 2 – The system of claim 1, wherein the third software instructions monitor at least a portion of the execution of at least one of the first software instructions and the second software instructions to identify execution that has become unresponsive.	107
3.	Claim 3 – The system of claim 1, wherein the first computer system and the second computer system are distinct.	109
4.	Claim 4	110
5.	Claim 5 – The system of claim 1, wherein the third software instructions are further configured to monitor at least a portion of the execution of at least one of the first software instructions and the second software instructions, to identify a transformation pipeline that causes at least one of the first computer system and the second computer system to transmit information that is outside established data stability boundaries.....	118
6.	Claim 6 – The system of claim 5,.....	119
B.	Ground 2: Claims 1-6 Are Obvious Over Barsness in View of Chakradhar	122
1.	Motivation to Combine and Reasonable Expectation of Success	123

2.	Claim 1	125
3.	Claim 4	128
C.	Ground 3: Claims 1-6 Are Obvious Over Barsness in View of Siripurapu	129
1.	Motivation to Combine and Reasonable Expectation of Success	130
2.	Claims 1-6	133
D.	Ground 4: Claims 1-6 Are Obvious Over Barsness in View of Chakradhar and Siripurapu	137
VIII.	Secondary Considerations of Non-obviousness.....	138
IX.	Conclusion.....	139

TABLE OF EXHIBITS LIST

Ex.	Description
1001	U.S. Patent No. 12,143,425 (“the ’425 patent”)
1002	Prosecution History for U.S. Patent No. 12,143,425
1003	Intentionally Omitted
1004	U.S. Patent No. 8,732,300 (“Barsness”)
1005	U.S. Patent Application Pub. No. 2016/0269247 A1 (“Chakradhar”)
1006	U.S. Provisional Application No. 62/132,686 (“Chakradhar Provisional”)
1007	U.S. Patent Application Pub. No. 2014/0059185 A1 (“Siripurapu”)
1008	Thomas Heinze, et al., Cloud-based Data Stream Processing, Proc. of the 8th ACM Int’l Conf. on Distributed Event-Based Sys., 238-245 (May 2014), https://di.acm.org/doi/10.1145/2611286.2611309 (“Heinze”)
1009	Ashkon Soroudi, et al., High Availability on a Distributed Real Time Processing System, Electrical Engineering and Computer Sciences, University of California at Berkeley (May 15, 2015), http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-135.html (“Soroudi”)
1010	Naresh Kumar Reddy Rapolu, Dynamic Re-Optimization Techniques for Stream Processing Engines and Object Stores (2015) (Ph.D. dissertation, Purdue University) (“Rapolu”)
1011	Ankit Toshniwal, et al., Storm @Twitter, SIGMOD ’14: Proc. of the 2014 ACM SIGMOD Int’l Conf. on Mgmt. of Data, 147-156 (2014), http://dx.doi.org/10.1145/2588555.2595641 (“Toshniwal”)
1012	Leonardo Neumeyer, et al., S4: Distributed Stream Computing Platform, IEEE Int’l Conf. on Data Mining Workshops, 170-177

	(2010), https://ieeexplore.ieee.org/document/5693297 (“Neumeyer”)
1013	Michael Duller, et al., Virtualizing Stream Processing, Middleware (2011) (“Duller”)
1014	Scott Schneider, et al., Tutorial: Stream Processing Optimizations, DEBS ’13 (2013) (“Schneider”)
1015	U.S. Patent No. 8,407,360 (“Andrade”)
1016	U.S. Patent No. 8,560,527 (“Santosuosso”)
1017	U.S. Patent No. 8,190,960 (“Bahadur”)
1018	<i>Curriculum Vitae</i> of Dr. Paul Min
1019	Plaintiff Qomplx LLC’s Responsive Claim Construction Brief, (Dkt. 57), <i>Qomplx LLC v. Microsoft Corporation</i> , Case No. 1:25-cv-01383-ADA

I. INTRODUCTION

1. I, Paul S. Min, Ph.D., submit this declaration to state my opinions on the matter described below.

2. I have been retained by Petitioner Palo Alto Networks, Inc. (“Petitioner”), as an independent expert in this proceeding before the United States Patent and Trademark Office. Although I am being compensated at my usual and customary rate of \$500 per hour, no part of my compensation depends on the outcome of this proceeding, and I have no other interest in this proceeding.

3. I understand that this proceeding involves U.S. Patent No. 12,143,425 (the “’425 patent”), and I have been asked to consider the validity of certain claims of the ’425 patent based on certain prior art references. I have also been asked to consider the state of the art and prior art available as of October 28, 2015. Based on the prior art discussed in this declaration, it is my opinion that claims 1-6 of the ’425 patent are unpatentable for the reasons provided below.

II. QUALIFICATIONS AND BACKGROUND

4. I believe that I am well qualified to serve as a technical expert in this matter based upon my educational and work experience, and specifically, in the field of vehicle electronics and computing devices. My curriculum vitae (“CV”) is submitted as Exhibit 1018.

5. I received a B.S. degree in Electrical Engineering in 1982, an M.S. degree in Electrical Engineering in 1984, and a Ph.D. degree in Electrical Engineering in 1987 from the University of Michigan in Ann Arbor. I received several academic honors, including my B.S. degree with honors, a best graduate student award and a best teaching assistant award during my M.S. study, and a best paper award from a major international conference for reporting results from my Ph.D. thesis.

6. After receiving my Ph.D., I worked at Bellcore in New Jersey from August 1987 until August 1990. At Bellcore, I was responsible for evolving the public switched telephone network (POTS) into a multi-services voice and data network that incorporated packet switches, optical technologies, and wireless technologies.

7. In September 1990, I joined the faculty at Washington University in St. Louis. In July 1996, I was promoted to an Associate Professor of Electrical Engineering with tenure. I am currently a Senior Professor at Washington University of the Electrical and Systems Engineering. I have also served as the Chair of the Graduate Curriculum (2000-2002) and the Chair of the Undergraduate Curriculum (2011-2014) for the Electrical and Systems Engineering department.

8. At Washington University, I have conducted research in communication, computing, and related electronic hardware and software. My

research group has pioneered a new paradigm for designing electronic circuits that can alleviate the speed and performance mismatch against optical technology. I have received several grants from the U.S. Federal Agencies, including the National Science Foundation and the Defense Advanced Research Project Agency, and numerous contracts from companies and organizations around the world. Specifically related to the technology matters in this Investigation, I have researched various computing paradigms including pipeline architecture, distributed computing platform, peer-to-peer multi-processor architecture. In addition, I have developed various monitoring/inspection techniques for the extremely high-speed Internet traffic I have an extensive background and experience in each of these technologies.

9. As a faculty member at Washington University, I have taught a number of courses in electronics, communication, and computing at both the undergraduate and graduate levels. For example, I have taught communication theory (Washington University ESE 471), transmission and multiplexing (Washington University ESE 571), and signaling and control of communication networks (Washington University ESE 572).

10. I have supervised numerous graduate students, many of whom received a doctoral degree under my guidance. A number of doctoral theses that I have supervised relate specifically to multi-processor distributed computing technology.

11. In addition to my responsibilities as a university faculty member, I have founded two companies. In May 1997, I founded MinMax Technologies, Inc., a fabless semiconductor company that developed switch fabric integrated circuit chips for the Internet. In March 1999, I founded Erlang Technology, Inc., a fabless semiconductor company that focused on the design and development of integrated circuit chips and software for the Internet. One of Erlang's products received a best product of the year award in 2004 from a major trade journal for the electronics industry.

12. Outside my own start-up companies, I have also served in various technology and business advisor roles for other companies and organizations around the world. I was the main technical author for one of two winning proposals to the Korean government for CDMA wireless service licenses (1996). I was responsible for designing a commercial scale IS-95 CDMA cellular network, which I understand to be one of the earliest such networks deployed in the world. I worked with numerous engineers and scientists around the world to implement this commercial-scale cellular network before IS-95 CDMA was widely accepted. This provided me with extensive insight into various components of CDMA technology, which by and large are used in WCDMA network. I have also been involved in a semiconductor company that specializes in semiconductor memories such as flash EEPROMs as a board member and as a technical advisor (2007-2011).

13. I am a named inventor on eleven U.S. patents. I have extensively published technical papers in international conferences and journals, technical memoranda and reports, and given a number of seminars and invited talks. Many of these papers are specifically within the context of the distributed computing technology. I have organized several international conferences and served as an international journal editor.

14. I am a member of and have been actively involved in a number of professional organizations. For example, I have served as the Chair of the Saint Louis Section of the IEEE with more than 3,000 members (2014), and a member of the Eta Kappa Nu Honor Society for electrical engineers. I have also been an Ambassador of the McDonnell International Scholars Academy (2007-2013).

III. LEGAL STANDARDS

15. In forming my opinions and considering the subject matter of the '425 patent and its claims in light of the prior art, I am relying on certain legal principles that counsel in this case explained to me. My understanding of these concepts is summarized below.

16. I understand that the claims define the invention. I also understand that an unpatentability analysis is a two-step process. First, the claims of the patent are construed to determine their meaning and scope. Second, after the claims are construed, the content of the prior art is compared to the construed claims.

17. I understand that a claimed invention is only patentable when it is new, useful, and non-obvious in light of the “prior art.” That is, the invention, as defined by the claims of the patent, must not be anticipated or rendered obvious by the prior art.

A. Claim Construction

18. I understand that the United States Patent and Trademark Office interprets claim terms in an *inter-partes* review proceeding under the same claim construction standard that is used in a United States federal court. I understand that under this standard, the meaning of claim terms is considered from the viewpoint of one of ordinary skill in the art at the time of the alleged invention.

19. I understand that claim terms are generally given their ordinary and customary meaning as understood by one of ordinary skill in the art in light of the specification and the prosecution history pertaining to the patent. I understand, however, that claim terms are generally not limited by the embodiments described in the specification.

20. I understand that in addition to the claims, specification, and prosecution history, other evidence may be considered to ascertain the meaning of claim terms, including textbooks, encyclopedias, articles, and dictionaries. I have been informed that this other evidence is often less significant and less reliable than the claims, specification, and prosecution history.

B. Anticipation Under 35 U.S.C. § 102

21. I understand that under 35 U.S.C. § 102, a patent claim is invalid if its subject matter was patented or described in a printed publication before the effective filing date of the claimed invention. I have been told that this is referred to as invalidity by anticipation. I have been told that a patent claim is anticipated under § 102 if a single prior art reference discloses all limitations of the claimed invention.

C. Obviousness Under 35 U.S.C. § 103

22. I understand that a patent claim is invalid as obvious if the claimed invention would have been obvious to a person of ordinary skill in the art (“POSITA”) at the time the claimed invention was made. This means that even if all of the elements of the claim cannot be found in a single prior art reference that would anticipate the claim, a person of ordinary skill in the field who knew about all the prior art would have come up with the claimed invention. I understand that in an obviousness determination, the person of ordinary skill in the art is presumed to have knowledge of all material prior art. I understand that whether a claim is obvious is based upon the determination of several factual issues. I understand that obviousness is a determination of law based on underlying determinations of fact. I understand that these factual determinations include the scope and content of the prior art, the level of ordinary skill in the art, the differences between the claimed invention and the prior art, and secondary considerations of non-obviousness.

23. In considering obviousness, I understand that one must determine the scope and content of the prior art. I understand that, in order to be considered as prior art to a patent being considered, a prior art reference must be reasonably related to the claimed invention of that patent. A reference is reasonably related if it is in the same field as the claimed invention or is from another field to which a person of ordinary skill in the art would look to solve a known problem.

24. I understand that one must determine what differences, if any, existed between the claimed invention and the prior art.

25. I understand that a patent claim composed of several elements is not proved obvious merely by demonstrating that each of its elements was independently known in the prior art. In evaluating whether such a claim would have been obvious, one may consider whether a reason has been identified that would have prompted a person of ordinary skill in the art to combine the elements or concepts from the prior art in the same way as in the claimed invention. There is no single way to define the line between true inventiveness on the one hand (which is patentable) and the application of common sense and ordinary skill to solve a problem on the other hand (which is not patentable). For example, market forces or other design incentives may be what precipitated a change, rather than true inventiveness.

26. I understand that whether a prior art reference renders a patent claim unpatentable as obvious is determined from the perspective of a person of ordinary

skill in the art at the time of the alleged invention. I have been told that there is no requirement that the prior art contain an express suggestion to combine known elements to achieve the claimed invention, but a suggestion to combine known elements to achieve the claimed invention may come from the prior art, as filtered through the knowledge of one skilled in the art. In addition, I have been told that the inferences and creative steps a person of ordinary skill in the art would employ are also relevant to the determination of obviousness.

27. I understand that there is no rigid rule that a reference or combination of references must contain a “teaching, suggestion, or motivation” to combine references. But I also understand that the “teaching, suggestion, or motivation” test can be a useful guide in establishing a rationale for combining elements of the prior art. I have been told that this test poses the question as to whether there is an express or implied teaching, suggestion, or motivation to combine prior art elements in a way that realizes the claimed invention, and that it seeks to counter impermissible hindsight analysis.

28. I understand that one may consider, e.g., whether (1) the change was merely the predictable result of using prior art elements according to their known functions, or whether it was the result of true inventiveness; (2) there is some teaching or suggestion in the prior art to make the modification or combination of elements claimed in the patent; (3) the claimed innovation applies a known technique

that had been used to improve a similar device or method in a similar way; (4) the claimed invention would have been obvious to try, meaning that the claimed innovation was one of a relatively small number of possible approaches to the problem with a reasonable expectation of success by those skilled in the art; (5) the invention merely substituted one known element for another known element in order to obtain predictable results; (6) the invention merely applies a known technique to a known device, method, or product to yield predictable results; or (7) known work in the field may have prompted variations of use of the same inventions in the same or different fields due to market forces or design incentives that would have been predictable to a person of ordinary skill in the art.

29. I understand that any assertion of secondary considerations of non-obviousness must be accompanied by a nexus between the merits of the invention and the evidence offered.

IV. THE '425 PATENT

A. Background of Technology

1. Distributed, Highly Scalable, Stream Processing Engines Were Commonly Used For Real-Time Data Processing

30. By 2015, distributed stream processing engines were already well established tools for real-time analysis of continuous data streams. Contemporary literature such as Heinze explained that cloud-based data stream processing systems were designed for high-throughput, low-latency processing of unbounded streams,

-10-

and that newer “third generation” systems were specifically built to be “highly scalable and robust towards faults.” EX1008, 238. Heinze explained that cloud-based streaming engines were “designed to dynamically scale to hundreds of computing nodes and automatically cope with varying workloads.” EX1008, 240. These systems provided parallel processing of data streams and distributed multiple data streams / queries across a cluster. EX1008, 240. These “elastic data stream processing systems” could “dynamically vary the number of processing nodes within the system based on the workload.” EX1008, 241; *id.*, 238, 240 (“Elasticity describes the behavior of a system to dynamically scale in or out based on a changing workload.”).

31. Other references confirm that these characteristics were standard features of practical stream processing systems. As early as 2010, there was already “a clear need for highly scalable stream computing solutions that can operate at high data rates and process massive amounts of data.” EX1012, 170. Neumeyer described S4 in 2010 as a “distributed, scalable” platform that “allow[ed] programmers to easily develop applications for processing continuous unbounded streams of data,” and explained that S4’s design was intended for “low latency” operation in “large clusters built with commodity hardware.” EX1012, 170. Neumeyer further explained that production environments required “scalability (ability to add more servers to increase throughput with minimal effort) and high availability (ability to achieve

continuous operation with no human intervention in the presence of system failures).” EX1012, 170. Stream processing engines such as Storm were designed to be scalable, resilient, extensible, efficient, and easy to administer. EX1011, 147. Toshniwal described Storm as a “real-time fault-tolerant and distributed stream data processing system” used at Twitter “at scale, and in real-time,” and explained that Storm was specifically designed so operators could “add or remove nodes from the Storm cluster without disrupting existing data flows,” while continuing to operate on “large clusters.” EX1011, 147.

32. Other literature reflecting the knowledge of a POSITA recognized scalability through parallelization and scale-out as a core requirement of streaming systems. Schneider explained that handling “large volumes of live data in short periods of time” made “high throughput processing” a “critical requirement,” which in turn required taking advantage of “multiple cores and/or host machines to achieve scale.” EX1014, 249; *see also* EX1011, 147 (“Many modern data processing environments require processing complex computation on streaming data in real-time.”); EX1009, 3 (“The term ‘real-time processing’ in this context means that the system is able to respond within seconds or sub-second to requests, while ‘distributed’ means that it is running on multiple computers.”). Duller similarly explained that a stream-processing application could be “scale[d] out” across nodes

and that the tested application “linearly scale[d] up the load ... with the number of nodes.” EX1013, 283.

33. By 2015, various “distributed stream processing engines,” such as Aurora, Borealis, TimeStream, System S, and open-source systems such as S4, Samza, and Storm were available “[t]o process continuous streams of data at low latency.” EX1010, 2, 12; *see also* EX1011, 147.

2. Stream Processing Engines Typically Used Directed Graphs To Organize And Manage Processing Data Streams

34. It was well known by 2015 to use directed graphs to represent and manage the flow of data through streaming applications: “Stream engines code the application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology.” EX1010, 2; *see also* EX1013, 271-273 (describing a “data processing model” comprising a “graph” of operators). Further, “[t]uples are processed in a pipelined fashion as they traverse through the topology.” EX1010, 2. “A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components.” EX1011, 148 (“The basic Storm data processing architecture consists of *streams of tuples* flowing through *topologies*.”).

35. Schneider explained that a “stream processing application is organized as a *graph*, formed by a set of *operators* connected to each other by *streams*,” where

an operator is a “generic data manipulator[],” and “[a] stream is a series of *data items*.” EX1014, 250.

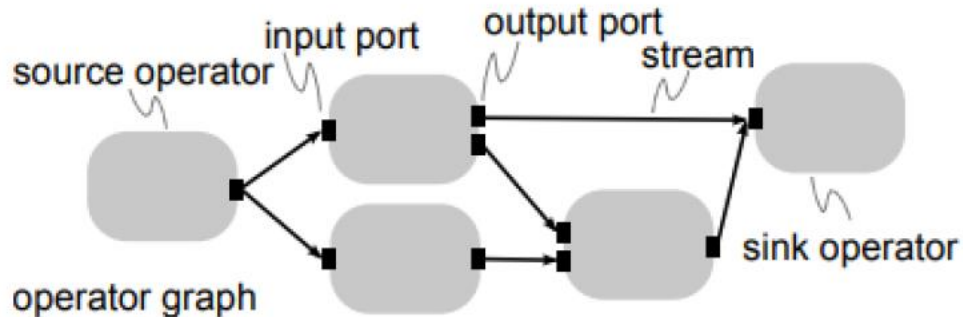


Figure 1: Basic concepts related to streaming applications.

EX1014, 250. Schneider explains how the operators of the graph are connected via input ports and output ports. *Id.* Further, source operators perform “edge adaptation to receive data from an external source,” while sink operators perform “edge adaptation to deliver data” to an external sink. *Id.* Andrade, an IBM patent, similarly explains:

A data-flow graph can include a set of processing elements (PEs) connected by streams, wherein each stream carries a series of stream data objects SDOs.) The PEs implement data stream analytics and are basic execution units that are distributed over the compute nodes. The PEs also communicate with each other via input and output ports, connected by streams.

EX1015, 5:31-39.

B. Overview of the '425 patent

1. The Claims' Earliest Effective Filing Date

36. For this declaration, I have been asked to assume that the claims' earliest possible effective filing date is October 28, 2015. That said, my opinions herein would not be materially different if the effective filing date was July 21, 2024.

2. The '425 Patent's Specification

The '425 patent describes the “analysis of very large data sets using distributed computational graph tools which allow for transformation of data through both linear and non-linear transformation pipelines.” EX1001, 3:32-35. Among other things, it describes determining when aspects of the streaming data analysis “are processing slowly or may have hung ... so that actions can be implemented to resolve the issue.” EX1001, 16:43-58. The '425 patent does not, however, further describe these issue-resolving “actions” or how to implement them.

37. The '425 patent provides examples of prior art transformation pipelines (FIGS. 6, 12) and other transformation pipelines (FIGS. 7-9, 13-15). EX1001, 6:46-50 (describing “FIG. 6 (Prior Art)”), 7:9-12 (describing “FIG. 12 (Prior Art)”), EX1001, 17:13-19:25, 21:33-22:59 (describing transformation pipelines). However, the '425 patent provides no disclosure of how to implement these transformation pipelines using the general hardware architecture shown in FIG. 3, with “any number of clients and/or servers.” EX1001, 13:52-64.

C. Prosecution History of the '425 patent

38. The first Office action granted a request for prioritized examination. EX1002, 420-21 (Decision Granting Request for Prioritized Examination (Track I)). Thereafter, the applicant and Examiner agreed to an examiner's amendment, and the Examiner issued a notice of allowability. EX1002, 422 ("proposed amendments were agreed upon, and included as examiner's amendment"), 501-509 (Notice of Allowance and Examiner's Amendments). The Reasons for Allowance concluded that the prior art of record did not disclose the amended claim elements, namely:

in response to the monitoring, cause a fourth computer system of the third plurality of computer systems to execute fourth software instructions that perform at least one of: processing second portions of the first stream of data by applying the first transformation pipeline to the second portions of the first stream of data, or processing second portions of the first pipeline output messages by applying a second transformation pipeline to the second portions of the first pipeline output messages, wherein the first computer system and the third computer system are distinct, and wherein the third computer system and the fourth computer system are distinct.

EX1002, 506-09 (Examiner's Amendments); *id.*, 509-513 (discussing prior art of record).

D. Person of Ordinary Skill in the Art

39. I understand that in evaluating obviousness of the '425 patent claims, the content of a patent or printed publication prior art should be interpreted the way a person of ordinary skill in the art would have interpreted the prior art as of the effective filing date of the challenged patent claims. I assume that the effective filing date is the earliest filing date of an alleged related application, namely October 28, 2015. My understanding of a person of ordinary skill in the art would not materially differ if the effective filing date were the date the application resulting in the '425 patent was filed, namely July 21, 2024, at least because the knowledge and level of skill of a person of ordinary skill in the art in 2024 would encompass the knowledge and skill of such person in 2015.

40. I understand that factors that may be considered in determining the level of ordinary skill in the art at the time of the effective filing date of the challenged patent include: (1) the educational level of the inventor; (2) the types of problems encountered in the art; (3) prior art solutions to those problems; (4) rapidity with which innovations are made; (5) sophistication of the technology; and (6) educational level of active workers in the field.

41. A person of ordinary skill in the art (“POSITA”) in July 2024 or October 2015 would have had a bachelor’s degree in computer science, computer engineering, electrical engineering, or an equivalent degree, with two to four years’ experience designing, implementing, or otherwise working with distributed computing systems. More education, e.g., a Master’s or Ph.D., would compensate for less work experience and vice versa.

42. A POSITA would have been familiar with common, then-existing technologies for distributed computing and cloud-based stream processing systems, as well as issues surrounding the design, implementation, and maintenance of such technologies. The state of the art below reflects certain technologies within the knowledge and skill of a POSITA.

V. CLAIM CONSTRUCTION OF TERMS OF THE ’425 PATENT

43. As I discussed above, I have been informed that for purposes of *inter-partes* reviews, the standard for claim construction of terms within the claims of the patent is the same as that applied in federal district court litigation.

44. For this declaration, I apply the plain and ordinary meaning of all claim terms, in addition to certain proposed constructions as noted below. I reserve the right to offer my opinions on any constructions or construction issues presented by Patent Owner.

45. I have been asked to apply the following proposed constructions:

“distributed computational graph” (claims 1, 4): “[a] directed graph with each transformation comprising a node and the output messages between transformations comprising edges.” EX1001, 9:33-37; see also *id.*, 9:5 (“Definitions”); FIGS. 6-9 & 12-15.

“transformation” (claims 1, 5, 6): “a function performed on zero or more streams of input data which results in a single stream of output.” EX1001, 9:53-56.

“transformation pipeline” / “pipeline” (claims 1, 5, 6): “a progression of functions which each perform a transformation on a data stream.” EX1001, 4:31-34, 9:56-66.

VI. OVERVIEW OF PRIOR ART

46. Below, I discuss the references I rely on in my declaration.

A. Barsness (EX1004)

47. U.S. Patent No. 8,732,300 (“Barsness”) issued May 20, 2014, from an application filed January 10, 2011. EX1004, Cover.

48. Barsness discloses a distributed stream-processing computing cluster comprising a plurality of compute nodes executing processing elements (“PEs”) that apply transformations to streaming data and transmit output data to downstream PEs according to an operator graph. Barsness explains that “[i]n a stream data processing application, operators are connected to one another such that data flows from one

processing element (PE) to the next” and that “[s]calability is achieved by distributing an application across nodes ... as well as replicating processing elements on multiple nodes and load balancing among them.” EX1004, 3:16-22. Barsness further describes a computing infrastructure including a management system, a monitoring system, and multiple compute nodes connected via a network, where “the operator graph ... represents a stream application beginning from one or more source processing elements (PEs) through to one or more sink PEs ... configurable to add or remove one or more PEs based on the data processing needs and scalability.” EX1004, 6:43-56. Barsness depicts these data streams generally flowing left to right, e.g., FIG. 1B shows data flowing through PEs (1-10) on respective compute nodes (130_{1-4}):

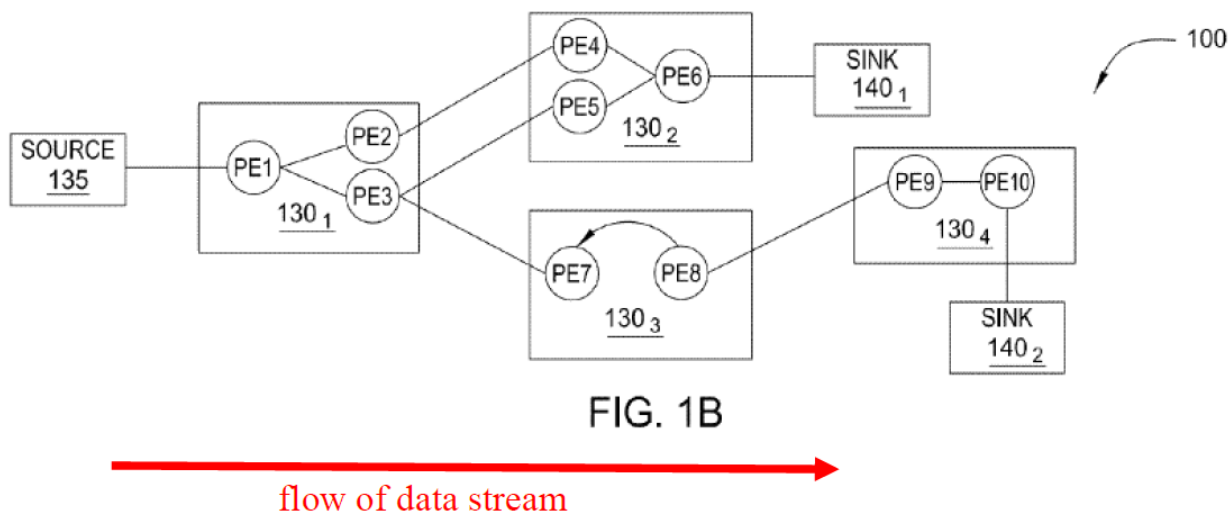


FIG. 1B

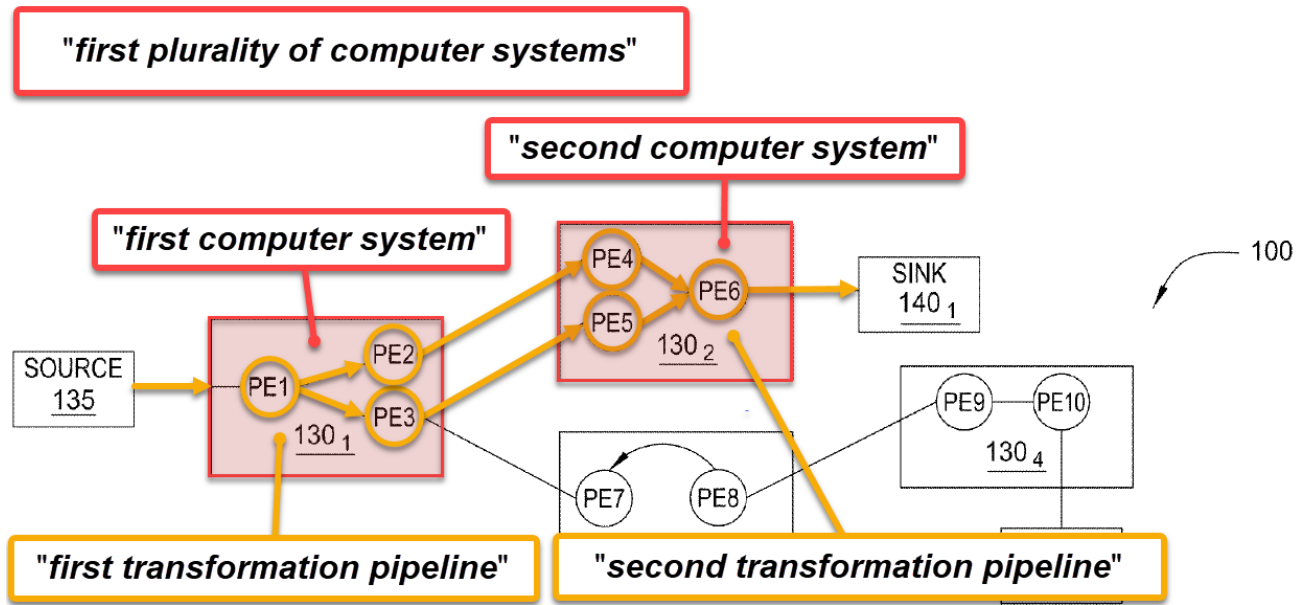
Id., FIG. 1B (red annotations added).

49. As illustrated in FIG. 1B of Barsness, upstream processing elements collectively form a first transformation pipeline (e.g., 130₁) whose output is provided as input to downstream processing elements forming a second transformation pipeline (e.g., 130₂, 130₃, 130₄). Barsness describes that “data elements flow into a source PE of a stream application and are processed by that PE,” and that output data elements subsequently flow to downstream PEs. EX1004, 6:49-56; 7:52-67. Barsness teaches that routing of pipeline output messages is governed by topology information and performed at the operator level. EX1004, 9:3-9 (“operator 240 may ... process data flowing into a processing element ... and ... emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132.”). Each compute node is a computer system comprising a processor, network interface, and memory. EX1004, 8:30-35. The memory stores processing elements and operators configured to process incoming data and to “emit data to other operators ... and to other PEs in the stream application as described by the operator graph.” EX1004, 8:66-9:20; 9:4-9. Thus, Barsness requires that operators locally determine where to send output messages based on topology-related information.

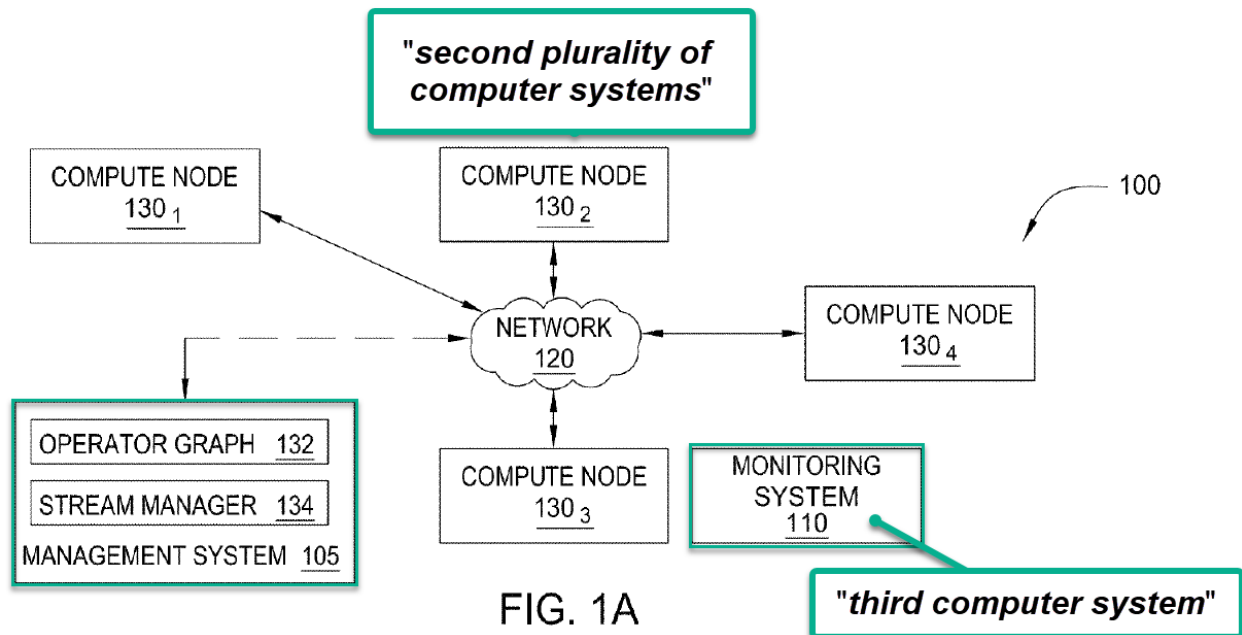
50. Barsness further emphasizes the importance of dynamic reconfiguration in stream-processing systems. Barsness teaches a monitoring system that collects processing rates for the PEs distributed across the compute nodes of a

streaming application. EX1004, 3:51-4:2. For example, Barsness teaches that a monitoring application may “increase the priority of the PE in question or even move the PE to separate machines to gain more resource,” and that processing elements may be moved or replicated to improve performance and fault tolerance. EX1004, 3:56-60, 14:26-39. Barsness also explains that the operator graph and compute nodes may be reconfigured by a user interface and management system. EX1004, 7:3-6; 10:28-39.

51. Barsness discloses the transformation of data using pipelines, and the monitoring of the transformation. Specifically, Barsness discloses transformation of data using multiple compute nodes, each of which has a pipeline of processing elements (“PEs”). Each PE applies a transformation to incoming data and transmits the output data to downstream PEs in accordance with an operator graph. EX1004, 3:16-22 (“In a stream data processing application, operators are connected to one another such that data flows from one processing element (PE) to the next (e.g., over a TCP/IP socket).”). As shown in the annotated Figure 1B below, each compute node is a *computer system* and the PEs are a *transformation pipeline*.



EX1004, FIG. 1B. Barsness also discloses computer systems for monitoring the transformation. Barsness discloses that its "monitoring system 110" (*third computer system*) stores the operator graph. EX1004, 3:45-50.



EX1004, Fig. 1A (annotations added).

52. The “monitoring system 110” uses the graph to monitor the performance of the compute nodes by collecting “tuple processing data” and calculating “current tuple processing rates” to compare with “historical tuple processing rates.” EX1004, 13:4-7. If the current processing rates fail to meet the historical rates, the monitoring system can issue “corrective actions” including “spawning new compute nodes, [and] duplicating a compute node” (*fourth computer system*). EX1004, 13:12-14; EX1004, 14:51-56.

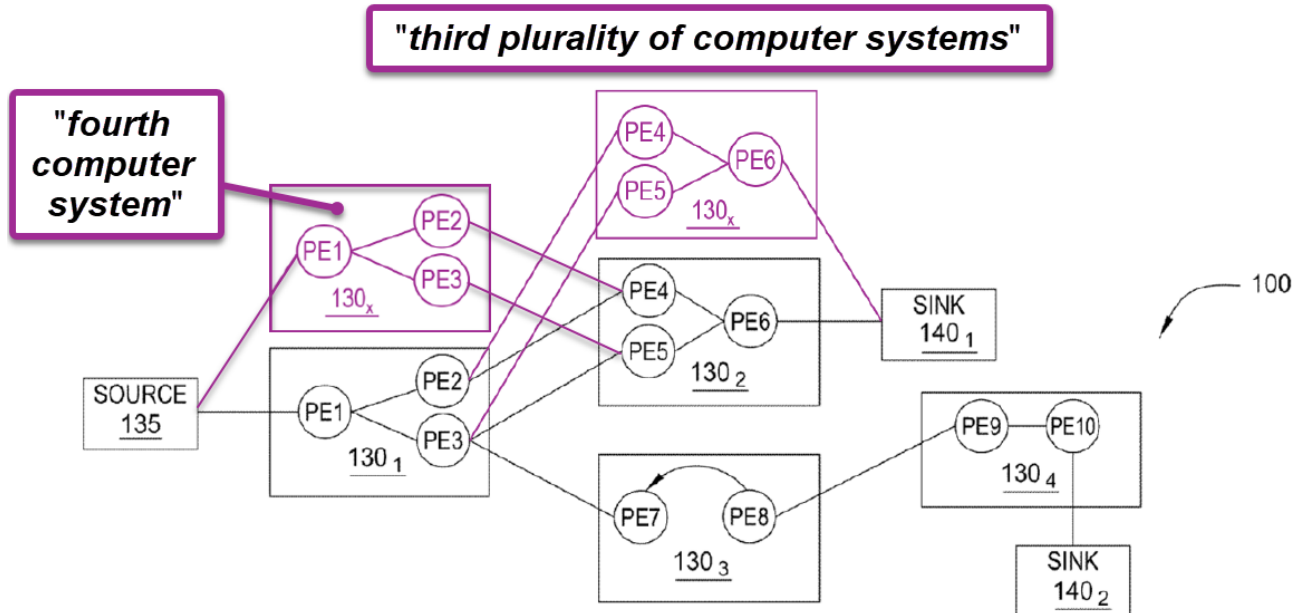


FIG. 1B

EX1004, FIG. 1B (annotations added).

B. Chakradhar (EX1005)

53. Chakradhar is a U.S. patent application publication that published September 15, 2016, from an application filed March 14, 2016, and alleges priority

to a provisional application filed on March 13, 2015 (“Chakradhar Provisional”).

EX1005, Cover. Chakradhar is assigned to NEC Laboratories America, Inc.

54. Chakradhar is directed to the same field of distributed stream processing and provides an explicit mechanism for encoding and using topology information at the operator level. Chakradhar teaches that “[e]ach operator maintains a route-map that specifies the routes on which output messages should be sent after processing incoming messages,” and provides examples in which route-map entries identify downstream operators. EX1005, [0122]. Chakradhar further explains that route-maps encode “tuple-routing information and also the topology link structure.” EX1005, [0054].

55. Chakradhar expressly ties the use of route-maps to concrete technical benefits. In particular, Chakradhar teaches that, “[b]y applying novel heuristics on the topology performance (feedback) metrics, ... the controller determines efficient route-maps,” and that these route-maps are “atomically injected into multiple operators, on-the-fly, using a light-weight, fault-tolerant protocol, for fast topology re-optimization.” EX1005, [0054]. Chakradhar further explains that operators use these injected route-maps to control routing behavior during execution, enabling topology changes without interrupting stream processing. EX1005, [0054], [0122].

1. Chakradhar Is Supported By The Chakradhar Provisional

56. As I note below, the Chakradhar Provisional supports (1) at least one claim of Chakradhar, and (2) the portions of Chakradhar relied upon in this declaration.

2. The Chakradhar Provisional Supports Claim 1 Of Chakradhar

57. The chart below indicates disclosure in the Chakradhar Provisional that provided written description support for each element of claim 1 of Chakradhar.

Claim 1 of Chakradhar (EX1005)	Written Description Support in Chakradhar Provisional (EX1006)
A stream processing acceleration method employing network-aware routing, said method comprising the computer implemented steps of:	EX1006, 4/22: “Stream processing applications...” ... “Stream processing engines...used to process continuous streams of data...” <i>Id.</i>, 5/22: “...a per-topology controller, which uses novel methods and protocols for dynamic, network-aware routing and on-the-fly topology modifications.”... “These new route-maps are atomically injected into multiple operators, onthe- fly, using a light-weight, fault-tolerant protocol, for fast topology re-optimization.” <i>Id.</i>, 13/22: “...the proposed MWD approach outperforms the random binary tree by a significant margin (more than two-fold speedup), for different model sizes.” <i>Id.</i>, 14/22: “...we focus on heuristics and protocols for fast topology re-

	optimization, to maintain high stream-throughput at all times.” <i>Id.</i> , 14/22: “...high-throughout stream processing...”
representing topology link structures using route-maps	<i>Id.</i> , 5/22: “(i) representing topology link structure using route-maps”
wherein said route-maps include tuple-routing information and topology structure encoded therein;	<i>Id.</i> , 5/22: “...the controller determines efficient route-maps, which encode tuple-routing information and also the topology link structure.”
applying, on-the-fly, the route maps into multiple operators using a topology route-map update method.	<i>Id.</i> , 5/22: “...(iii) a light-weight, fault-tolerant protocol for atomic route-map update.” ... “These new route-maps are atomically injected into multiple operators, on-the-fly, using a lightweight, fault-tolerant protocol, for fast topology re-optimization.” <i>Id.</i> , 10/22: “...all operators start using the new route-maps for the same batch.”

3. The Chakradhar Provisional Supports Relied-Upon Subject Matter Of Chakradhar

58. As demonstrated above, the Chakradhar Provisional discloses operators maintaining route-maps that include tuple-routing information and topology structure for directing output messages, i.e., the same technique disclosed in Chakradhar. In the discussion below, I also include citations to the Chakradhar Provisional supporting Chakradhar’s relied-upon disclosure in the mapping of

Barsness and Chakradhar to the challenged claims, which demonstrates that the Chakradhar Provisional supports all the relied upon-disclosure in Chakradhar.

C. Siripurapu (EX1007)

59. Siripurapu is a U.S. patent application publication that published February 27, 2014, and alleges priority to a provisional application filed November 18, 2010. EX1007, Cover.

60. Siripurapu is directed to the same field of processing real-time data streams with a “distributed, elastic architecture” and provides an explicit mechanism for handling “additional data streams” and “increased data stream flows.” EX1007, [0060], [0453]; *id.*, [0359]-[0363]. Specifically, Siripurapu teaches that the “distributed implementation provides scalability, wherein one or more additional worker nodes may be added ... to accommodate increased computational requirements, additional data streams, increased data stream flows, and the like.” *Id.*, [0453]. Siripurapu explains that “[a]t any point in time, there are usually several Streams: some of them external (e.g., Twitter, Facebook) and others that are generated as intermediate values and outputs by the application.” EX1007, [0178]; *id.*, [0439]-[0440] (describing various “[i]nput real-time data streams”).

61. Further, “[t]he distributed implementation also provides fault-tolerance so that there is no single point of failure, wherein a task at a failed worker node may be assigned to a different worker node without affecting the entire distributed

system.” EX1007, [0454]. Siripurapu also discloses assigning a stream to another compute node if the computational load of a compute node is too high, for example due to “hot spots in the flow of incoming stream events.” EX1007, [0470].

62. As depicted in Figure 36, multiple “real-time data streams” arrive at a stream bus and are distributed to worker nodes by a conductor that manages the realtime processing of stream data by “act[ing] as a task scheduling mechanism to provide instructions to designate worker nodes for performing ... operations.” EX1007, [0444], [0446]; id., [0448]-[0449] (explaining how the conductor monitors “performance of the worker nodes” and “perform[s] load balancing by reassigning tasks to the worker nodes”). For example, “[t]ask scheduling may be performed by the conductor in a manner to balance the loads among the worker nodes, prevent ‘hot spots’ from arising at any worker node and to reduce the response time required by each worker node to generate an output.” EX1007, [0481].

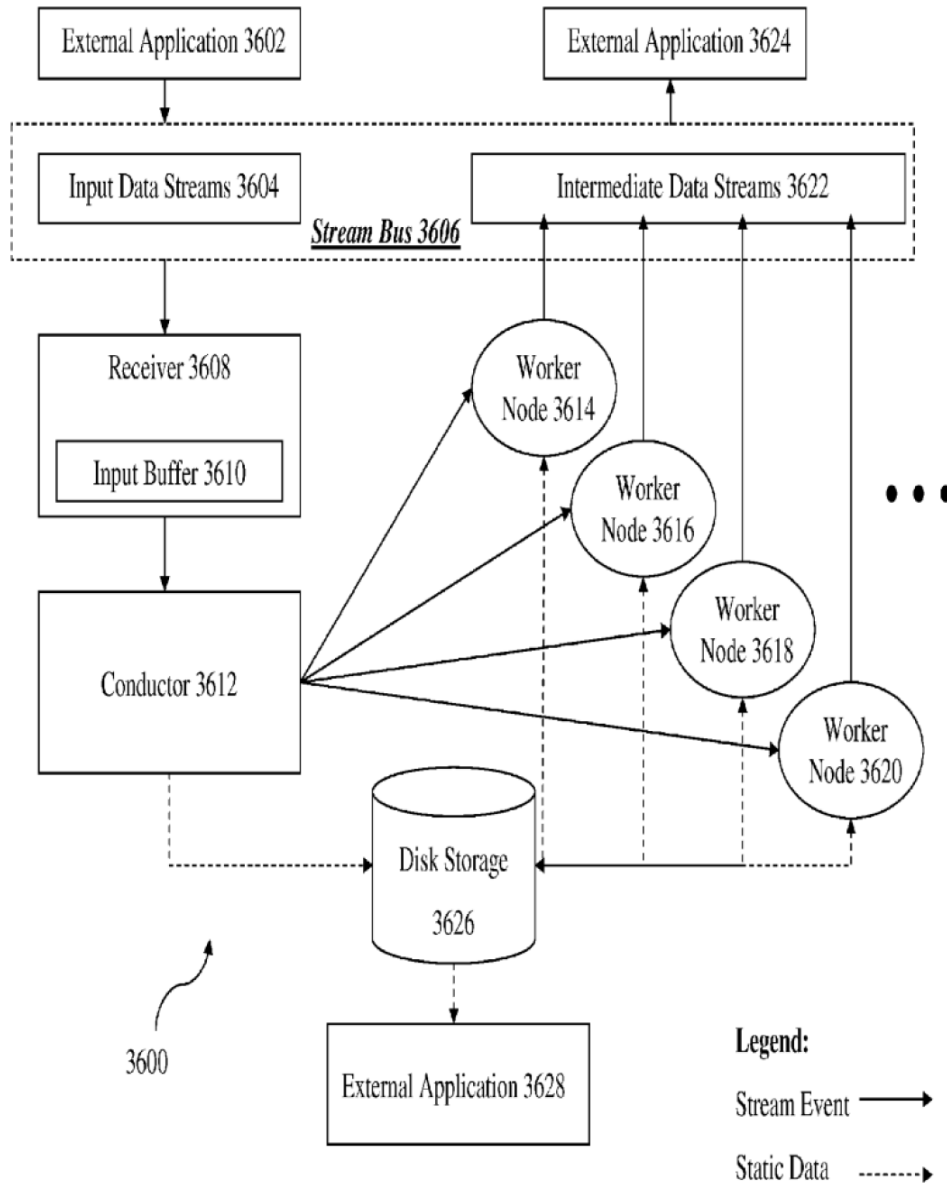


Figure 36

EX1007, FIG. 36.

VII. SUMMARY OF OPINIONS ON UNPATENTABILITY

63. In the analysis that follows, I identify the following combinations of prior art that, in my opinion, anticipate and/or render obvious the '425 patent claims:

Ground	Basis	Reference(s)	Claims
1	§103	Barsness	1-6
2	§103	Barsness and Chakradhar	1-6
3	§103	Barsness and Siripurapu	1-6
4	§103	Barsness, Chakradhar, and Siripurapu	1-6

A. Ground 1: Claims 1-6 are Rendered Obvious by Barsness

64. As explained below, claims 1-6 are obvious over Barsness.

1. Claim 1

(a) 1[pre] – A system comprising:

65. In my opinion, Barsness discloses the subject matter of the preamble. To the extent the preamble is limiting, Barsness discloses *a system*. Specifically, Barsness describes a “stream data processing application” implemented on a distributed computing infrastructure, in which “operators are connected to one another such that data flows from one processing element (PE) to the next,” and the application is distributed “across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them.” EX1004, 3:16-28.

66. In my opinion, a POSITA would have understood that the architecture Barsness describes, namely, interconnected processing elements distributed across compute nodes through which streaming data flows for processing, is *a system* within the meaning of the preamble. *See also* EX1004, 7:52-67 (describing the

operator graph and the compute nodes on which processing elements execute); EX1004, FIG. 1A, 1B (depicting compute nodes 130₁–130₄ executing processing elements PE1–PE10 connected in an operator graph 132 through which data tuples flow).

67. This understanding is consistent with the conventional architecture of distributed stream processing systems known to a POSITA at the relevant time. For example, Toshniwal described Storm, a contemporaneous distributed stream processing engine, as comprising “topologies” through which “streams of tuples” flow, where “[a] topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components.” EX1011, 148 (“The basic Storm data processing architecture consists of streams of tuples flowing through topologies. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components.”); *see also* EX1014, 250 (“A stream processing application is organized as a *graph*, formed by a set of *operators* connected to each other by *streams*.”); EX1010, 2 (“Stream engines code the application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology.”). Barsness’s stream data processing application reflects the same architectural paradigm.

- (b) 1[a] – a distributed computing network comprising a first plurality of computer systems, a second plurality of computer systems, and a third plurality of computer systems,

68. In my opinion, Barsness discloses *a distributed computing network* as recited in the limitation. Barsness discloses a “computing infrastructure 100” that includes “a management system 105 and a plurality of compute nodes 130₁₋₄, each connected to a communications network 120,” and monitoring system 110. EX1004, 6:43-48; 7:15-26 (“A monitoring system 110 may also be included in the computing infrastructure 100 to monitor data processing components of the stream application.”). Figure 1A of Barsness depicts this infrastructure, showing the management system 105, monitoring system 110, communications network 120, and the four compute nodes 130₁–130₄:

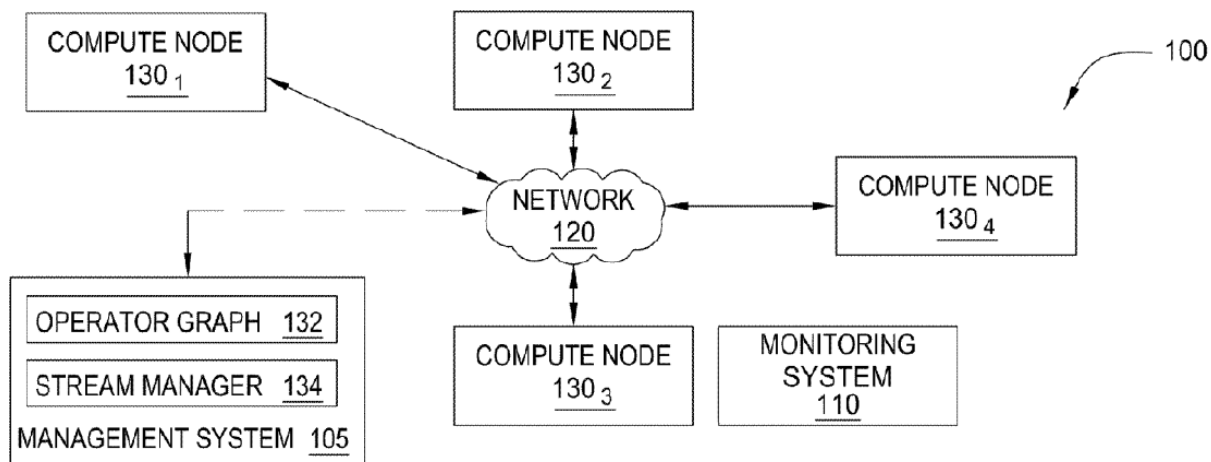


FIG. 1A

EX1004, FIG. 1A.

69. Barsness characterizes each compute node as “a distributed computing node.” EX1004, 2:35-37. Barsness further discloses that the monitoring system 110 of the computing infrastructure “may be a cluster of a plurality of computing resources, which may be distributed in the network 120.” EX1004, 7:17-21. Barsness also describes a “stream manager 134” residing on the management system that “control[s] a stream application running on the compute nodes,” including by “moving processing elements (PEs) from one compute node 130 to another, for example, to manage the processing loads of the compute nodes 130 in the computing infrastructure 100.” EX1004, 7:27-37.

70. In my opinion, a POSITA would have understood that the computing infrastructure Barsness describes is a *distributed computing network* as that term would be understood. The infrastructure has a plurality of compute nodes that are physically distinct and physically distributed; those nodes, together with the management system 105 and the monitoring system 110, are interconnected by communications network 120; and the components communicate over that network to perform coordinated processing of the streaming data. Each of those features is an attribute of a *distributed computing network*. See, e.g., EX1009, 4 (“[T]he term ‘real-time processing’ in this context means that the system is able to respond within seconds or sub-second to requests, while ‘distributed’ means that it is running on multiple computers.”); EX1012, 170 (describing S4 as a “distributed, scalable ...

platform” designed to “run in large clusters built with commodity hardware”); EX1011, 147 (describing Storm as a “real-time fault-tolerant and distributed stream data processing system” designed to operate on “large clusters”).

71. “*first plurality of computer systems*,”: In my opinion, Barsness discloses *a first plurality of computer systems* as recited in the limitation. Barsness discloses that its computing infrastructure includes “a plurality of compute nodes 130₁₋₄.” EX1004, 6:45-49. Figures 1A and 1B of Barsness depict these compute nodes:

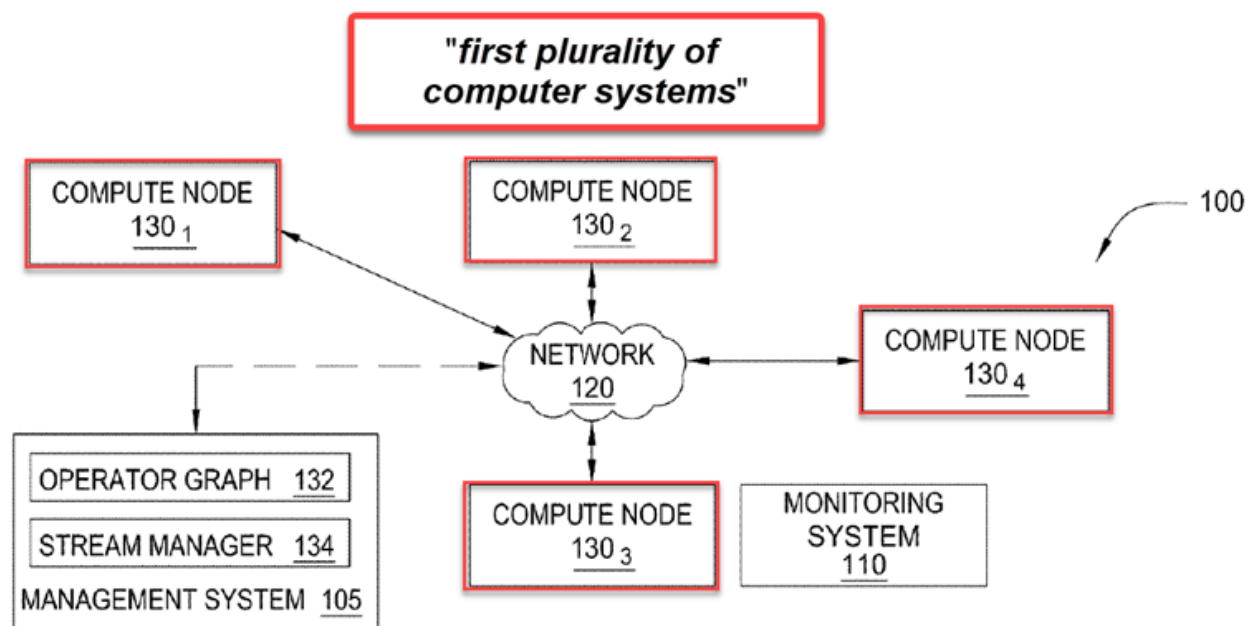


FIG. 1A

EX1004, FIG. 1A (annotations added).

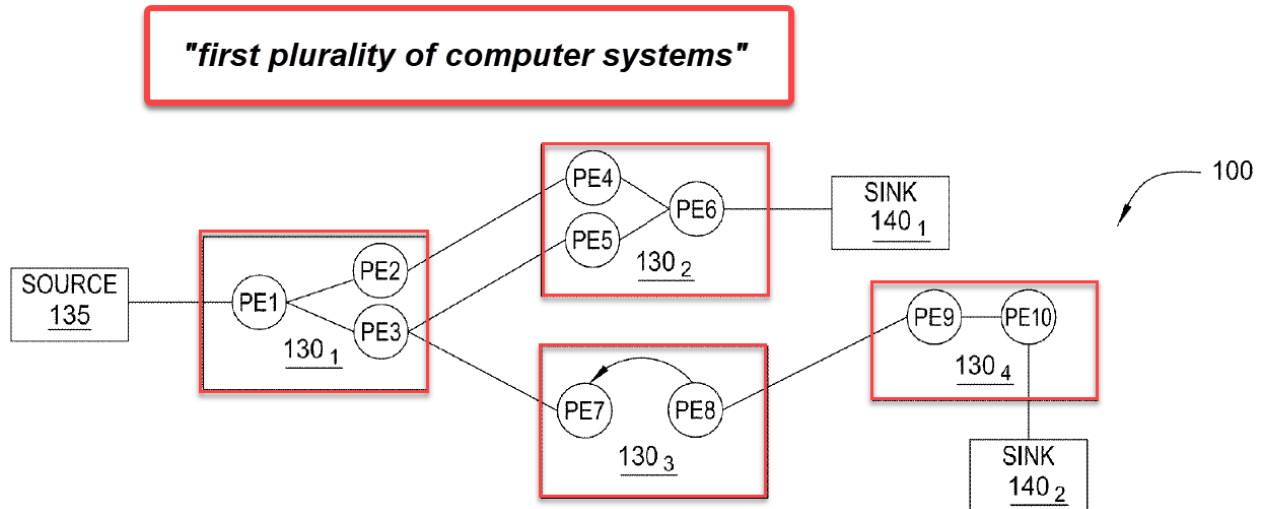


FIG. 1B

EX1004, FIG. 1B (annotations added).

72. In my opinion, each of Barsness's compute nodes is a "computer system" as a POSITA would have understood that term. Barsness describes each compute node as comprising the standard hardware components of a computer system, including "a central processing unit (CPU) 205, a network interface 215, an interconnect 220, a memory 225, and storage 230," together with "an I/O devices interface 210 used to connect I/O devices 212." EX1004, 8:30-39. The components Barsness describes are the same kinds of components the '425 patent itself identifies as comprising a computer system. *Compare* EX1004, 8:30-39, *with* EX1001, 14:65-15:25. Figure 2 of Barsness depicts these components in detail:

"computer system[s] of the first plurality of computer systems"

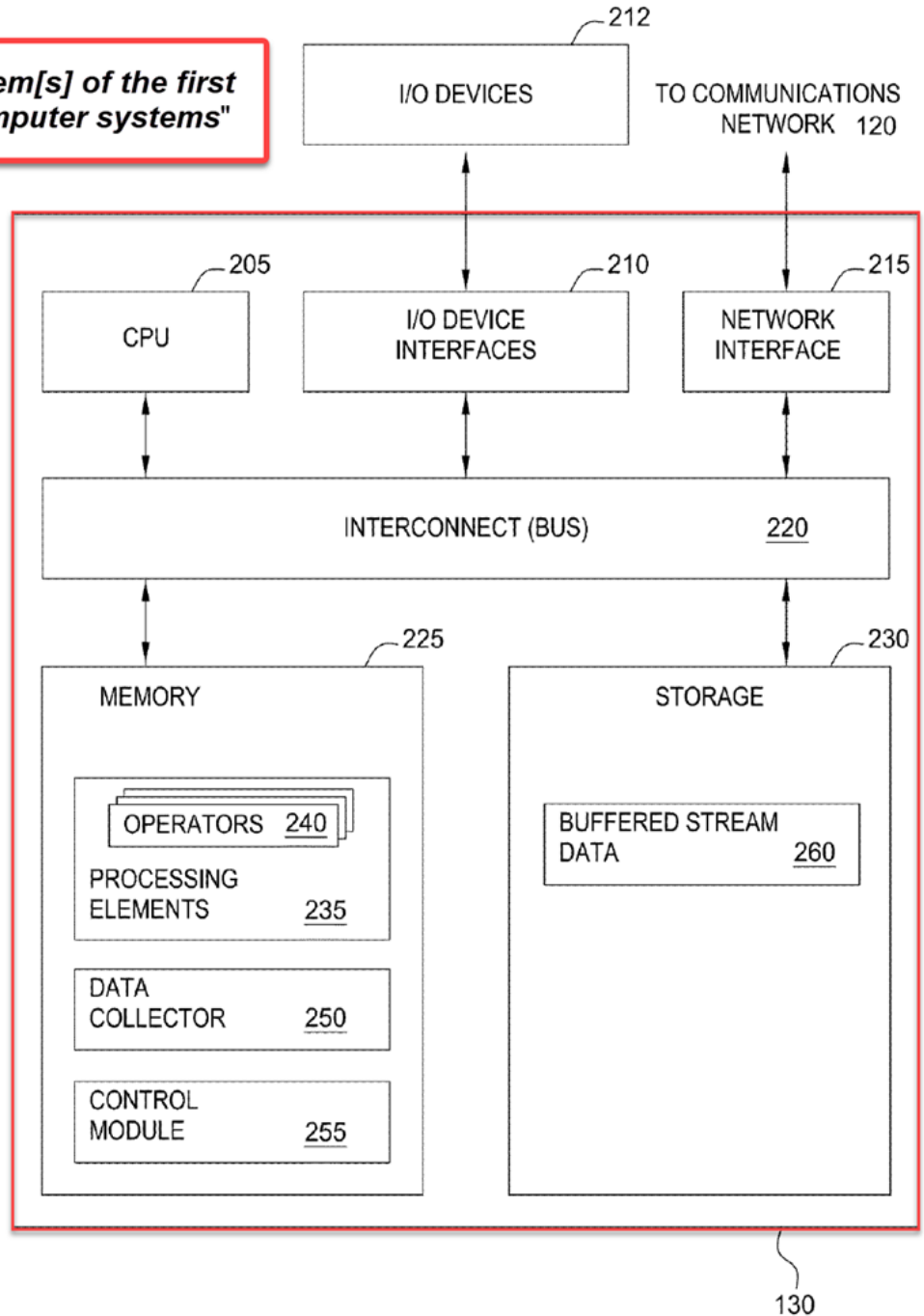


FIG. 2

EX1004, FIG. 2 (annotations added); *id.*, 8:53-65 (further describing the compute node hardware).

73. Barsness confirms that each compute node is itself a complete, independently operating computer system. Barsness states that “[t]he compute node 130 may be a physical host or a virtual machine,” that the compute node “provides a software abstraction of computing resources of a physical host computer,” and that the compute node “runs as an independent computer system.” EX1004, 8:37-44. In my opinion, a POSITA would have understood from this disclosure that each of Barsness’s compute nodes 130₁ through 130₄ constitutes a separate, independently operating computer system, and that the four compute nodes together constitute *a first plurality of computer systems*. This aligns with the state of the art at the time. See e.g., EX1010, 2 (“To process continuous streams of data at low latency,” distributed stream processing engines such as Borealis, TimeStream, Samza, and Storm distribute computation across multiple machines including “cloud environments”); EX1011, 148 (describing the Storm cluster architecture as comprising multiple worker nodes, each running independently and connected through the cluster).

74. “*second plurality of computer systems*,”: In my opinion, Barsness discloses *a second plurality of computer systems* as recited in the limitation. Barsness discloses that its computing infrastructure includes, in addition to the first plurality of compute nodes 130₁₋₄ discussed above, a “management system 105” and a “monitoring system 110.” EX1004, 6:45-49; *id.* at 7:15-17. Figure 1A of Barsness

depicts the management system 105 and the monitoring system 110 as separate components within the computing infrastructure:

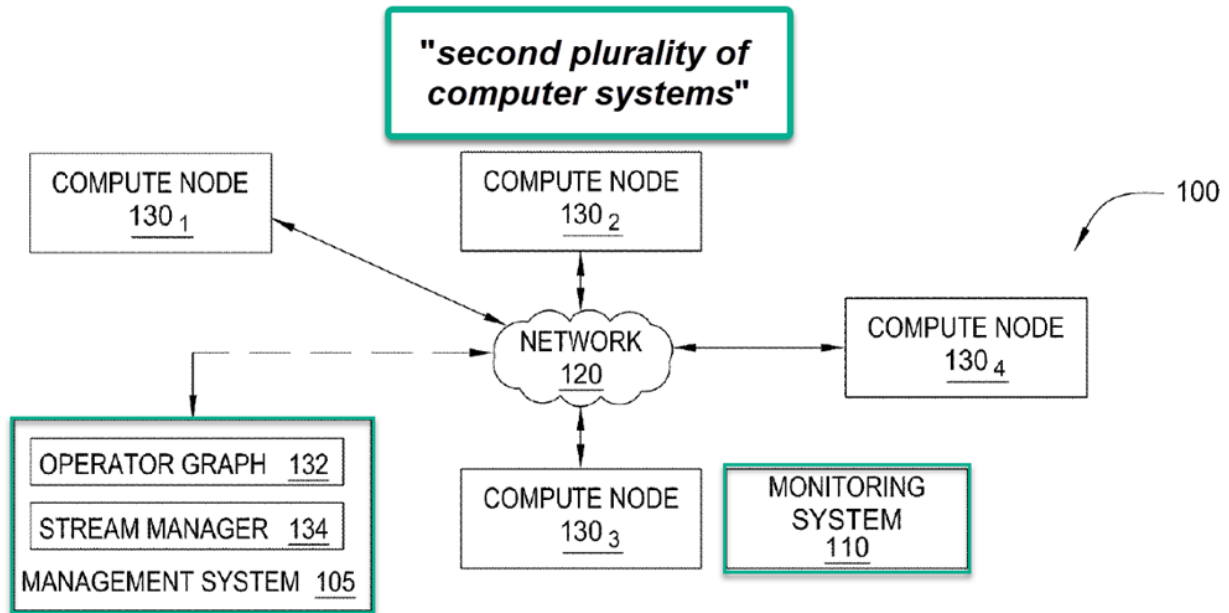


FIG. 1A

EX1004, FIG. 1A (annotations added). In my opinion, the management system 105 and the monitoring system 110 together constitute *a second plurality of computer systems* within the meaning of the limitation, separate and distinct from the first plurality of compute nodes 130₁₋₄.

75. The management system 105 is itself a *computer system* as a POSITA would have understood that term. Barsness describes the management system 105 as comprising the standard hardware components of a computer system, including “a central processing unit (CPU) 305, a network interface 315, an interconnect 320, a memory 325, and storage 330.” EX1004, 9:51-59. Figure 3 of Barsness depicts these components in detail:

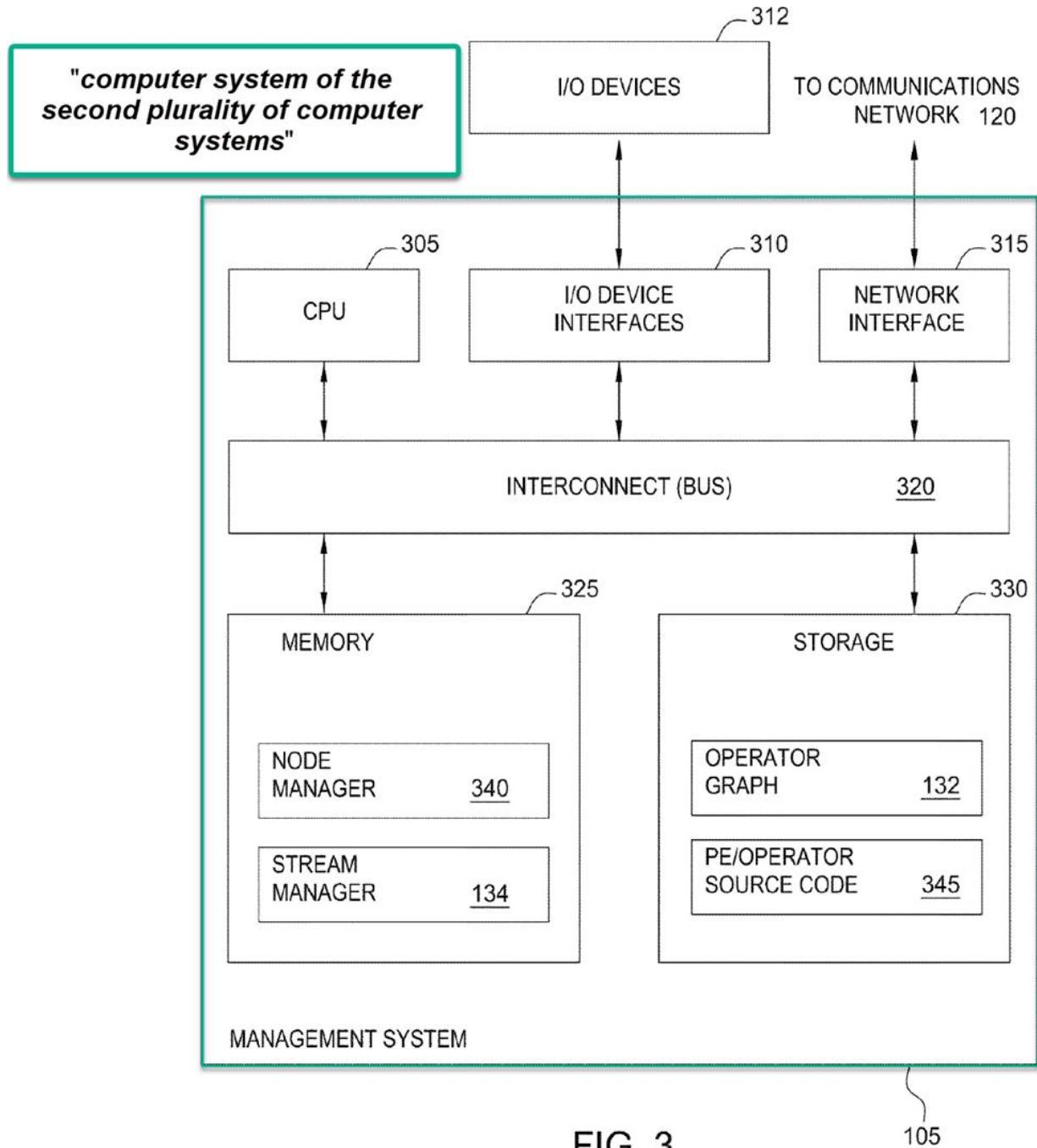


FIG. 3

EX1004, FIG. 3 (annotations added); *see also id.* at 9:51-10:64 (further describing the management system hardware and operation).

76. The monitoring system 110 is likewise a computer system as a POSITA would have understood that term. Figure 4 of Barsness depicts the monitoring system 110 with the same kinds of computer-system components, including a CPU, memory, storage, and network interface. EX1004, FIG. 4. Barsness further describes that the monitoring system 110 “may be a cluster of a plurality of computing resources, which may be distributed in the network 120.” EX1004, 7:17-21. In my opinion, a POSITA would have understood that a “cluster of a plurality of computing resources” is itself a computer system, or a set of computer systems, that operates as part of the *second plurality of computer systems*.

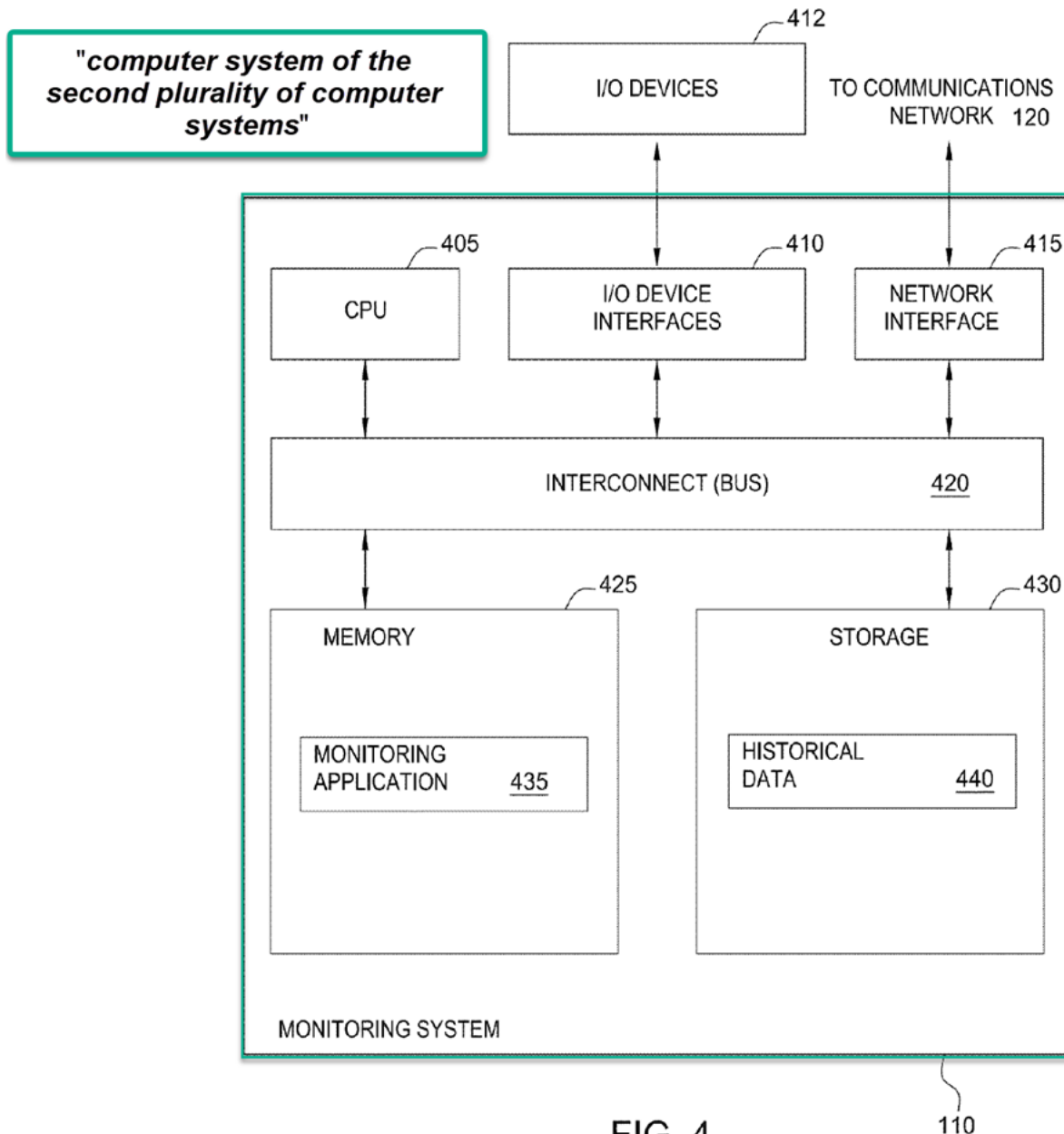


FIG. 4

EX1004, Fig. 4 (annotations added), 10:65-11:37.

77. *“third plurality of computer systems”*: In my opinion, Barsness discloses *a third plurality of computer systems* as recited in the limitation. As I discuss in greater detail below in connection with limitations 1[f], 1[f][i], and

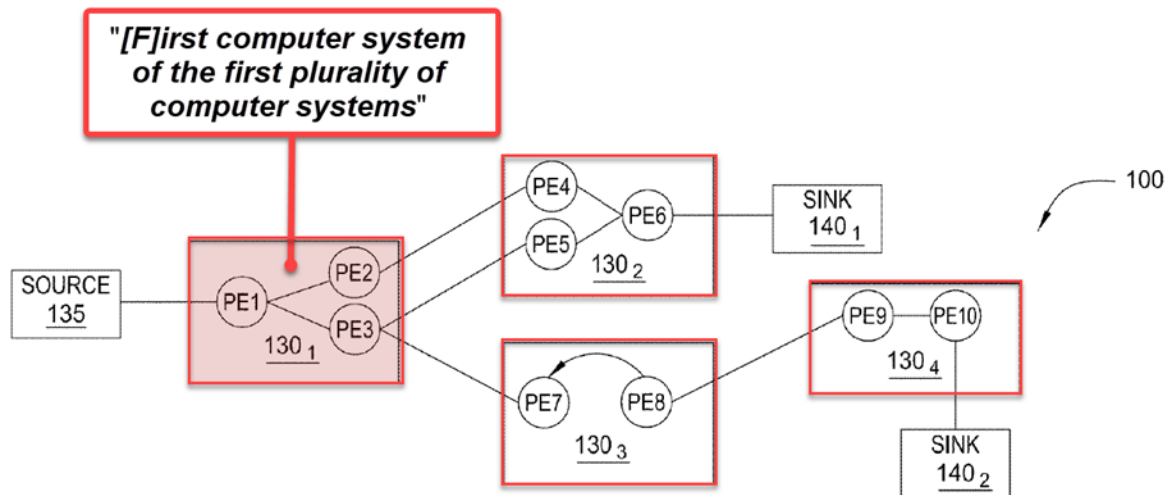
1[f][ii], Barsness discloses that the monitoring system 110 is configured, in response to its monitoring of the stream application, to cause additional compute nodes 130_x to be added to the computing infrastructure to improve the performance of the stream application. *See* EX1004, 14:26-56 (describing corrective actions including “spawning new compute nodes, [and] duplicating a compute node”); *id.* at 10:26-39 (describing the use of new compute nodes “in parallel” to process the same type of data to increase data throughput). Barsness further discloses that the compute nodes “may be virtual machines” and that the system can “spawn more virtual machines on the fly.” EX1004, 8:30-52. Barsness also describes the use of “cloud computing infrastructure” for “the provision of scalable computing resources,” such that “the nodes used to create a stream application may be virtual machines hosted by a cloud service provider.” EX1004, 6:16-42.

78. In my opinion, a POSITA would have understood from these disclosures that Barsness contemplates additional compute nodes that are available, beyond the four compute nodes 130₁₋₄ that constitute the *first plurality*, to be spawned, duplicated, or otherwise added to the computing infrastructure on demand by the monitoring system 110. *See, e.g.*, EX1004, 6:19-26 (“cloud computing may be defined as a computing capability that provides an abstraction between the computing resource and its underlying technical architecture ... enabling convenient on-demand network access to a shared pool of configurable computing resources

that can be rapidly provisioned and released with minimal management effort or service provider interaction.”). These additional compute nodes 130_x are themselves computer systems for the same reasons set forth above with respect to the first plurality and they constitute *a third plurality of computer systems* as the limitation recites, separate and distinct from the *first plurality* of compute nodes 130₁₋₄ and from the *second plurality* comprising the management system 105 and the monitoring system 110.

(c) **1[b] – wherein a first computer system of the first plurality of computer systems is configured to execute first software instructions that:**

79. In my opinion, Barsness discloses *a first computer system of the first plurality of computer systems* as recited in the limitation. As I discussed above in connection with the first plurality, the first plurality of computer systems comprises Barsness’s compute nodes 130₁₋₄. In my opinion, compute node 130₁ is *a first computer system* of that first plurality. Figure 1B of Barsness depicts compute node 130₁ as part of the first plurality of computer systems within the operator graph 132:



EX1004, FIG. 1B (annotations added); *see also id.* at 7:53-55 (“Compute node 130₁ includes source PE1 along with PE2 and PE3.”); *id.* at 8:30-34 (“FIG. 2 is a more detailed view of the compute node 130 of FIGS. 1A-1B ... the compute node 130 includes, without limitation, a central processing unit (CPU) 205, a network interface 215, an interconnect 220, a memory 225, and storage 230.”); *id.* at FIG. 2 (depicting the hardware components of a compute node 130).

80. “*configured to execute first software instructions*”: In my opinion, Barsness discloses this limitation. Barsness teaches that each compute node executes the executable code for its processing elements (PEs) using the compute node’s CPU and memory. Barsness states: “The CPU 205 retrieves and executes programming instructions stored in the memory 225.” EX1004, 8:53-55. Barsness further states that “the memory 225 includes one or more processing elements (PEs) 235.” EX1004, 8:66-9:1; *see also id.* at 8:66-9:14 (further describing the contents of

memory 225, including PEs, the data collector module, and the control module); *id.* at 7:13-15 (describing the use of a user interface to “load programming instructions and configurations to be executed by the compute nodes 130₁₋₄”).

81. In my opinion, a POSITA would have understood that each compute node 130 is configured, by way of its CPU and memory, to execute the software instructions corresponding to the PEs that reside in that compute node’s memory. The PEs themselves include the executable code that performs the processing. Barsness states that each PE is configured to “carry out any form of data processing functions on a received N-tuple, including, e.g., writing to database tables or performing other database operations such as data joins, splits, reads, etc., as well as performing other data analytic functions or operations.” EX1004, 6:65-7:3. Each PE in turn comprises one or more “operators 240,” each of which is “a small chunk of executable code configured to process data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132.” EX1004, 9:4-9; *see also id.* at 3:19-22 (“Scalability is achieved by distributing an application across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them.”). Figure 2 of Barsness depicts the CPU 205, memory 225, and PEs 235 within a compute node:

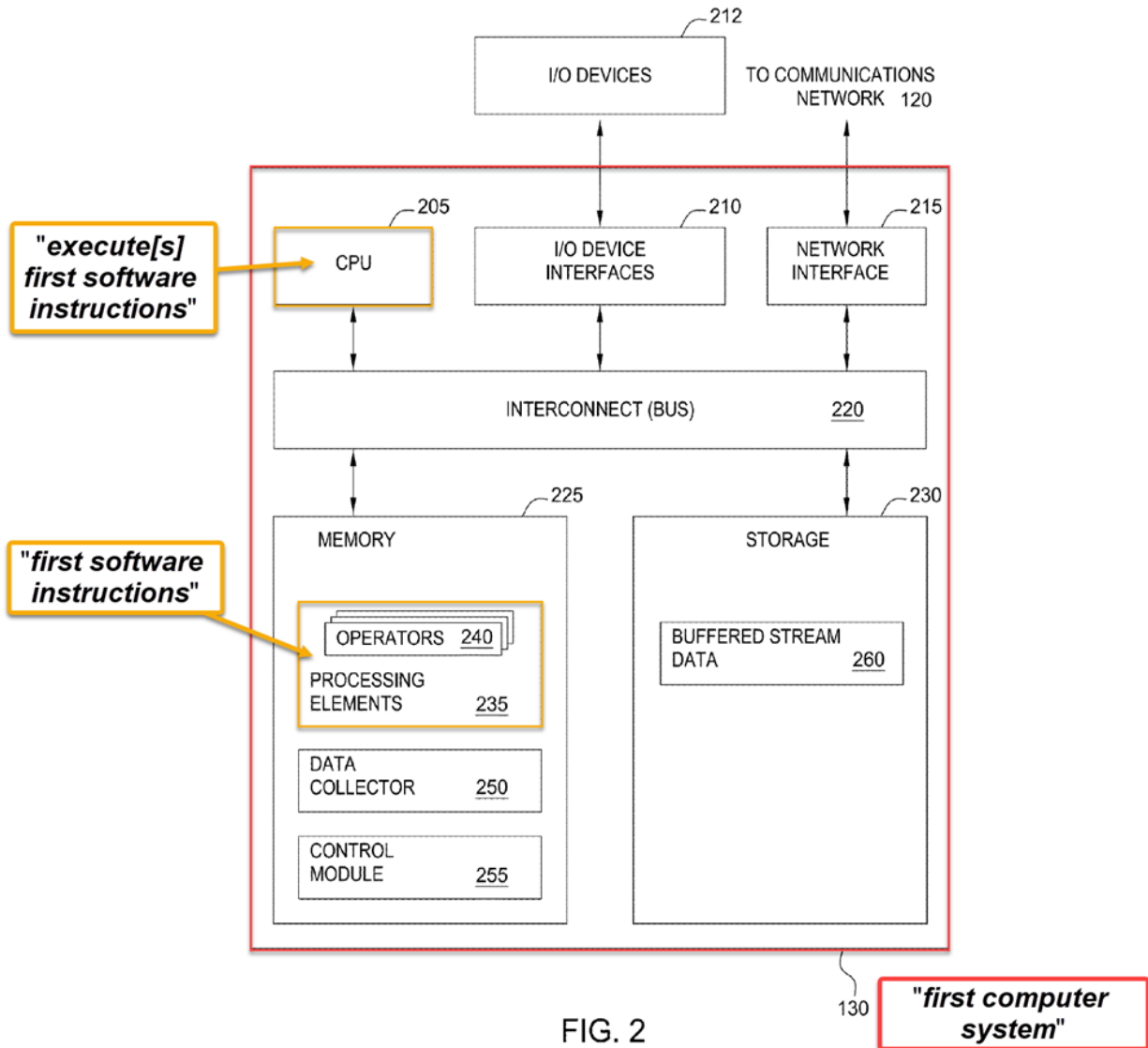


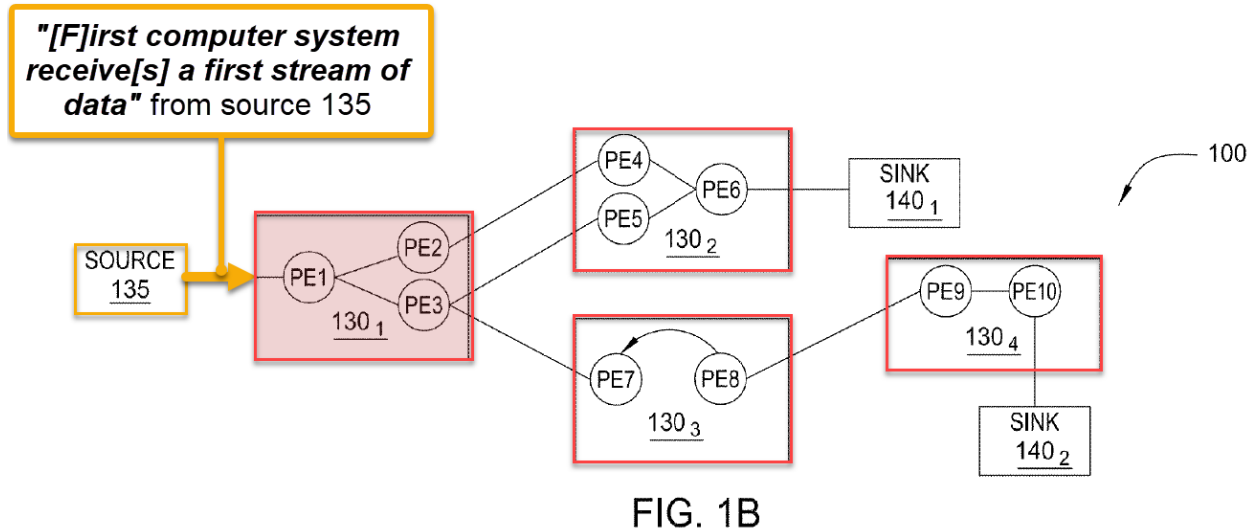
FIG. 2

EX1004, FIG. 2 (annotations added).

(d) 1[b][i] – receive a first stream of data,

82. In my opinion, Barsness discloses this limitation. Barsness describes that compute node 130_i contains the source processing element of the stream application and that the “data elements” of the stream (*a first stream of data*) flow into that source processing element. Barsness states that “the operator graph 132

represents a stream application beginning from of one or more source processing elements (PEs) through to one or more sink PEs,” and that “[d]ata elements flow into a source PE of a stream application and are processed by that PE.” EX1004, 6:49-56. Barsness further explains that the source PE is housed within compute node 130₁ (*a first computer system*): “Compute node 130₁ includes source PE1 along with PE2 and PE3.” EX1004, 7:54-55. Figure 1B of Barsness depicts compute node 130₁ containing the source PE 135 (labeled as PE1):



EX1004, FIG. 1B (annotations added). In my opinion, a POSITA would have understood that compute node 130₁, by virtue of containing the source PE that receives the data elements of the stream, *receive[s] a first stream of data* within the meaning of the limitation. See also EX1004, 6:57-65 (describing the N-tuple data received by processing elements from the stream).

83. Barsness further confirms that compute node 130_i is configured to *receive a first stream of data* by describing a buffered stream data area within each compute node. Barsness states that “[b]uffered stream data 260 represents a storage space for data flowing into the compute node 105 from upstream processing elements (or from a data source for the stream application).” EX1004, 9:43-45. Figure 2 of Barsness depicts buffered stream data 260 within the compute node 130:

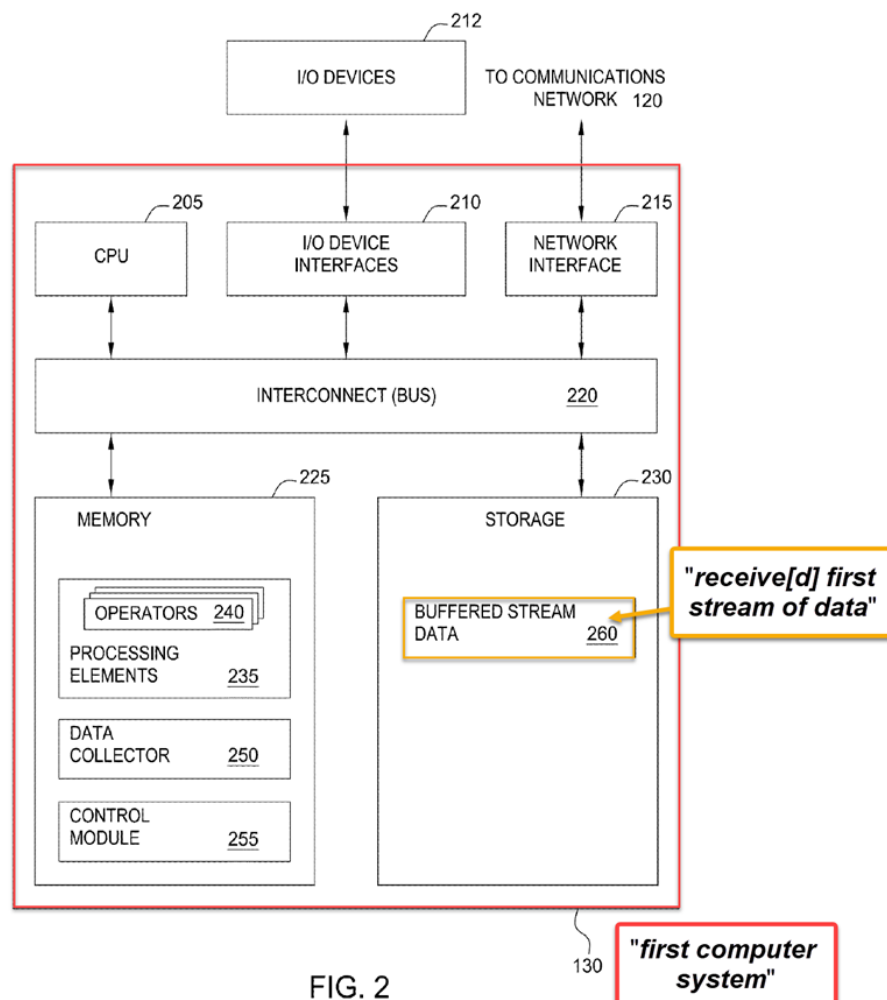


FIG. 2

EX1004, FIG. 2 (annotations added); *see also id.* at 1:22-24 (“products are available which allow users to create applications that process and query streaming data before
-49-

it reaches a database file.”); *id.* at 3:16-18 (“In a stream data processing application, operators are connected to one another such that data flows from one processing element (PE) to the next.”); *id.* at 10:20-26 (“The processing elements 235 and operators 240 setup may include configuring each processing element and operator to process particular type of streaming data attributes, accept the streaming data in a particular format and emit the streaming data in another particular format, which may be the same or different from the data format of incoming data tuples.”). In my opinion, a POSITA would have understood that the buffered stream data 260 storage space within compute node 130₁ is configured to *receive* the incoming *first stream of data* from the data source.

- (e) **1[b][ii] – process first portions of the first stream of data by applying a first transformation pipeline to the first portions of the first stream of data to generate first pipeline output messages, and**

84. In my opinion, Barsness discloses this limitation. Compute node 130₁’s PEs collectively process the data elements that flow in from source 135 and produce output tuples that propagate to other PEs downstream. Figure 1B of Barsness depicts compute node 130₁ and the flow of data elements through PE1, PE2, and PE3:

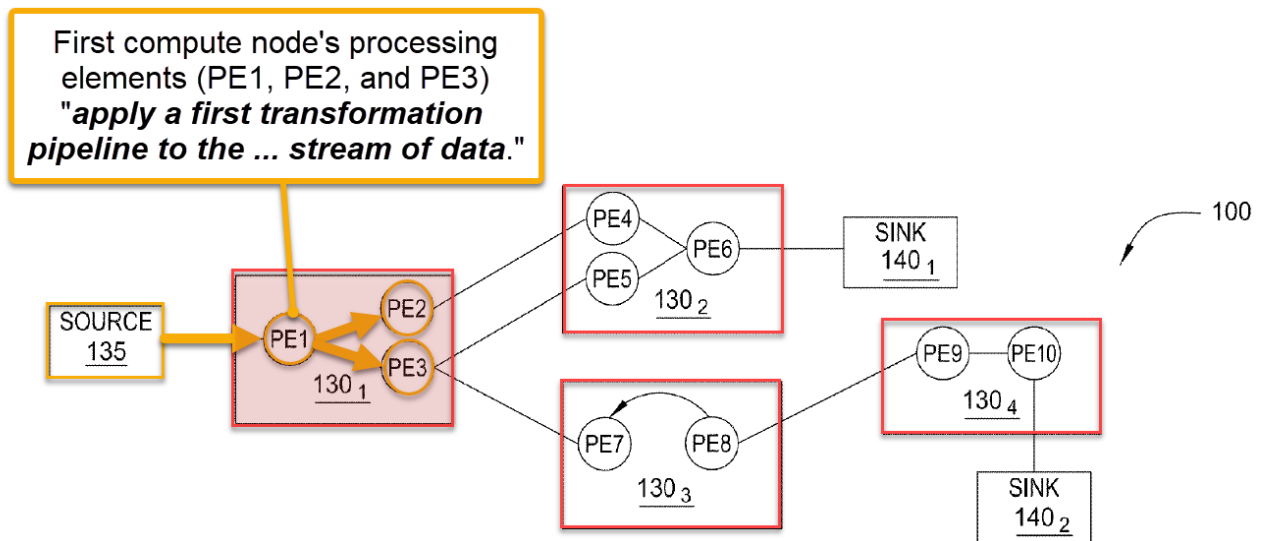


FIG. 1B

EX1004, FIG. 1B (annotations added).

85. “*process first portions of the first stream of data*”: In my opinion, the data elements arriving at PE1 from source 135 are the *first stream of data*, and the subset of those data elements that PE1, PE2, and PE3 consume and process within compute node 130₁ are the *first portions of the first stream of data*. I previously addressed PE1’s receipt of those data elements in connection with limitation 1[b][i].

86. “*by applying a first transformation pipeline to the first portions of the first stream of data*”: In my opinion, the three PEs residing on compute node 130₁, namely PE1, PE2, and PE3, operate together as a *first transformation pipeline* that is applied to those *first portions of the first stream of data*. PE1, the source PE, processes the incoming data elements and, as Barsness states, “emits tuples received by PE2 and PE3.” EX1004, 7:54-57; *see also id.*, 6:55-56 (“Data elements flow into

-51-

a source PE of a stream application and are processed by that PE.”); *id.*, 6:57-60 (“processing elements...emit an N-tuple of data elements into the stream”). PE2 and PE3 then process those incoming tuples and, in turn, emit further tuples downstream. EX1004, 7:57-67. The result is a sequenced flow of operations: data elements enter PE1, PE1 transforms them and routes them onward, and PE2 and PE3 each apply their own transformations before emitting their outputs. In my opinion, that sequenced flow is the architectural hallmark of a pipeline, and it constitutes the ***first transformation pipeline***. This pipelined flow is also consistent with the conventional design of distributed stream processing systems known to a POSITA at the relevant time. *See* EX1010, 2 (“Tuples are processed in a pipelined fashion as they traverse through the topology.”); EX1011, 148 (“The basic Storm data processing architecture consists of streams of tuples flowing through topologies. A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components.”); EX1014, 250 (describing operator-graph architecture in which “[a] stream processing application is organized as a *graph*, formed by a set of *operators* connected to each other by *streams*”). In my opinion, a POSITA would have understood that the ***first transformation pipeline*** therefore comprises:

- (1) PE1 processing of data elements into tuples,
- (2) the receipt of those tuples by PE2 and PE3, and

(3) the processing of those tuples by PE2 and PE3.

87. Under Petitioner's proposed construction of *transformation pipeline*, a *progression of functions which each perform a transformation on a data stream*, the series of PEs running on compute node 130₁ readily satisfies the limitation. Barsness teaches that, in a stream data processing application, "operators are connected to one another such that data flows from one processing element (PE) to the next." EX1004, 3:16-18. As the data flows from one PE to the next, each PE performs a data processing function on the data through its operators and emits the transformed result to the next PE. EX1004, 6:65-7:3 ("each processing element may be configured to carry out any form of data processing functions on a received N-tuple, including, e.g., writing to database tables or performing other database operations such as data joins, splits, reads, etc., as well as performing other data analytic functions or operations."); *id.*, 6:55-56 ("Data elements flow into a source PE of a stream application and are processed by that PE."); *id.*, 3:16-22 ("In a stream data processing application, operators are connected to one another such that data flows from one processing element (PE) to the next (e.g., over a TCP/IP socket). Scalability is achieved by distributing an application across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them."); *id.*, 9:4-9 ("Each operator 240 may provide a small chunk of executable code configured to process

data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132.”). The function each PE performs *transforms* the data: “the N-tuple received by a processing element need not be the same N-tuple sent downstream” (EX1004, 6:60-62), and “PEs may be configured to *transform* incoming data tuples by either adding or removing one or more data attributes before emitting outgoing data tuples to other PEs” (EX1004, 8:11-15 (emphasis added)). Barsness’s Figure 1B example makes this concrete: PE1 “may split data elements received in a tuple and pass some data elements to PE2 [and] others to PE3,” a *transformation* that produces output tuples and routes them from the input. EX1004, 7:55-57. The technical principle reflected in these passages, namely that streaming operators implement input-changing functions chained together to form a set of such functions, was conventional and well known to a POSITA at the relevant time. *See* EX1014, 250 (“[A]n operator can perform processing and produce data items on its output ports. Streams connect output ports of operators to input ports of other operators using FIFO semantics.”); EX1010, 2 (describing stream engines as coding “the application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology”); EX1013, 271-273 (describing a “data processing model” comprising a “graph” of operators in which each operator transforms its inputs). In my opinion, the series of PEs within compute node 130i accordingly comprises a

-54-

progression of functions that each perform some action or transformation on the data stream.

88. The same conclusion follows under the Patent Owner's asserted plain and ordinary meaning of ***transformation pipeline***. The specification's own illustrative transformation pipelines, for example those depicted at FIG. 7 of the '425 patent, are themselves *a progression of functions that each perform some action or transformation on the data stream*. See, e.g., EX1001, 17:54-18:36, FIG. 7. The Barsness PEs operating on compute node 130₁ have precisely that structure: each PE performs a ***transformation***, and the PEs together form a ***transformation pipeline*** of sequenced ***transformations*** on the stream. EX1004, 6:65-7:3; *id.*, 6:55-56; *id.*, 3:16-22; *id.*, 9:4-9.

89. Barsness's description of PE ***process[ing]*** is broad: "each processing element may be configured to carry out any form of data processing functions on a received N-tuple, including, ... data analytic functions or operations." EX1004, 6:65-7:3. The processing is implemented through operators, which Barsness describes as "executable code configured to process data flowing into a processing element." EX1004, 9:3-9. And the processing changes the data: PEs "transform incoming data tuples by either adding or removing one or more data attributes before emitting outgoing data tuples to other PEs." EX1004, 8:11-15. In my opinion, this

kind of in-PE data transformation is what Barsness's PEs do when they execute their assigned operators on incoming tuples.

90. “*to generate first pipeline output messages*”: In my opinion, the tuples that PE2 and PE3 emit at the downstream boundary of compute node 130₁ are the *first pipeline output messages* the *first transformation pipeline* produces. Figure 1B of Barsness depicts the data tuples emitted by PE2 and PE3:

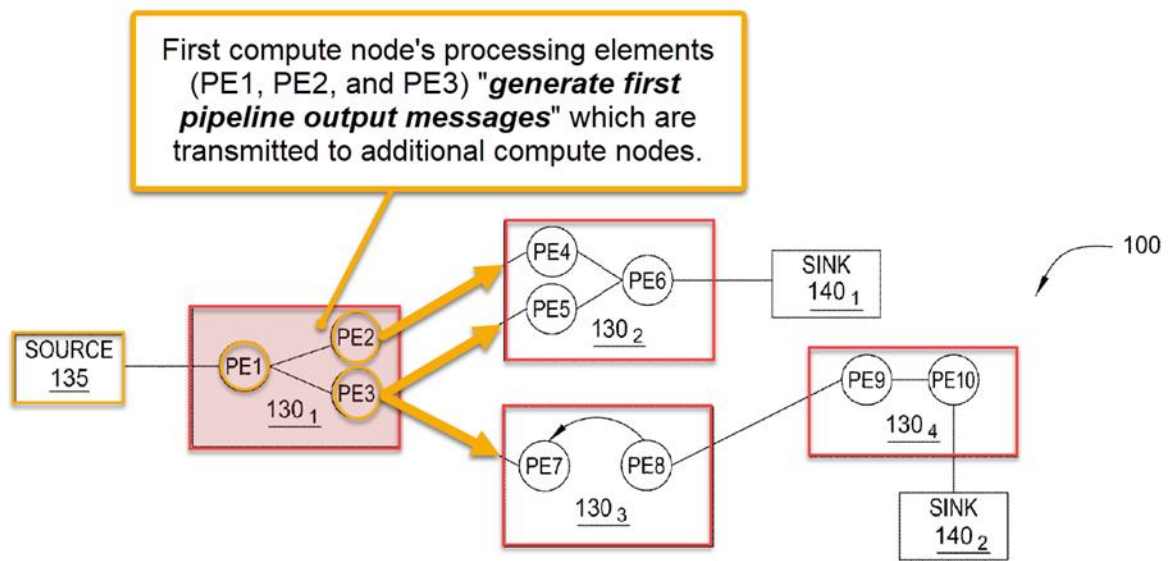


FIG. 1B

EX1004, FIG. 1B (annotations added); *see also id.*, 6:62-65 (“the processing elements could be configured to receive or emit data in formats other than an N-tuple (e.g., the processing elements could exchange data marked up as XML documents).”); *id.*, 7:52-67; *id.*, 9:4-9 (“Each operator 240 may provide a small chunk of executable code configured to process data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other

PEs in the stream application as described by the operator graph 132.”). Barsness states more generally that “processing elements...emit an N-tuple of data elements into the stream.” EX1004, 6:57-60.

(f) 1[b][iii] – transmit the first pipeline output messages;

91. In my opinion, Barsness discloses this limitation. Compute node 130₁ *transmit[s]* the data tuples (*the first pipeline output messages*) emitted by PE2 and PE3 to compute node 130₂ and compute node 130₃. Barsness states: “Data that flows to PE2 results in tuples emitted to PE4 on compute node 130₂.” EX1004, 7:57-58. Barsness further states: “This example operator graph also shows data tuples flowing from PE3 to PE7 on compute node 130₃.” EX1004, 7:64-65. Figure 1B of Barsness depicts these transmissions as edges running from compute node 130₁’s PE2 and PE3 to compute node 130₂’s PE4 and PE5 and to compute node 130₃’s PE7:

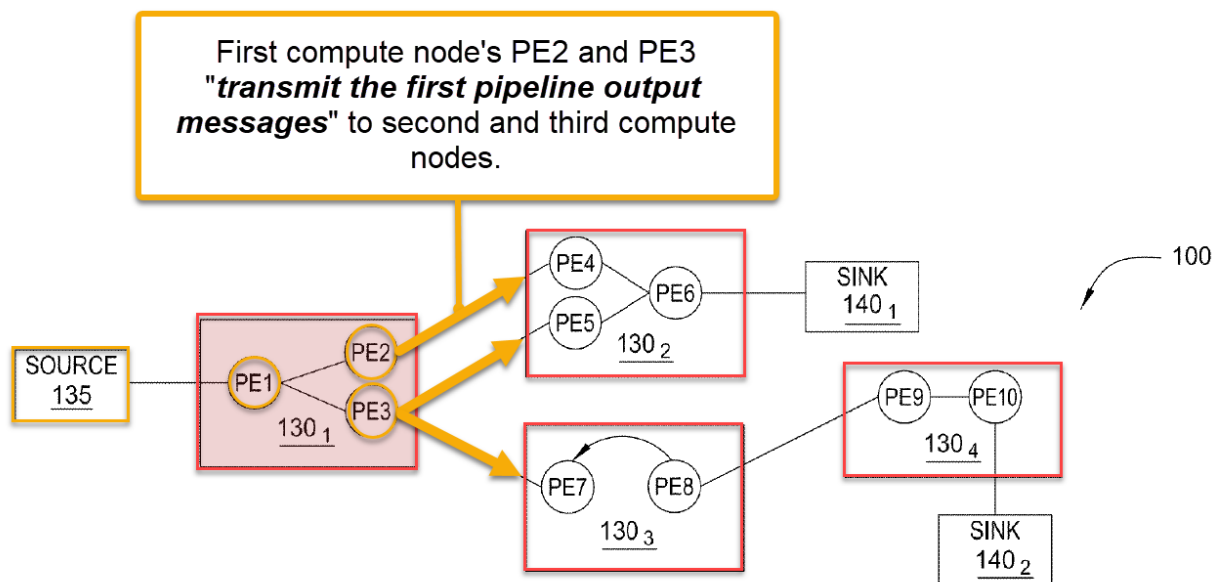


FIG. 1B

EX1004, FIG. 1B (annotations added). Further, Barsness discloses the “data flow from one processing element (PE) to the next” as using “a TCP/IP socket.” EX1004, 3:16-18. A TCP/IP socket (Transport Control Protocol / Internet Protocol) “provide[s] a tool for computer applications to communicate with other applications across an IP network.” EX1017, 1:26-54. Barsness’s reference to TCP/IP sockets for “data flow[] from one processing element (PE) to the next” indicates a transmission of that data from one compute node 130 to another compute node 130 over a communications network, allowing for “[s]calability ... by distributing an application across nodes.” EX1004, 3:19-22. In my opinion, the cross-compute node 130 communication Barsness describes is the *transmit[ting] of first pipeline output messages*. This kind of inter-node tuple transmission was a conventional feature of distributed stream processing systems known to a POSITA at the relevant time. See EX1011, 150 (describing Storm worker processes communicating tuples across nodes through dedicated send and receive threads); EX1014, 250 (“Streams connect output ports of operators to input ports of other operators.”); EX1012, 172 (describing event routing among processing elements distributed across processing nodes in S4).

- (g) **1[c] – wherein a second computer system of the first plurality of computer systems is configured to execute second software instructions that:**

92. In my opinion, Barsness discloses *a second computer system of the first plurality of computer systems* as recited in the limitation. Figure 1B of Barsness depicts compute node 130₂ as part of the first plurality of computer systems within the operator graph 132:

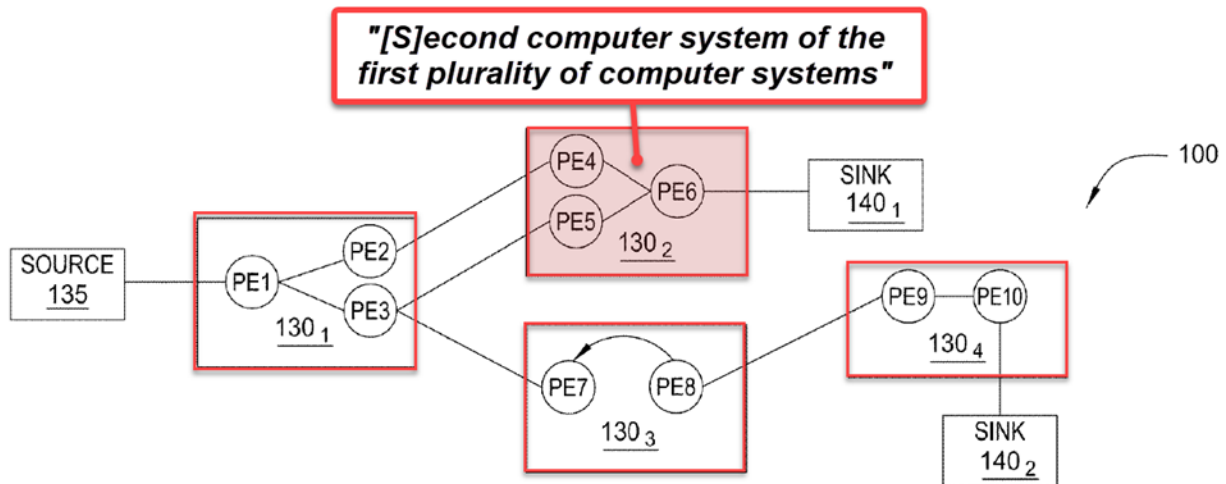


FIG. 1B

EX1004, FIG. 1B (annotations added). As I discussed earlier in connection with limitation 1[a], compute node 130₂ is one of the four compute nodes 130₁₋₄ that constitute *the first plurality of computer systems*. Compute node 130₂ has the same hardware architecture as compute node 130₁ that I addressed in connection with limitation 1[b], and it is independently operating in the same manner. EX1004, 8:30-34 (“As shown, the compute node 130 includes, without limitation, a central processing unit (CPU) 205, a network interface 215, an interconnect 220, a memory 225, and storage 230.”); *id.*, 8:43-44 (“a virtual machine runs as an independent computer system”).

93. In my opinion, compute node 130₂ is *configured to execute second software instructions* through the operators of PE4, PE5, and PE6, which reside on compute node 130₂. As I discussed earlier in connection with limitation 1[b], each of compute nodes 130₁₋₄ contains PEs and uses its CPU and memory to execute the executable code corresponding to those PEs. The same mechanism applies to compute node 130₂: its CPU executes the code for PE4, PE5, and PE6 in the same manner that compute node 130₁'s CPU executes the code for PE1, PE2, and PE3. EX1004, 8:53-55 ("The CPU 205 retrieves and executes programming instructions stored in the memory 225."); *id.*, 8:66-9:1 ("the memory 225 includes one or more processing elements (PEs) 235"); *id.*, 9:4-9 ("Each operator 240 may provide a small chunk of executable code configured to process data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132."). The code that compute node 130₂ executes through PE4, PE5, and PE6 is the *second software instructions*.

(h) 1[c][i] – process first portions of the first pipeline output messages by applying a second transformation pipeline to the first portions of the first pipeline output messages;

94. In my opinion, Barsness discloses this limitation. Compute node 130₂ (*the second computer system*) processes the data tuples it receives from compute

node 130₁ (*the first pipeline output messages*) using PE4, PE5, and PE6, which together constitute *the second transformation pipeline*.

95. “*first portions of the first pipeline output messages*”: In my opinion, the data tuples that PE2 and PE3 of compute node 130₁ emit, and that compute node 130₂ receives, are the *first portions of the first pipeline output messages* the limitation describes. Barsness states: “Data that flows to PE2 results in tuples emitted to PE4 on compute node 130₂.” EX1004, 7:57-58. Barsness further confirms the cross-node flow that delivers the upstream output to compute node 130₂: tuples from compute node 130₁’s PE2 and PE3 reach compute node 130₂’s PE4 and PE5, which then continue the processing within compute node 130₂. EX1004, 7:57-67. Figure 1B of Barsness depicts the operator graph’s flow of output data tuples from compute node 130₁’s PE1, PE2, and PE3 (*the first transformation pipeline*) to compute node 130₂’s PE4, PE5, and PE6 (*the second transformation pipeline*):

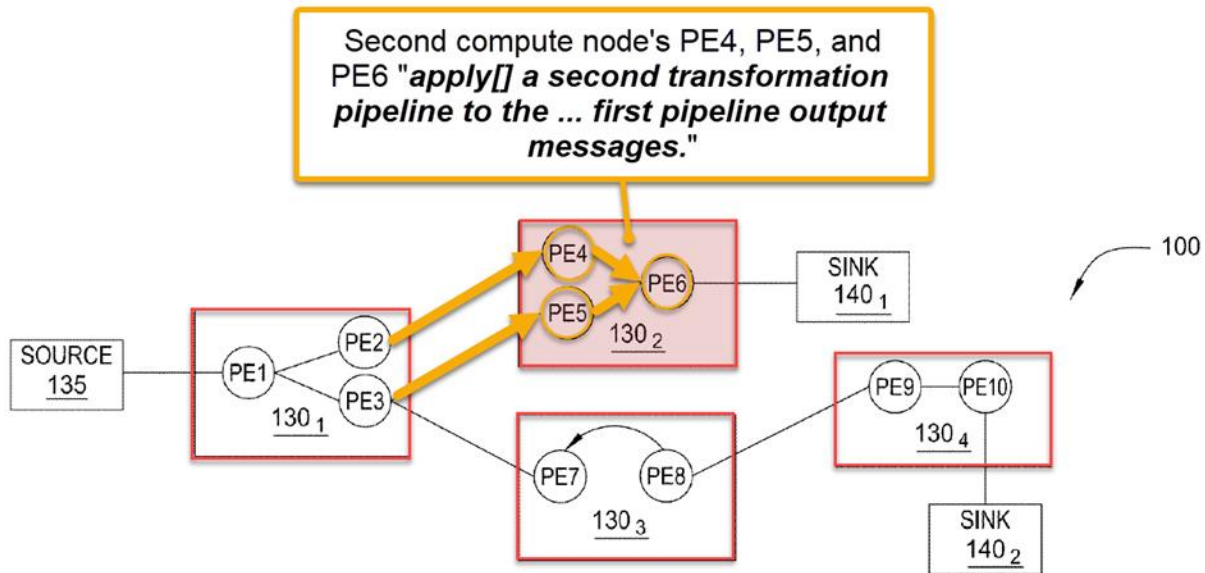


FIG. 1B

EX1004, FIG. 1B (annotations added); EX1004, 7:58-61, EX1004, 7:44-47.

96. “*second transformation pipeline*”: In my opinion, Barsness discloses this limitation. PE4, PE5, and PE6 of compute node 130₂ process the data tuples received from compute node 130₁. Barsness states: “And data tuples emitted by PE4 flow to sink PE6 140₁. Similarly, data tuples flowing from PE3 to PE5 also reach sink PE6 140₁. Thus, in addition to being a sink for this example operator graph, PE6 could be configured to perform a join operation for combining tuples received from PE4 and PE5.” EX1004, 7:59-64. As I discussed earlier in connection with limitation 1[b][ii], each PE comprises a “collection of operators 240” comprised of executable code “configured to process data flowing into a processing element” and to “carry out any form of data processing functions on a received N-tuple.” EX1004, 9:3-9; *id.*, 6:65-7:3; *id.*, 8:11-15. PE4, PE5, and PE6 operate the same way on compute

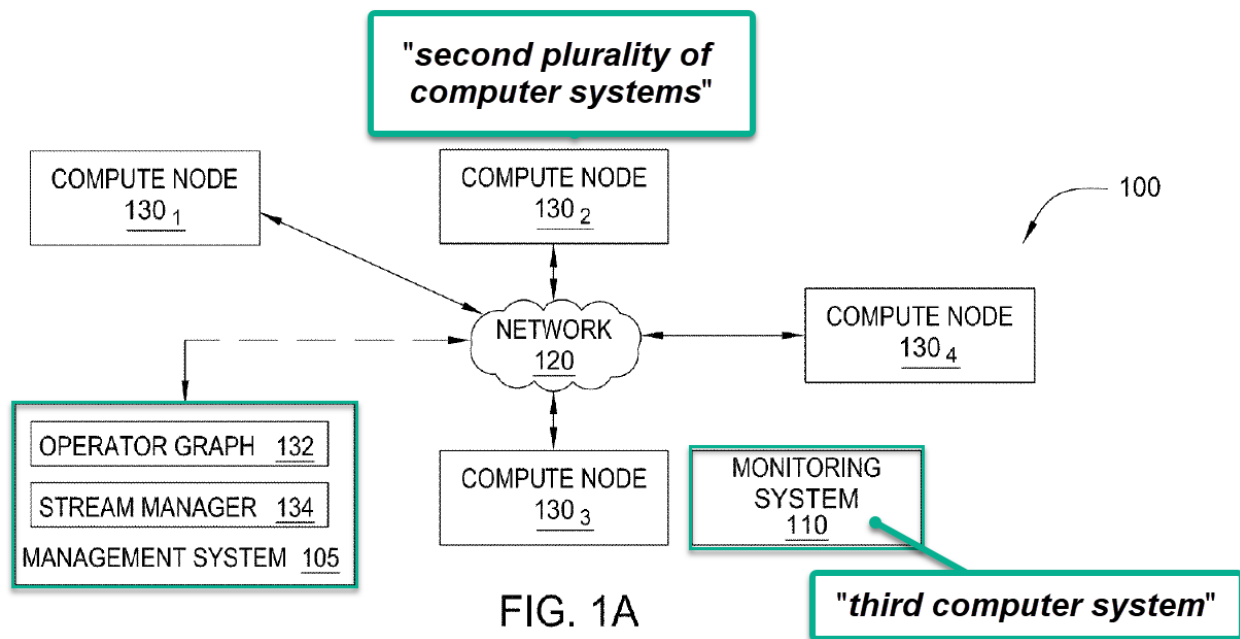
node 130₂ as PE1, PE2, and PE3 operate on compute node 130₁: each PE performs a data processing function on incoming tuples through its operators, and the data flows from one PE to the next within the compute node.

97. In my opinion, the series of PE4, PE5, and PE6 within compute node 130₂ accordingly comprises a *second transformation pipeline* under both Petitioner's proposed construction and the Patent Owner's asserted plain and ordinary meaning of the term. Under Petitioner's proposed construction, *a progression of functions which each perform a transformation on a data stream*, PE4, PE5, and PE6 satisfy the construction for the same reasons I explained earlier in connection with PE1, PE2, and PE3 on compute node 130₁: each PE performs a transforming data processing function via its operators, and the PEs are chained together to form a sequence of such functions over the stream. The Patent Owner's asserted plain and ordinary meaning, which I understand to encompass the specification's illustrative transformation pipelines (*see, e.g.*, EX1001, 17:54-18:36, FIG. 7), is satisfied for the same reasons. The Barsness disclosures cited in connection with limitation 1[b][ii] establish each component of that proposition. EX1004, 6:65-7:3; *id.*, 6:55-56; *id.*, 3:16-22; *id.*, 9:4-9.

- (i) **1[d] – wherein a third computer system of the second plurality of computer systems comprises a memory that stores a first data,**

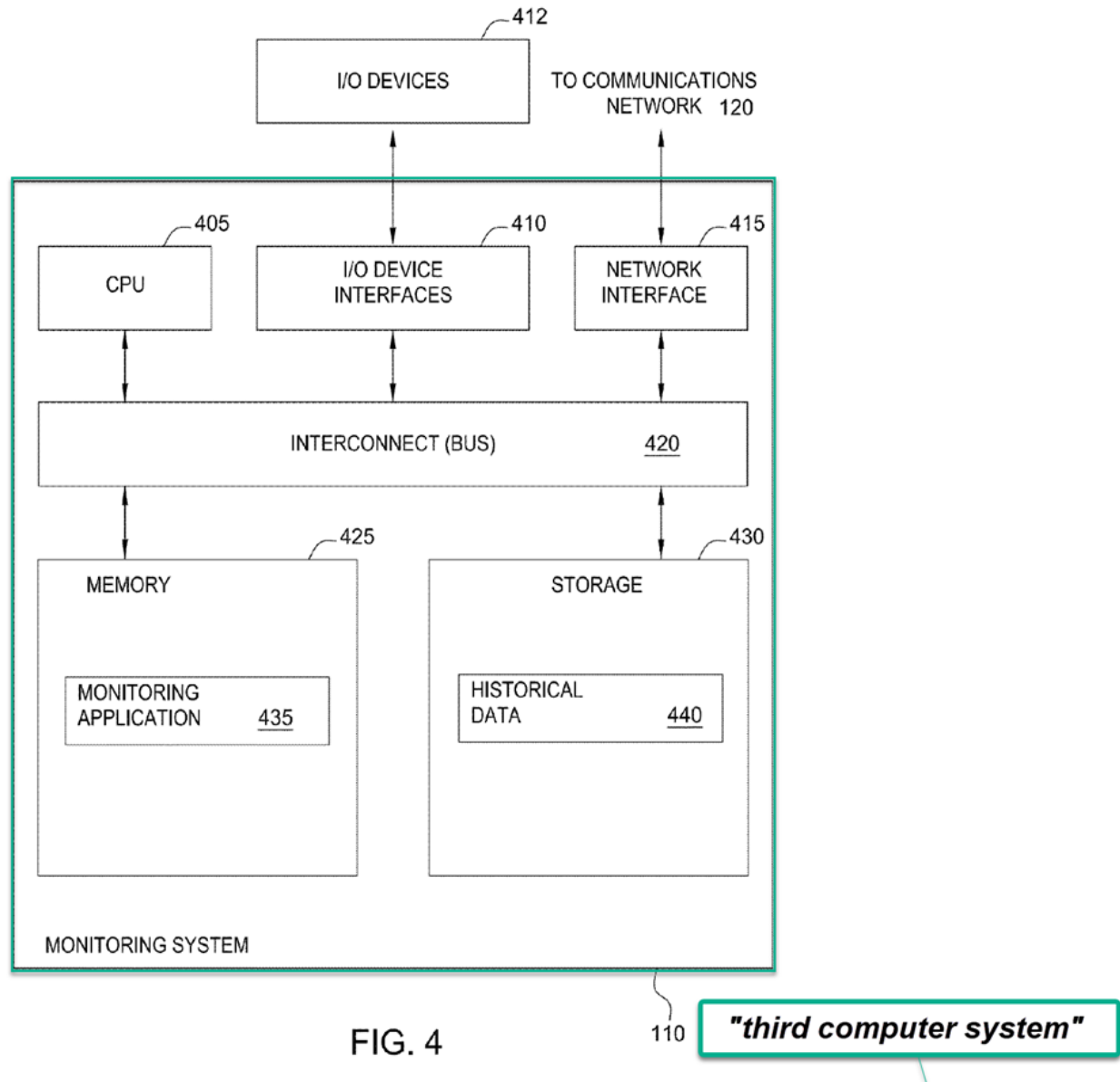
98. In my opinion, Barsness satisfies this element.

99. “*a third computer system of the second plurality of computer systems*”: In my opinion, Barsness discloses *a third computer system of the second plurality of computer systems* as recited in the limitation. Barsness discloses monitoring system 110 (*third computer system*) which Barsness describes as configured “to monitor data processing components of the stream application.” EX1004, 7:15-17. As I discussed earlier in connection with limitation 1[a], the *second plurality of computer systems* comprises Barsness’s management system 105 and monitoring system 110. Monitoring system 110 is one of those two members of the second plurality, and it is the member that performs the monitoring functions the third computer system carries out under the limitation.



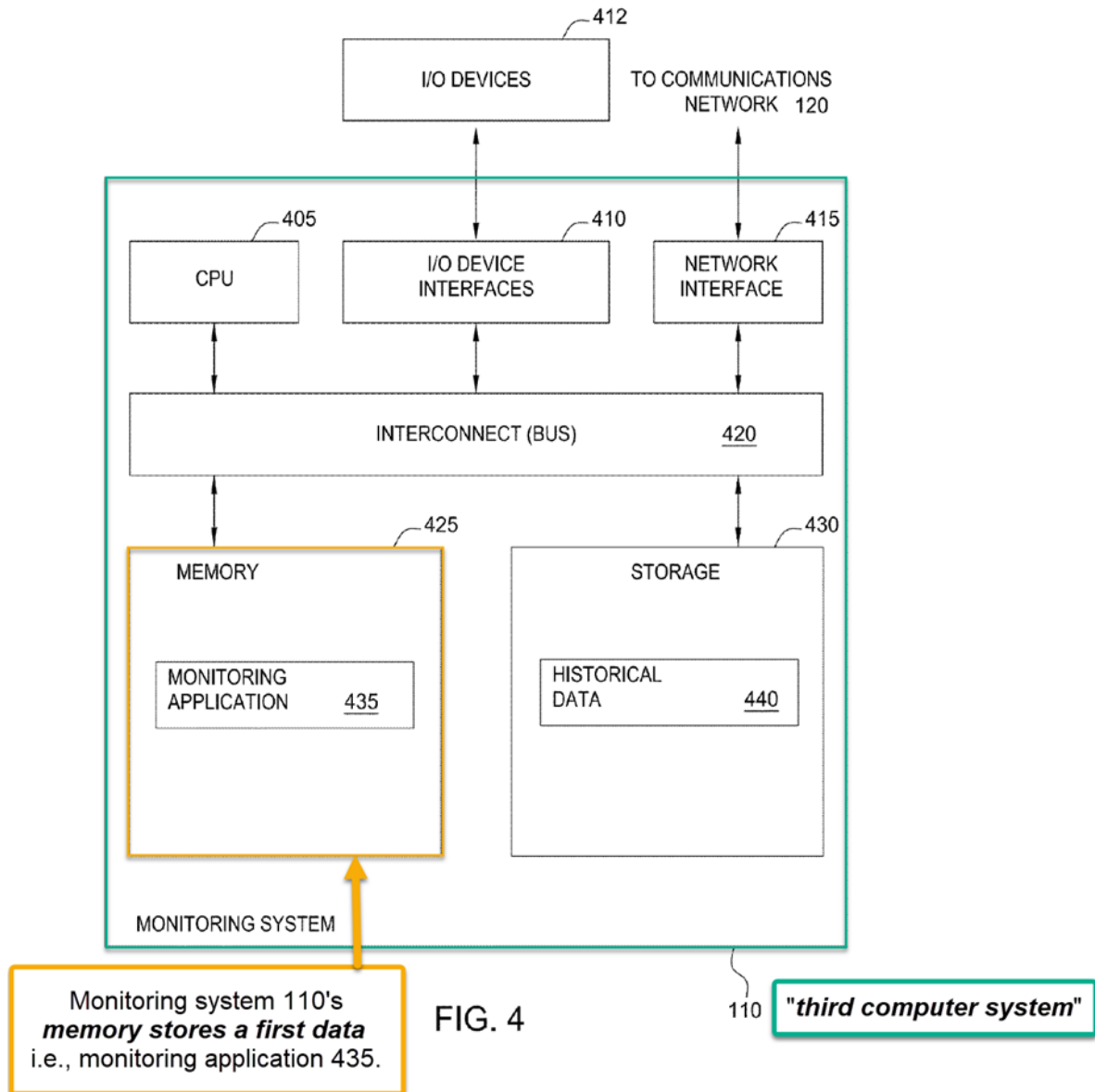
EX1004, Fig. 1A (annotations added).

100. Monitoring system 110 is itself a computer system as a POSITA would have understood that term. Barsness states that monitoring system 110 may be implemented either as “a single computing system” or as “a cluster of a plurality of computing resources, which may be distributed in the network 120.” EX1004, 7:17-21. In either implementation, the monitoring system has the standard hardware components of a computer system. Barsness describes monitoring system 110 as comprising a CPU 405, memory 425, storage 430, network interface 415, interconnect 420, and I/O device interface 410. EX1004, 11:7-11 (“Like CPU 305 of FIG. 3, CPU 405 is configured to retrieve and execute programming instructions stored in the memory 425 and storage 430.”); *id.*, 11:11-25 (further describing the interconnect 420, network interface 415, memory 425, and storage 430 of monitoring system 110); *id.*, FIG. 4 (depicting these components). In my opinion, a POSITA would have understood that monitoring system 110, whether realized as a single computing system or as a distributed cluster, is a computer system within the meaning of the *second plurality*.



101. “*a memory that stores a first data*”: In my opinion Barsness discloses this element. Monitoring system 110’s memory 425 stores programming instructions and application data, and is therefore *a memory* as the limitation requires. EX1004, 11:7-11 (“Like CPU 305 of FIG. 3, CPU 405 is configured to retrieve and execute programming instructions stored in the memory 425 and storage 430. Similarly, the

CPU 405 is configured to store and retrieve application data residing in the memory 425 and storage 430.”). Memory 425 stores a monitoring application 435, which Barsness identifies as the application that performs the system’s monitoring functions: “the memory 425 stores a monitoring application 435....The monitoring application 435 may be used to monitor performance of processing elements (PEs).” EX1004, 11:26-30; *id.*, FIG. 4. In my opinion, the monitoring application 435 stored in memory 425 is the *first data*.



EX1004, Fig. 4 (annotations added).

102. In my opinion, the monitoring application 435 (**first data**) includes operator graph 132. Barsness describes that the monitoring system uses a “graphical representation of the operator graph” to present the structure and behavior of the stream application. EX1004, 3:45-50 (“The graphical representation of the operator

graph may present not only how the job's PEs are connected to each other but may also allow the user to access logs, parts of a log, and performance metrics associated with the processing elements, operators, nodes, and data flows of a stream application."). More specifically, Barsness teaches that the monitoring application 435 includes a graphical user interface (GUI) 510 that is "configured to depict the operator graph 132 of a selected stream application," and that the GUI 510 "may display the interconnected computer nodes 130 and processing elements (PEs) as shown in FIG. 1B." EX1004, 11:53-61. The GUI 510 displays the operator graph 132 along with the PEs and operators and their interconnections. EX1004, 14:1-5 ("At step 606, PEs of the stream application are depicted and displayed in the GUI 510 of the monitoring application 435. As noted above, the operator graph 132 of the stream application includes PEs (and operators) and their interconnections."); *id.*, 14:5-7 ("Note that method steps described in steps 602 and 604 are performed independent of the displaying of the operator graph 132"); *see also id.*, 11:65-12:5; *id.*, 14:11-12 ("the user may be looking at the operator graph 132 of running code.").

103. The GUI 510's depiction of operator graph 132 includes, for each PE: (1) **Tuple Processing Data**, (2) **Performance Metrics**, and (3) **Configurations**. EX1004, 14:1-25; *id.*, 11:50-12:12. In my opinion, a POSITA would have understood that the monitoring application 435 stored in memory 425 contains an

operator graph that represents the stream application including the PEs, their interconnections, and their associated data.

104. **Tuple Processing Data:** In my opinion, the monitoring application 435 (*first data*) includes “tuple processing data” collected from each PE. Barsness teaches that compute nodes 130 collect tuple processing data from each PE: “At step 602, individual compute nodes 130 collect **tuple processing data** from each processing element 235, including operators 240.” EX1004, 13:32-34 (emphasis added). The compute nodes then transfer that collected data to monitoring system 110: “Compute nodes 130 then transfer this collected data at pre-configured intervals to the monitoring system 110, which persistently stores the collected data.” EX1004, 13:34-37; *see also id.*, 12:13-21 (describing how the data collector module 515 of monitoring application 435 “interact[s] with the data collector 255 of compute nodes 130 to retrieve the historical data related to each processing element 235 and operator 240 from each compute node 130”).

105. Monitoring system 110 stores the data it receives from the compute nodes in memory 425. Barsness states: “the monitoring system 110 may cache the collected data in memory 425.” EX1004, 13:43-45. Barsness similarly states that “[t]he transformed collected data is stored either in a database or in the memory 425.” EX1004, 13:66-67; *see also id.*, 9:30-33 (“The retrieved data is then stored at a database or memory cache coupled to the monitoring system 110.”). In my opinion,

a POSITA would have understood that the tuple processing data Barsness describes is stored in the memory 425 of monitoring system 110 and is part of the monitoring application 435 that constitutes the *first data*.

106. Barsness teaches that the GUI 510 of monitoring application 435 presents the tuple processing data as part of the operator graph 132. Barsness states that “[t]he graphical representation of the operator graph may present...[and] allow the user to access logs ... [that] typically includes pre-configured data related to the data processing of data tuples at the selected PE.” EX1004, 11:65-12:11. Barsness further confirms that the displayed data is tied to specific PEs in the graph: “the granularity of the collected data ... may be automatically increased for the selected operator or processing element being visualized.” EX1004, 14:22-25.

107. **Performance Metrics:** In my opinion, the monitoring application 435 (*first data*) calculates performance metrics from the tuple processing data it receives for each PE. Barsness teaches that monitoring system 110 transforms the collected tuple processing data into performance metrics “such as ‘average tuple rates within a pre-configured time intervals (e.g., average tuple rate in the last one hour at the input of PE1).’” EX1004, 13:64-66; *id.*, 11:65-12:3 (describing the graphical representation of the operator graph that presents performance metrics associated with PEs, operators, nodes, and data flows). The performance metrics are stored in memory 425. EX1004, 13:66-67 (“The transformed collected data is stored either in

a database or in the memory 425.’”). And the GUI 510 of monitoring application 435 presents those performance metrics as part of the operator graph 132. EX1004, 11:65-12:3 (“The graphical representation of the operator graph may present ... performance metrics associated with the processing elements”); *id.*, 14:10-25 (describing how the GUI 510 displays current and historical tuple processing data tied to the selected processing element in the operator graph). In my opinion, a POSITA would have understood that the calculated performance metrics, like the underlying tuple processing data, are stored in memory 425 as part of the monitoring application 435 and are part of the *first data*.

108. **Configuration:** In my opinion, the monitoring application 435 (*first data*) includes the configurations of the PEs in the operator graph 132. Barsness teaches that the GUI 510 of monitoring application 435 “may also be configured to display configurations of PEs (for example, which attributes of incoming data tuples are to be processed by a selected PE, etc.).” EX1004, 12:5-8. In my opinion, a POSITA would have understood that displaying the PE configurations in the GUI 510 requires the monitoring application 435 to have access to those configurations, which means the configurations are stored in memory 425. The GUI 510 cannot display data it does not have, and Barsness elsewhere makes clear that the data displayed in the GUI is drawn from the monitoring application 435’s stored data. EX1004, 11:26-30 (“the memory 425 stores a monitoring application 435 ... [t]he

monitoring application 435 may be used to monitor performance of processing elements (PEs) based on user interactions with the user interface of the monitoring application 435.”); *id.*, 13:43-45 (“the monitoring system 110 may cache the collected data in memory 425.”).

(j) **1[d][i] – wherein the first data represents a portion of a distributed computational graph,**

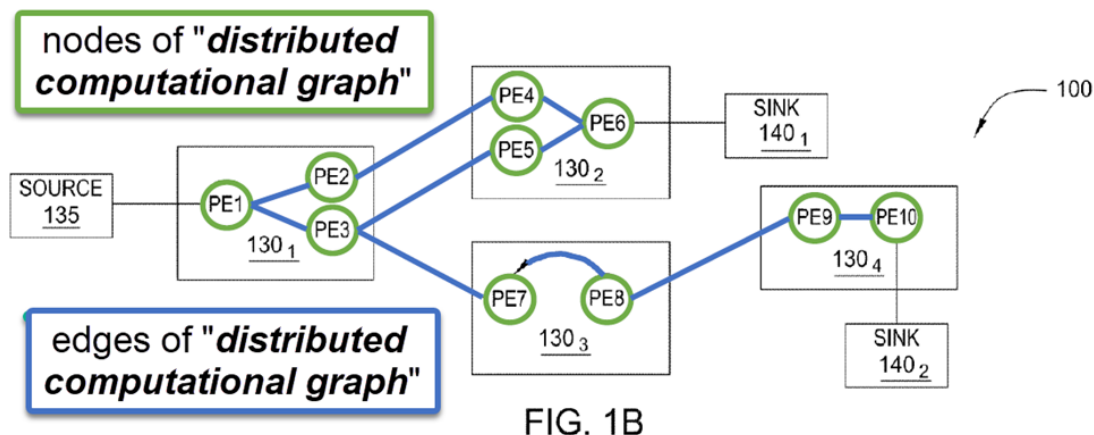
109. In my opinion, Barsness satisfies this element. Barsness discloses that monitoring application 435 (*first data*) includes data *represent[ing]* operator graph 132 (*distributed computational graph*).

110. “*first data*”: As discussed above regarding limitation 1[d], monitoring application 435 includes operator graph 132 combined with each PE’s: (1) **Tuple Processing Data**, (2) **Performance Metrics**, and (3) **Configurations**.

111. “*distributed computational graph*”: Operator graph 132 (*the distributed computational graph*) *represents* the plurality of processing elements and the links between processing elements, corresponding to a flow of data tuples through the stream application. EX1004, 1:43-46 (“The operator graph represents the plurality of processing elements, and links between processing elements, corresponding to a flow of data tuples through the stream application.”); *id.*, 2:12-15; *id.*, 1:59-62; *id.*, 6:43-56; *id.*, 10:49-51 (“the operator graph 132 maintains the

connectivity configuration of a plurality of compute nodes and processing elements in a streaming application”); *id.*, FIGS. 1A, 1B.

112. Figure 1B of Barsness depicts an example operator graph in which the processing elements (PEs) are graph nodes and the links between processing elements are graph edges:



EX1004, FIG. 1B (annotations added). The PEs are grouped into respective compute nodes 130_{1-4} . Each PE includes one or more operators, each of which is executable code that performs an operation on the data input to the PE. EX1004, 3:16-35 (describing the connection of operators between PEs and the distribution of operators across nodes); *id.*, 6:57-7:3 (“each processing element may be configured to carry out any form of data processing functions on a received N-tuple, including, e.g., writing to database tables or performing other database operations such as data joins, splits, reads, etc., as well as performing other data analytic functions or operations.”); *id.*, 9:3-8 (“The PEs 235 includes a collection of operators 240. Each

operator 240 may provide a small chunk of executable code configured to process data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132.”); *id.*, FIG. 2.

113. In my opinion, a POSITA would have understood that Barsness’s operator graph 132 is a ***distributed computational graph*** within the meaning of the limitation. The graph is ***computational*** because each node represents executable computation (a PE comprising operators that carry out data processing functions) and each edge represents a flow of data tuples that those computations consume and produce. *See* EX1014, 250 (“An operator fires when a data item is delivered to one of its input ports. During its firing, an operator can perform processing and produce data items on its output ports.”); *id.* (“Streams connect output ports of operators to input ports of other operators using FIFO semantics.”). The graph is ***distributed*** because the computations the graph represents are spread across multiple distinct compute nodes 130₁₋₄ that are connected by a communications network. EX1004, 6:43-56 (describing the computing infrastructure 100 with its plurality of compute nodes, each connected to communications network 120). The directed-graph-of-operators architecture Barsness describes was conventional in distributed stream processing systems known to a POSITA at the relevant time. *See* EX1011, 148 (“Vertices are further divided into two disjoint sets — spouts and bolts. Spouts are

tuple sources for the topology ... bolts process the incoming tuples and pass them to the next set of bolts downstream.”); EX1013, 271-273 (describing the operator-graph data processing model); EX1010, 2, 12 (describing distributed stream processing engines such as Aurora, Borealis, TimeStream, S4, Samza, and Storm as conventional approaches to “process continuous streams of data at low latency”).

114. As I discussed earlier in connection with claim construction, ***distributed computational graph*** means “[a] directed graph with each transformation comprising a node and the output messages between transformations comprising edges.” In my opinion, Barsness’s operator graph 132 meets this construction because each *node*¹ corresponds to a PE that performs a transformation on data in the stream, and each edge corresponds to the output flowing from one PE to another PE.

- “***directed graph***”: In my opinion, operator graph 132 is a ***directed graph***. The data stream flows through the PEs (nodes) from source 135 to sinks 140₁₋₂, and each edge indicates the direction of data flow from one PE node to the next PE node. EX1004, 1:43-46 (“The operator graph represents the plurality of processing elements, and links between

¹ Within the Barsness operator graph, each *graph node* is a PE; Barsness uses the term “compute node” to refer to the computer systems that execute or run the PEs.

processing elements, corresponding to a flow of data tuples through the stream application.”); *id.*, 2:12-15; *id.*, 1:59-62; *id.*, 6:43-56; *id.*, FIG. 1B; *see also* EX1001, 9:29-33 (describing directed graphs in the ’425 patent specification). Barsness illustrates the directionality of operator graph 132 with the data flow in Figure 1B: data flows from PE1 to PE2 and PE3, then from PE2 and PE3 to PE4 and PE5, and then from PE4 and PE5 to PE6. EX1004, 7:52-61. The directed-graph architecture Barsness describes was the conventional way of organizing distributed stream processing applications at the relevant time. *See* EX1011, 148 (“A topology is a directed graph where the vertices represent computation and the edges represent the data flow between the computation components.”); EX1010, 2 (“Stream engines code the application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology.”).

- “*transformation comprising a node*”: In my opinion, each PE (node) of operator graph 132 corresponds to a ***transformation***, which I have understood from the discussion of claim construction earlier in this declaration to mean a function performed on zero or more streams of input data that results in a single stream of output. Barsness teaches that each PE is configured to “carry out any form of data processing

functions” on the data it receives, including “data analytic functions or operations.” EX1004, 6:49-7:3; *see also id.*, 9:3-6. Each PE includes a “collection of operators 240,” and each operator is “a small chunk of executable code configured to process data flowing into a processing element.” EX1004, 9:3-9; *see also id.*, 3:16-22. The function each PE performs *transforms* the input: “the N-tuple received by a processing element need not be the same N-tuple sent downstream” (EX1004, 6:60-62), and PEs are “configured to transform incoming data tuples by either adding or removing one or more data attributes before emitting outgoing data tuples to other PEs” (EX1004, 8:11-15). The transformed result is emitted as a single stream of output to other PEs in the stream application. EX1004, 9:4-9 (“Each operator 240 may provide a small chunk of executable code configured to process data flowing into a processing element (e.g., PEs 235) and to emit data to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132.”); *id.*, 9:43-50; *id.*, 7:55-56 (“Source PE1 emits tuples received by PE2 and PE3.”); *compare* EX1001, 18:43-48 (describing a node that sends the same “output stream to two transformational pipelines”). Where Barsness describes that a PE “may split data elements” (EX1004, 7:56-57), the resulting splits collectively

comprise a single output stream produced by that PE. A streaming operator implementing a function that transforms input into output was conventional and well known to a POSITA at the relevant time. *See* EX1014, 250 (“An operator fires when a data item is delivered to one of its input ports. During its firing, an operator can perform processing and produce data items on its output ports.”).

- “*output messages between transformations comprising edges*”: In my opinion, the edges of operator graph 132 correspond to the output messages flowing between transformations. Barsness describes the edges as “links between processing elements, corresponding to a flow of data tuples through the stream application.” EX1004, 2:12-15; *see also id.*, 6:57-65 (describing how processing elements receive N-tuples from upstream and emit N-tuples downstream into the stream). The data tuples emitted from a PE on one edge of the operator graph are the *output messages* produced by that PE. *E.g.* EX1004, 12:45-50. The ’425 patent provides no specific definition of “output messages” but uses that term to refer to data flowing from a transformation node. *See, e.g.*, EX1001, 17:19-21. In my opinion, a POSITA would have understood that the data tuples Barsness describes as flowing along the edges of operator graph 132 are the *output messages* that satisfy this

prong of the construction. *See* EX1014, 250 (“Streams connect output ports of operators to input ports of other operators using FIFO semantics.”), EX1005, [0122], EX1015, 12:11-14.

115. In my opinion, Barsness’s operator graph 132 is a ***distributed computational graph*** under the Patent Owner’s asserted plain and ordinary meaning of the term. Operator graph 132 is a graph that describes the distributed computations performed on the data stream of the stream application: it identifies the PEs that carry out the computational functions, identifies the compute nodes 130₁₋₄ across which those PEs are distributed, and represents the flow of data between PEs along the edges of the graph. EX1004, 10:49-51 (“the operator graph 132 maintains the connectivity configuration of a plurality of compute nodes and processing elements in a streaming application”); *id.*, 2:12-15. Operator graph 132 thus has each defining attribute of a ***distributed computational graph*** as a POSITA would have understood that term: it is a graph with nodes and edges; it represents computations; it represents inputs and outputs of computations (e.g. the output of one computation is the input to another computation); and the computations are distributed across multiple compute nodes. *See* EX1010, 79-80 (describing the streaming-system architecture in which “tuples are communicated among operators running on compute nodes,” with topologies expressed as series of groupings between operators). In my opinion, a POSITA would have understood that Barsness’s operator graph 132 satisfies the

-80-

limitation under both Petitioner's proposed construction and the Patent Owner's asserted plain and ordinary meaning.

(k) 1[d][iii] – and wherein the portion of the distributed computational graph describes a flow of data from the first transformation pipeline to an input of the second transformation pipeline;

116. In my opinion, Barsness satisfies this element.

117. “*distributed computational graph describes a flow of data*”: In my opinion, Barsness discloses this limitation. As I discussed earlier in connection with limitation 1[d], the monitoring application 435 (*the first data*) includes the operator graph 132 combined with, for each PE, the **Tuple Processing Data**, the calculated **Performance Metrics**, and the PE's **Configurations**. This combined data describes the flow of data through each PE and among the compute nodes 130. EX1004, 13:32-37 (“At step 602, individual compute nodes 130 collects tuple processing data from each processing element 235, including operators 240. Compute nodes 130 then transfer this collected data at pre-configured intervals to the monitoring system 110, which persistently stores the collected data.”); *id.*, 14:1-12 (describing the GUI 510's display of the operator graph 132 and its PEs and interconnections, and the user's ability to view “the operator graph 132 of running code”).

118. The **Tuple Processing Data** and the calculated **Performance Metrics** that the monitoring application 435 stores are themselves expressed as flows of data

between PEs. For example, Barsness describes a representative performance metric as the “average tuple rate in the last one hour at the input of PE1.” EX1004, 13:64-66. That metric describes the rate at which data tuples flow into PE1 from upstream over a measured interval. Barsness similarly describes the collected data as “tuple rates at each inputs and outputs of each of the operators 240 of processing elements (PEs of FIG. 1B).” EX1004, 13:45-47. In my opinion, a POSITA would have understood that these tuple-rate measurements characterize the flow of data along the edges of operator graph 132, including the flow from one PE on one compute node to another PE on a different compute node. The portion of the operator graph 132 stored in monitoring application 435 therefore describes the *flow of data from the first transformation pipeline to an input of the second transformation pipeline*: PE1, PE2, and PE3 on compute node 130₁ form *the first transformation pipeline*; PE4, PE5, and PE6 on compute node 130₂ form *the second transformation pipeline*; and the Tuple Processing Data and Performance Metrics describe the inter-node flow from compute node 130₁’s PE2 and PE3 to compute node 130₂’s PE4 and PE5 along the edges of the operator graph. See EX1004, 7:57-67 (describing the cross-node tuple flow from PE2 to PE4 and from PE3 to PE5).

119. “*from the first transformation pipeline to an input of the second transformation pipeline*”: As I discussed earlier in connection with limitation 1[b][ii], the progression of transformations performed by PE1, PE2, and PE3 on

-82-

compute node 130₁ is the *first transformation pipeline*. As I discussed earlier in connection with limitation 1[c][i], the progression of transformations performed by PE4, PE5, and PE6 on compute node 130₂ is the *second transformation pipeline*.

120. In my opinion, monitoring application 435's operator graph 132 describes the *flow of data* between those two transformation pipelines through the tuple processing data and performance metrics it contains for each PE. The edges of operator graph 132 represent that flow: specifically, the edge from PE2 (in *the first transformation pipeline*) to PE4 (in *the second transformation pipeline*), and the edge from PE3 (in *the first transformation pipeline*) to PE5 (in *the second transformation pipeline*). Figure 1B of Barsness depicts these inter-pipeline edges:

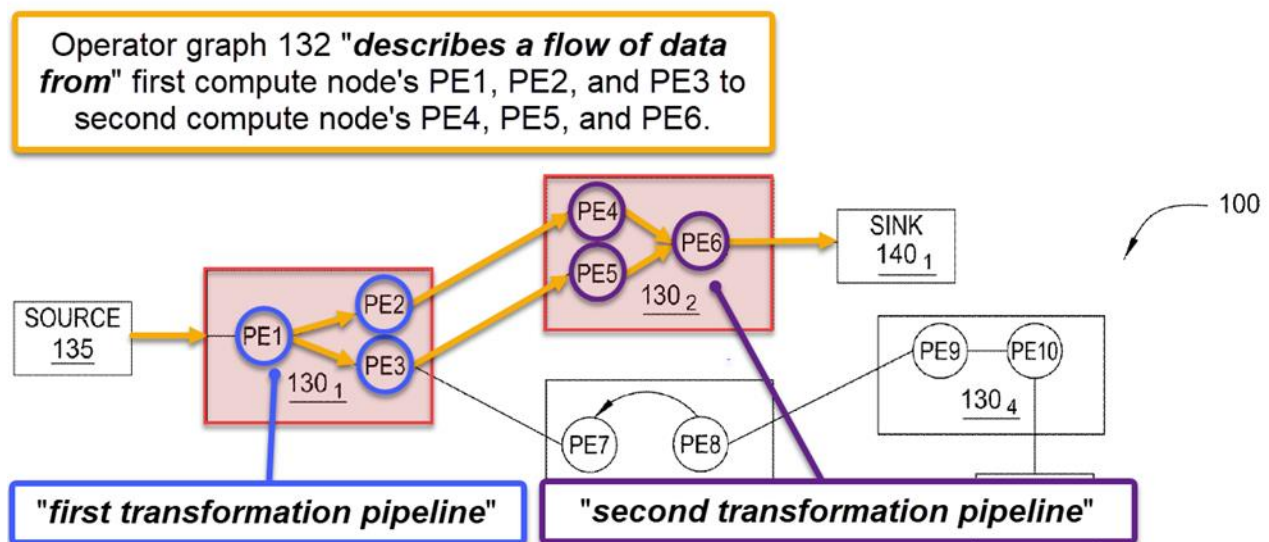


FIG. 1B

EX1004, FIG. 1B (annotations added); *id.*, 7:52-67 (describing the flow of data tuples through the nodes and PEs of the operator graph). In my opinion, a POSITA -83-

would have understood that the flow of data along these edges, namely the tuple flow from PE2 and PE3 on compute node 130₁ to PE4 and PE5 on compute node 130₂, is the *flow of data from the first transformation pipeline to an input of the second transformation pipeline*.

(l) 1[e] – wherein the third computer system is configured to execute third software instructions that:

121. In my opinion, Barsness discloses this limitation. Monitoring system 110 (*the third computer system*) is configured, by way of its CPU and memory, to execute the software instructions corresponding to the monitoring application 435 stored in its memory. Barsness states: “Like CPU 305 of FIG. 3, CPU 405 is configured to retrieve and execute programming instructions stored in the memory 425 and storage 430.” EX1004, 11:7-9. Barsness further states: “the memory 425 stores a monitoring application 435.” EX1004, 11:26-27. The monitoring application 435 stored in memory 425 is the executable code that monitoring system 110’s CPU 405 retrieves and executes, and that code is the *third software instructions*.

(m) 1[e][i] – process the first data to determine information about at least one of the first transformation pipeline and the second transformation pipeline,

122. In my opinion, Barsness discloses *processing the first data to determine information about* PE1, PE2, and PE3 on compute node 130₁ (*the first*

transformation pipeline) and PE4 and PE5 on compute node 130₂ (the *second transformation pipeline*). Barsness discloses three alternative ways in which monitoring application 435 carries out that *processing to determine information: Identifying Compute Nodes, Configuring Processing Elements, and Calculating Performance Metrics*. I address each in turn below.

123. **Identifying Compute Nodes:** In my opinion, monitoring system 110 *processes the first data* by examining monitoring application 435's operator graph 132 *to determine* the compute node on which each PE resides. That determination is itself *information about the first transformation pipeline and the second transformation pipeline*. Barsness teaches that monitoring system 110 collects data from compute nodes 130 and arranges that data according to the associated PE: the historical data 440 in monitoring system 110's storage "may include tuple rates at each processing element in the stream application" and "may also include more detailed data such as tuple rate at each input and output port of each operators 240 in a processing element (PE)." EX1004, 11:40-49; *see also id.*, 13:45-47 ("The collected data may include tuple rates at each inputs and outputs of each of the operators 240 of processing elements (PEs of FIG. 1B)."); *id.*, 13:56-67 (describing how monitoring system 110 "formats and transforms the collected data according to pre-set configurations," including "average tuple rates within a pre-configured time intervals (e.g., average tuple rate in the last one hour at the input of PE1)"). To collect

data from each compute node, “monitoring system 110 is configured to communicate with compute nodes 130₁₋₄.” EX1004, 7:21-22; *see also id.*, 7:17-21; *id.*, 11:18-20 (“The network interface 415 is configured to transmit data via the communications network 120.”); *id.*, 6:45-48; *id.*, FIGS. 1B, 2, and 4. Barsness specifically describes how monitoring system 110 retrieves the data from each compute node: “The monitoring system 110 may be configured to interact with the data collector 250 either continuously or at pre-configured intervals to retrieve data from the data collector 250.” EX1004, 9:29-32. In my opinion, a POSITA would have understood that for monitoring system 110 to communicate with the appropriate compute node, retrieve the data from that compute node’s data collector 250, and arrange the retrieved data according to the associated PE, monitoring system 110 must *process* operator graph 132 to determine which PEs reside on which compute nodes. The result of that processing is *information* identifying the compute node for each PE, including PE1, PE2, and PE3 on compute node 130₁ and PE4 and PE5 on compute node 130₂.

124. **Configuring Processing Elements:** In my opinion, monitoring system 110 *processes the first data* by examining the PE configurations stored in monitoring application 435 and using those configurations to set up the data collection that occurs at each compute node. Barsness states: “a user configures the monitoring system 110 with regards to the type of data to be collected. The monitoring system

-86-

110 configures compute nodes 130 to collect the data accordingly.” EX1004, 13:49-52. As I discussed earlier in connection with limitation 1[d], monitoring application 435 includes the configurations for each compute node 130 and PE. Similarly, each compute node 130 includes a control module 255 that allows the monitoring system 110 to configure the compute node 130. EX1004, 9:34-42. For example, Barsness explains that, based on a user’s focus on a particular PE, “the monitoring application could turn on additional tracing in the log file or set of log files that are connected to a given PE,” and likewise “could remove tracing (or step down a logging level)” when appropriate. EX1004, 4:26-29; *see also id.*, 12:3-12 (describing the GUI 510’s display of PE configurations and the user’s ability to view configurations of selected PEs). Barsness further confirms that the configuration extends to per-PE granularity: “the granularity of the collected data is configurable by a user or may be set to variable based on occurrences of pre-configured events.” EX1004, 13:53-55. In my opinion, a POSITA would have understood that for monitoring system 110 to use the per-PE configurations stored in monitoring application 435 to configure each compute node’s data collection, monitoring system 110 must *process* those configurations. The result is *information* about the per-PE collection setup, including for compute node 130₁’s PE1, PE2, and PE3 (*first transformation pipeline*) and compute node 130₂’s PE4 and PE5 (*second transformation pipeline*).

125. **Calculating Performance Metrics:** In my opinion, monitoring system 110 *processes the first data* by taking the tuple processing data collected from each PE and calculating performance metrics from that data. Barsness discloses that monitoring system 110 collects tuple processing data for each PE: “At step 602, individual compute nodes 130 collect tuple processing data from each processing element 235, including operators 240. Compute nodes 130 then transfer this collected data at pre-configured intervals to the monitoring system 110.” EX1004, 13:32-36. Monitoring system 110 then “formats and transforms the collected data according to pre-set configurations” to calculate, for example, “average tuple rates within a pre-configured time intervals (e.g., average tuple rate in the last one hour at the input of PE1).” EX1004, 13:56-67. Monitoring system 110 uses those calculated performance metrics to compare current tuple processing rates against historical tuple processing rates: “the data analysis module 530 may compare these rates with the historical processing rates (or historical processing averages) to determine if there is any issue with the selected processing element.” EX1004, 12:40-54. In my opinion, a POSITA would have understood that the calculated performance metrics are the result of monitoring system 110 *processing the first data*, and they are *information* about the PEs in the *first transformation pipeline* and the *second transformation pipeline*, including compute node 130₁’s PE1, PE2, and PE3 and compute node 130₂’s PE4 and PE5.

- (n) **1[e][iii] – monitor at least a portion of the execution of at least one of the first software instructions and the second software instructions in accordance with the determined information, and**

126. In my opinion, Barsness discloses this limitation. Monitoring system 110 monitors the execution of each PE on compute nodes 130 by continuously collecting tuple processing data from those compute nodes and comparing the current rates against historical performance metrics. Barsness states that monitoring system 110 is included in the computing infrastructure “to monitor data processing components of the stream application.” EX1004, 7:15-17; *see also id.*, 9:30-33 (“The retrieved data is then stored at a database or memory cache coupled to the monitoring system 110.”); *id.*, 11:27-30 (“The monitoring application 435 may be used to monitor performance of processing elements (PEs) based on user interactions with the user interface of the monitoring application 435.”). Barsness explains that monitoring system 110 continuously collects tuple processing data from each compute node as part of its method for monitoring the stream application: “At step 602, individual compute nodes 130 collect tuple processing data from each processing element 235, including operators 240.” EX1004, 13:32-34; *see also id.*, 14:7-9 (“the monitoring system 110 may continue to collect data from compute nodes 130 while step 606 is being performed.”); *id.*, 9:23-27 (describing the data collector 250’s collection of tuple rates at pre-configured intervals); *id.*, 12:13-21

(describing the data collector module 515 of monitoring application 435 retrieving historical data per PE). The collected data includes “tuple rates at each inputs and outputs of each of the operators 240 of processing elements (PEs of FIG. 1B).” EX1004, 13:45-47. Monitoring system 110’s GUI 510 “may be configured to continuously monitor the current tuple processing rates at various processing elements and compare the current tuple processing rates with the historical tuple processing rates.” EX1004, 13:4-7. In my opinion, monitoring system 110 *monitors* the rate of execution of each PE based on the inputs and outputs of that PE.

127. *The execution of at least one of the first software instructions and the second software instructions.* In my opinion, monitoring system 110 monitors each PE’s execution, namely the *first software instructions* and the *second software instructions*, through the tuple processing rates that measure how each PE processes data. Barsness illustrates the monitoring concretely: a processing element could be “receiving 100 tuples per second at the input but only producing 30 tuples at the output,” and if “the selected processing element in the last one hour was processing on an average 70 tuples, the current processing of 30 tuples per second may be an indication of performance issues with the selected processing element.” EX1004, 12:45-54. As I discussed earlier in connection with limitation 1[b], compute node 130i’s CPU executes the code for PE1, PE2, and PE3, which is the *first software instructions*. As I discussed earlier in connection with limitation 1[c], compute node

-90-

1302's CPU executes the code for PE4 and PE5, which is the *second software instructions*. The tuple processing rates measure the rate at which each of those PEs *executes* its software instructions, expressed in terms of data tuples processed per unit time. In my opinion, by tracking those rates, monitoring system 110 monitors *the execution of at least one of the first software instructions and the second software instructions*. The kind of execution-level monitoring Barsness describes was conventional in distributed stream processing systems known to a POSITA at the relevant time. *See* EX1011, 149 (describing how Storm's Nimbus is responsible for "monitoring the progress of the tuples flowing through the topology"); EX1010, 110 ("[A] single bottleneck in the pipeline (congested link or an overloaded operator) can drastically impact the system throughput."); *id.*, 75 ("To detect and diagnose temporally varying pipeline-bottlenecks, SEs need a feedback-driven control loop.").

128. **Identifying Compute Node** - "*in accordance with the determined information*": In my opinion, monitoring system 110's monitoring of tuple processing rates is performed *in accordance with* the PEs identified from monitoring application 435's operator graph 132 (the *determined information* under the **Identifying Compute Nodes** alternative I addressed earlier). Monitoring system 110 collects the tuple processing rates for each PE from compute nodes 130. EX1004, 13:45-47 ("The collected data may include tuple rates at each inputs and

-91-

outputs of each of the operators 240 of processing elements (PEs of FIG. 1B).”). In my opinion, the tuple processing rates monitoring system 110 collects from each compute node 130 accord with the PE-to-compute-node identification monitoring system 110 derives from operator graph 132.

129. **Configuring Processing Elements** - “*in accordance with the determined information*”: In my opinion, monitoring system 110’s monitoring of tuple processing rates is performed *in accordance with* the configuration of each compute node 130 that monitoring system 110 determines from the PE configurations stored in monitoring application 435 (the *determined information* under the **Configuring Processing Elements** alternative I addressed earlier). EX1004, 13:49-55 (“[A] user configures the monitoring system 110 with regards to the type of data to be collected. The monitoring system 110 configures compute nodes 130 to collect the data accordingly.”); *id.*, 14:22-25 (“the granularity of the collected data (step 602) may be automatically increased for the selected operator or processing element being visualized or interacted with in the GUI 510.”).

130. **Calculating Performance Metrics** - “*in accordance with the determined information*”: In my opinion, monitoring system 110’s monitoring of tuple processing rates is performed *in accordance with* the performance metrics calculated for each PE (the *determined information* under the **Calculating Performance Metrics** alternative I addressed earlier). Monitoring system 110 stores

historical data 440 retrieved from each compute node 130's PEs. EX1004, 11:38-49 (describing how historical data 440 includes "tuple rates at each processing element in the stream application" and "tuple rate at each input and output port of each operators 240 in a processing element (PE)"); *id.*, 12:13-21 (describing how the data collector module 515 of monitoring application 435 "interact[s] with the data collector 255 of compute nodes 130 to retrieve the historical data related to each processing element 235 and operator 240 from each compute node 130"). Monitoring system 110 then analyzes the historical data along with the current tuple processing rates to identify any issues at a PE: "the data analysis module 530 may compare these rates with the historical processing rates (or historical processing averages) to determine if there is any issue with the selected processing element." EX1004, 12:47-50; *see also id.*, 13:4-12 (describing the GUI 510 continuously monitoring current tuple processing rates and comparing them against historical tuple processing rates); *id.*, 14:26-35 (describing decision step 610 in which monitoring system 110 determines if the current tuple processing rate is outside the pre-configured range based on historical averages).

- (o) **1[f] – in response to the monitoring, cause a fourth computer system of the third plurality of computer systems to execute fourth software instructions that perform at least one of:**

131. In my opinion, Barsness discloses this limitation. Monitoring system 110 (*the third computer system*) is configured to provision and configure additional compute nodes 130_x (*a fourth computer system of the third plurality of computer systems*) to the stream application when it identifies discrepancies between a monitored PE's current tuple processing rates and its historical tuple processing rates (*in response to the monitoring*).

132. "*in response to the monitoring*": Barsness teaches that while monitoring system 110 monitors the tuple processing rates of each PE, it "determines if the current tuple processing rate is outside a pre-configured range based on the historical tuple processing rate averages." EX1004, 14:26-29. If the current rate falls outside the pre-configured range, monitoring system 110 takes corrective action: "if the current tuple processing rate is above or below the pre-configured tuple processing rate, the monitoring system 110 may take pre-configured corrective actions." EX1004, 14:31-34; *see also id.*, 12:37-13:3 (describing the data analysis module 530's comparison of current and historical rates and the corrective steps that follow); *id.*, 13:4-14 (describing the GUI 510's continuous monitoring and the monitoring application 435's automatic corrective measures); *id.*, 13:22-29 (describing user-configured discrepancy handling). The corrective actions monitoring system 110 takes include "spawning new compute nodes, [and] duplicating a compute node." EX1004, 14:54-55; *see also id.*, 10:28-39 (describing

-94-

how monitoring system 110 instructs management system 105 to spawn more compute nodes when the monitoring system identifies a performance issue with a compute node or its PEs); *id.*, 11:30-34 (describing monitoring application 435's configuration capabilities including spawning new PEs and operators); *id.*, 12:24-28 (describing application control module 525 sending messages to compute nodes and management system 105 for reconfiguration); *id.*, 12:54-67 (describing the monitoring application 435's interaction with management system 105 to remedy performance issues including by adding more processing elements); *id.*, cls. 2, 5. To carry out these corrective actions, monitoring system 110 interacts with management system 105. EX1004, 10:29-34 ("the monitoring system 110 may interact with the node manager 340 to instruct the management system 105 to spawn more process nodes 130 to process certain type of data when the monitoring system 110 determines a need based on a detection of performance issues with a computer node 130 or the processing elements 235 in a computer node 130.").

133. In my opinion, the chain of activity Barsness describes—monitoring system 110 monitors the per-PE tuple processing rates; identifies discrepancies between current and historical rates; and triggers corrective actions including spawning or duplicating compute nodes—satisfies the *in response to the monitoring* language of the limitation. The kind of feedback-driven response Barsness teaches was conventional in distributed stream processing systems known to a POSITA at

-95-

the relevant time. *See* EX1010, 75 (“To detect and diagnose temporally varying pipeline-bottlenecks, SEs need a feedback-driven control loop.”); EX1010, 79 (“The per-topology controller periodically collects metrics from the system. Based on the observed bottlenecks, the controller computes new route-maps that minimize the maximum network and CPU utilization”); EX1008, 242 (describing the “Apache S4 ... lightweight approach” in which “Operators ... hosted on failed servers Processing Nodes are automatically moved to a standby server”); EX1012, 173 (describing S4’s communication layer that “provides cluster management and automatic failover to standby nodes”).

134. “*cause a fourth computer system of the third plurality of computer systems to execute fourth software instructions*”: In my opinion, monitoring system 110 *causes* new compute nodes 130_x (*third plurality of computer systems*) to “spawn” in order to provide additional computing resources to the stream application, and once spawned, those new compute nodes *execute fourth software instructions*. EX1004, 10:29-34 (“the monitoring system 110 may interact with the node manager 340 to instruct the management system 105 to spawn more process nodes 130 to process certain type of data when the monitoring system 110 determines a need based on a detection of performance issues with a computer node 130 or the processing elements 235 in a computer node 130.”). Each of the spawned compute nodes 130_x is a *fourth computer system*. Barsness further teaches that the

new compute nodes process the same type of data as the existing compute nodes “in parallel,” which occurs when a new compute node is assigned the same processing elements as an existing compute node. EX1004, 10:26-28 (“More than one processing elements 235 and/or operators 240 may be used in parallel to process same type of data to increase data throughput.”); *see also id.*, 14:30-57 (describing the corrective actions monitoring system 110 takes when current tuple processing rates fall outside the pre-configured range, including spawning new compute nodes and duplicating a compute node, to maintain the stream application’s performance).

135. In my opinion, a POSITA would have understood that “spawning” a new compute node 130 means provisioning and bringing online an additional ***computer system***, beyond those already participating in the stream application, that is then configured to execute one or more PEs and to begin processing data. Barsness teaches that the compute nodes hosting the stream application can be instantiated dynamically. Specifically, Barsness discloses that compute nodes 130 may be virtual machines, and the computing infrastructure can “spawn more virtual machines on the fly.” EX1004, 8:30-52. Barsness pairs that capability with a cloud computing deployment model that supplies the elastic resource pool from which spawned nodes are drawn. Barsness explains: “Embodiments of the invention may be provided to end users through a cloud computing infrastructure. Cloud computing generally refers to the provision of scalable computing resources as a service over a

network....[C]loud computing allows a user to access virtual computing resources (e.g., storage, data, applications, and even complete virtualized computing systems) in ‘the cloud,’ without regard for the underlying physical systems (or locations of those systems) used to provide the computing resources.” EX1004, 6:16-32. Barsness further teaches that this cloud model is the resource pool from which the stream application’s compute nodes are drawn: “the nodes used to create a stream application may be virtual machines hosted by a cloud service provider.” EX1004, 6:40-42.

136. In my opinion, a POSITA would have understood that the cloud computing infrastructure Barsness describes provides exactly the kind of elastic, on-demand resource pool needed to spawn new compute nodes in response to monitoring system 110’s detection of performance issues. Barsness identifies key attributes of cloud computing that enable this dynamic provisioning: it provides “convenient, on-demand network access to a shared pool of configurable computing resources that can be rapidly provisioned and released with minimal management effort,” and the resources are charged on a “pay-per-use basis, where users are charged only for the computing resources actually used (e.g., an amount of storage space consumed by a user or a number of virtualized systems instantiated by the user).” EX1004, 6:23-37. The “rapid[] provision[ing]” and pay-per-use model enable monitoring system 110 to scale the stream application’s compute resources

-98-

up by spawning a new compute node from the cloud pool when discrepancies in tuple processing rates are detected, and to scale back down when the additional resources are no longer needed. A POSITA would have known that cloud-based, elastically scalable stream processing was a well-developed and well-understood architecture for distributed stream processing systems at the relevant time. *See* EX1008, 238 (“Third generation system design is strongly driven by the trend towards cloud computing, which requires the data stream processing engines to be highly scalable and robust towards faults.”); *id.*, 240 (“Elasticity describes the behavior of a system to dynamically scale in or out based on a changing workload.”); *id.*, 240 (describing cloud-based stream processing systems as “designed to dynamically scale to hundreds of computing nodes and automatically cope with varying workloads”); EX1012, 170 (describing S4 as a platform designed to “run in large clusters built with commodity hardware” to meet “high availability” requirements). In my opinion, a POSITA would have understood that Barsness’s combination of dynamic virtual-machine instantiation and cloud-based scalable infrastructure enables monitoring system 110 to provision new compute nodes 130_x on demand to add to the stream application, as needed to maintain expected performance.

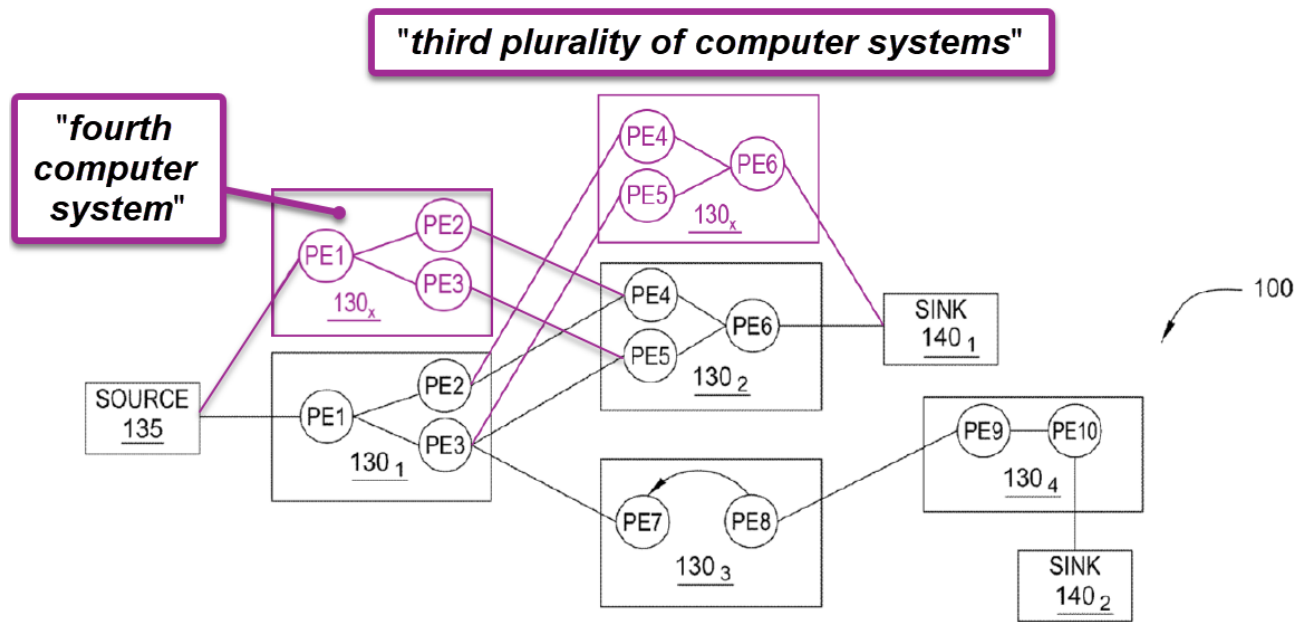


FIG. 1B

EX1004, FIG. 1B (annotations added).

137. In my opinion, the new or duplicated compute nodes 130_x that monitoring system 110 spawns comprise the *third plurality of computer systems*. As I discussed in the preceding paragraphs, Barsness teaches that compute nodes 130 may be implemented as virtual machines, and that the computing infrastructure can “spawn more virtual machines on the fly.” EX1004, 8:30-52. Barsness further teaches that the stream application’s compute nodes may be drawn from a cloud computing infrastructure that supplies “the provision of scalable computing resources,” including that “the nodes used to create a stream application may be virtual machines hosted by a cloud service provider.” EX1004, 6:16-42. In my opinion, a POSITA would have understood that the new compute nodes 130_x

monitoring system 110 spawns from this elastic resource pool constitute a *third plurality of computer systems*, distinct from the first plurality of compute nodes 130₁₋₄ that execute the existing transformation pipelines and from the second plurality comprising management system 105 and monitoring system 110.

- (p) **1[f][i] – processing second portions of the first stream of data by applying the first transformation pipeline to the second portions of the first stream of data, or**
- 1[f][ii] – processing second portions of the first pipeline output messages by applying a second transformation pipeline to the second portions of the first pipeline output messages,**

138. In my opinion, Barsness discloses this limitation. Monitoring system 110 *causes* compute node 130_x (*the fourth computer system*) to be provisioned with the processing elements PE1, PE2, and PE3 of the *first transformation pipeline* to address performance issues that arise with compute node 130₁. EX1004, 10:30-34 (“the monitoring system 110 may interact with the node manager 340 to instruct the management system 105 to spawn more process nodes 130 to process certain type of data when the monitoring system 110 determines a need based on a detection of performance issues with a computer node 130 or the processing elements 235 in a computer node 130.”); *see also id.*, 10:34-39 (describing how monitoring system 110 “may also interact with the node manager 340 to instruct the management system 105 to move processing elements 235 from one computer node 130 to another to improve the data processing performance of the stream application”).

-101-

Compute node 130_x processes the same type of stream data in parallel with compute node 130₁, increasing data throughput and maintaining the desired processing rate for those PEs. EX1004, 10:26-28 (“More than one processing elements 235 and/or operators 240 may be used in parallel to process same type of data to increase data throughput.”); *id.*, 14:54-55 (corrective actions include “spawning new compute nodes, [and] duplicating a compute node”); *id.*, 14:30-39 (describing the corrective-action sequence triggered when current rates fall outside the pre-configured range). Figure 1B of Barsness, as annotated, depicts compute node 130_x (*the fourth computer system*) configured with the processing elements of compute node 130₁ (*the first transformation pipeline*):

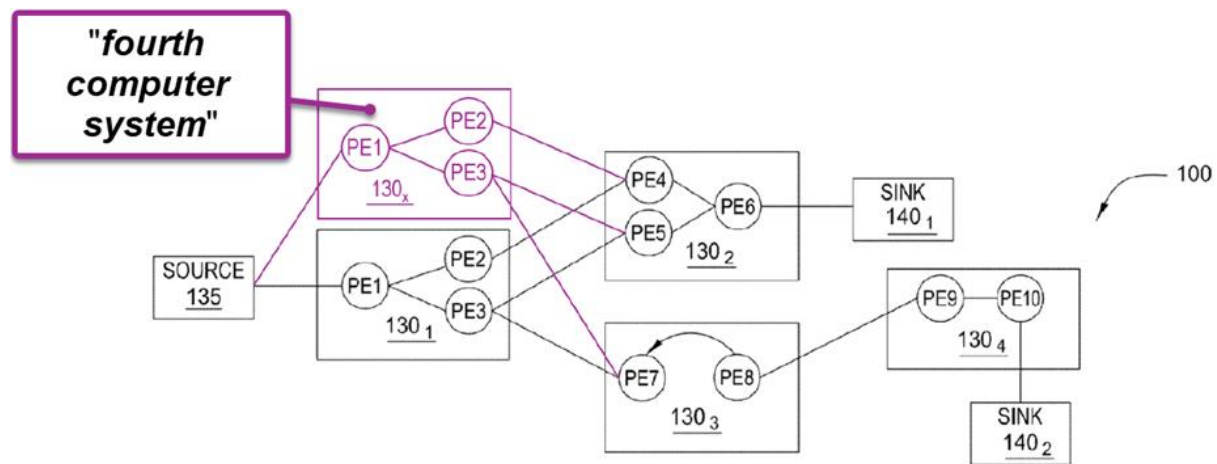


FIG. 1B

EX1004, FIG. 1B (annotations added).

139. The same Barsness teachings apply to the second transformation pipeline: if performance issues arise on compute node 130₂, monitoring system 110

causes compute node 130_x to be provisioned and configured with the processing elements of compute node 130₂ (the *second transformation pipeline*). EX1004, 10:34-39; *id.*, 10:28-34; *id.*, 14:54-55. Figure 1B of Barsness, as annotated, depicts this alternative configuration:

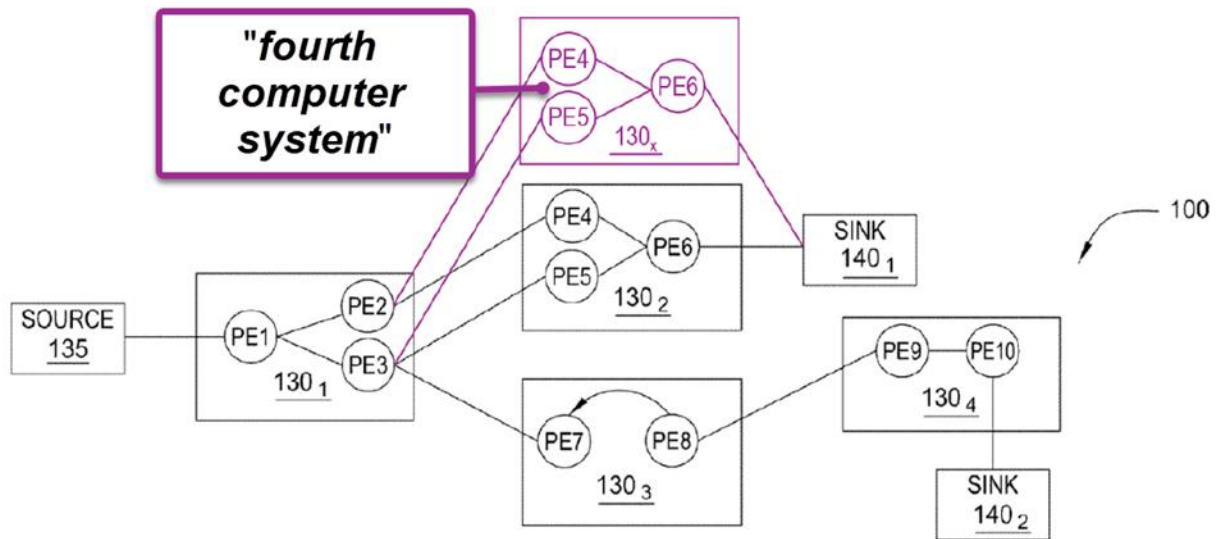


FIG. 1B

EX1004, FIG. 1B (annotations added).

140. Although Barsness discloses “spawning new compute nodes” and “duplicating a compute node” generally, EX1004, 14:54-55, in my opinion a POSITA would have found it obvious to duplicate compute node 130₁ or compute node 130₂ specifically. Barsness teaches that monitoring system 110 interacts with management system 105 to duplicate compute nodes as a corrective measure for improving data processing rates, and the discrepancies monitoring system 110 detects are tied to specific PEs and the compute nodes on which those PEs reside.

EX1004, 10:26-39 (describing the parallel-processing and node-spawning corrective measures); *id.*, 14:26-56 (describing the decision step 610 sequence in which monitoring system 110 identifies the discrepancy and takes corrective action with respect to the affected processing element). In my opinion, when monitoring system 110 detects a discrepancy (e.g. higher or lower than expected) in the tuple processing rate of, for example, PE2 on compute node 130₁, the application of Barsness's corrective-action teaching is to duplicate compute node 130₁ (and its constituent PEs) so that the increased tuple processing capacity is applied to the same type of data PE2 processes. Otherwise, spawning a compute node configured with unrelated PEs would not address the performance issues detected by monitoring system 110.

141. Barsness teaches that the new, duplicated compute nodes perform the same processing as the compute nodes they duplicate. As I explained in connection with limitations 1[b][ii] and 1[c][i], compute node 130₁ processes the first stream of data by applying *the first transformation pipeline*, and compute node 130₂ processes *the first pipeline output messages by applying the second transformation pipeline*.

Barsness teaches that a new, duplicated compute node processes a different, *second portion* of the respective data because it processes the “same type of data to increase data throughput.” EX1004, 10:26-28. Data throughput is determined by the tuple processing rate, namely how many tuples from the data stream are processed per second. EX1004, 12:40-54. To increase the tuple processing rate, the duplicated

compute node 130_x must process different tuples within the incoming data stream than the compute node it duplicated. In other words, when monitoring system 110 spawns compute node 130_x to duplicate compute node 130₁, compute node 130_x processes the *second portions of the first stream of data by applying the first transformation pipeline* to those second portions; and when monitoring system 110 spawns compute node 130_x to duplicate compute node 130₂, compute node 130_x processes the *second portions of the first pipeline output messages by applying the second transformation pipeline* to those second portions. See EX1014, 250 (describing data parallelism as “different data items are processed by the same operator in parallel,” “typically achieved by replicating the operator in question and routing data items to different replicas”).

(q) 1[g] – wherein the first computer system and the third computer system are distinct, and

142. In my opinion, Barsness discloses this limitation. Compute node 130₁ (*the first computer system*) and monitoring system 110 (*the third computer system*) are *distinct* computer systems. As I discussed earlier in connection with limitations 1[a] and 1[b], Barsness discloses these computer systems as part of distinct pluralities: compute node 130₁ is part of *the first plurality of computer systems*, and monitoring system 110 is part of the *second plurality of computer systems*. The two are also structurally distinct in their hardware. Barsness describes each compute

node 130 as a physical host or virtual machine comprising a CPU 205, network interface 215, interconnect 220, memory 225, and storage 230. EX1004, 8:30-65; *id.*, FIG. 2. Barsness separately describes monitoring system 110 as comprising a CPU 405, network interface 415, interconnect 420, memory 425, and storage 430. EX1004, 10:65-11:25; *id.*, FIG. 4. The two computer systems also operate at arm's length and communicate over the network: "monitoring system 110 is configured to communicate with compute nodes 130₁₋₄." EX1004, 7:21-22. In my opinion, a POSITA would have understood that compute node 130₁ and monitoring system 110 are physically and functionally *distinct* computer systems.

(r) 1[h] – wherein the third computer system and the fourth computer system are distinct.

143. In my opinion, Barsness discloses this limitation. As I explained in connection with limitation 1[f], Barsness discloses *the fourth computer system* through monitoring system 110 (*the third computer system*) "spawning new compute nodes, [and] duplicating a compute node," where the new or duplicated compute node 130_x is the *fourth computer system*. EX1004, 14:40-56. Barsness draws the same distinction between compute nodes 130 and monitoring system 110 that I explained in connection with limitation 1[g]: monitoring system 110 is part of the *second plurality of computer systems*, and the spawned or duplicated compute nodes 130_x are part of the *third plurality*. The two have separate hardware, occupy

distinct roles in the computing infrastructure, and communicate at arm's length over the network. In my opinion, monitoring system 110 (*the third computer system*) and the spawned compute node 130_x (*the fourth computer system*) are *distinct* computer systems within the meaning of the limitation.

2. **Claim 2 – The system of claim 1, wherein the third software instructions monitor at least a portion of the execution of at least one of the first software instructions and the second software instructions to identify execution that has become unresponsive.**

144. In my opinion, Barsness discloses Claim 2. As I explained in connection with limitation 1[e], Barsness teaches that monitoring system 110 monitors the tuple processing rates of each PE in the stream application. In my opinion, a POSITA would have recognized that a tuple processing rate of zero, or a sustained rate that drops below a meaningful threshold, would *identify execution that has become unresponsive*. A PE that is no longer producing output tuples or no longer processing input tuples is no longer making forward progress on its assigned computation. Thus, it would have not been responsive to incoming input data tuples. Barsness frames its monitoring system as addressing exactly this type of problem: it teaches that distributed stream applications create “challenges in term of application monitoring and detecting performance issues,” and that “monitoring and fault detection systems” must be designed to handle “fault detections [that] need to be

fixed in real-time.” EX1004, 3:7-15. The detection of unresponsive execution falls squarely within that fault-detection function.

145. Barsness alternatively teaches that monitoring system 110 acts on processing errors as well as tuple rates: “processing errors in a selected PE or operator may also be a factor on which the monitoring application 435 may act and reconfigure the processing modules of the stream application.” EX1004, 12:31-36. In my opinion, a POSITA would have recognized that a processing error in a PE indicates that the PE’s execution has become unresponsive, in the sense that the PE has stopped producing the expected output. Barsness further discloses tracing in log files as a monitoring mechanism that, in a POSITA’s understanding, would surface execution failures and unresponsive states. EX1004, 4:25-30; *id.*, 12:8-12 (describing log files associated with selected PEs that include “pre-configured data related to the data processing of data tuples at the selected PE”). The monitoring activities Barsness describes are accordingly directed at, among other things, *identify[ing] execution that has become unresponsive.*

146. This kind of unresponsive-execution detection through monitoring of per-component tuple flow was the conventional approach to fault detection in distributed stream processing systems known to a POSITA at the relevant time. *See* EX1011, 150 (describing how Storm’s Supervisor classifies workers as “timed out” when “the worker did not provide a heartbeat in the specified time frame, and is now

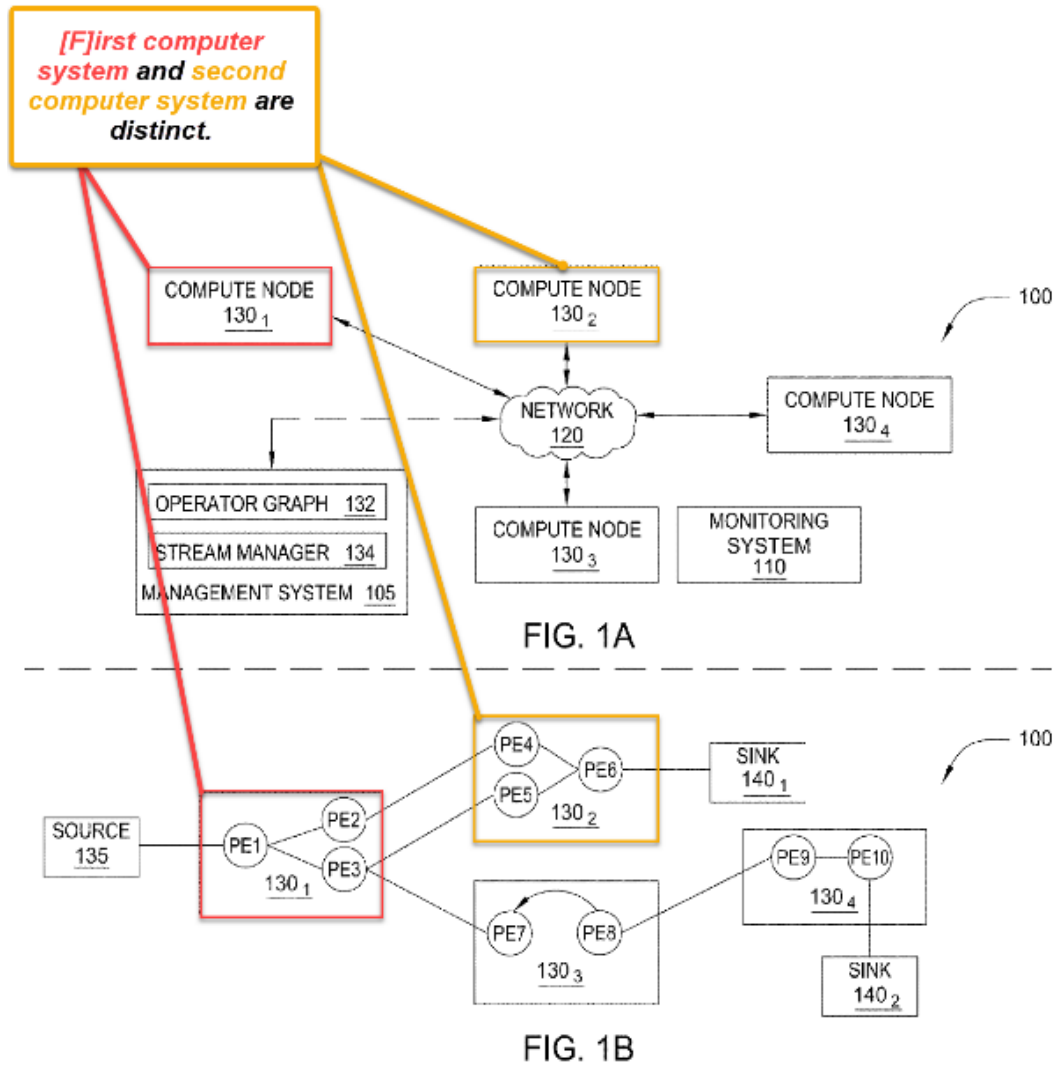
-108-

assumed to be dead”); *id.* (Storm Supervisor “monitors the health of the workers and respawns them if necessary”); EX1008, 242 (“[F]aults are no more exceptions but rather normal events that must be possibly dealt with during a vast majority of the executions.”); EX1009, 41 (describing a Storm-based system in which “after a supervisor went down, the scheduler successfully reassigned the task to another supervisor”). In my opinion, the monitoring system Barsness describes performs the same general function the prior art described: it observes per-PE execution metrics, detects when a PE is no longer functioning normally, and triggers corrective action.

3. Claim 3 – The system of claim 1, wherein the first computer system and the second computer system are distinct.

147. In my opinion, Barsness discloses Claim 3. Compute node 130₁ (*the first computer system*) and compute node 130₂ (*the second computer system*) are *distinct* computer systems. As I explained in connection with limitation 1[a], Barsness teaches a “computing infrastructure 100” that includes “a plurality of compute nodes 130₁₋₄, each connected to a communications network 120.” EX1004, 6:43-48. The plurality includes four physically distinct compute nodes 130₁, 130₂, 130₃, and 130₄, each of which I addressed earlier in this declaration. As I explained in connection with limitation 1[b], compute node 130₁ is *the first computer system*, and as I explained in connection with limitation 1[c], compute node 130₂ is *the*

second computer system. Figure 1A of Barsness depicts the computing infrastructure with its four physically distinct compute nodes:



EX1004, FIG. 1A. In my opinion, a POSITA would have understood that compute node 130₁ and compute node 130₂ are physically and functionally *distinct* computer systems.

4. Claim 4

(a) 4[pre] – The system of claim 1,

4[a] – wherein the first computer system and the second computer system each further comprise one or more memories that each store a respective second data,

148. In my opinion, Barsness discloses Claim 4. Compute node 130₁ (*the first computer system*) and compute node 130₂ (*the second computer system*) each comprise a memory 225 that stores data and a storage 230 that stores additional data, which together store the *respective second data*. Barsness states: “The CPU 205 retrieves and executes programming instructions stored in the memory 225. Similarly, the CPU 205 stores and retrieves application data residing in the memory 225.” EX1004, 8:53-55. Memory 225 itself contains processing elements 235, a data collector module 250, and a control module 255. EX1004, 8:65-9:1. Each compute node also includes “Storage 230, such as a hard disk drive, solid state device (SSD), or flash memory storage drive,” that may store non-volatile data. EX1004, 8:61-63. The non-volatile data stored in storage 230 includes the buffered stream data 260 that holds incoming and outgoing tuples for the PEs running on the compute node. EX1004, 9:43-50 (“Buffered stream data 260 represents a storage space for data flowing into the compute node 105 from upstream processing elements (or from a data source for the stream application).”). Figure 2 of Barsness depicts the memory 225 and storage 230 of a compute node:

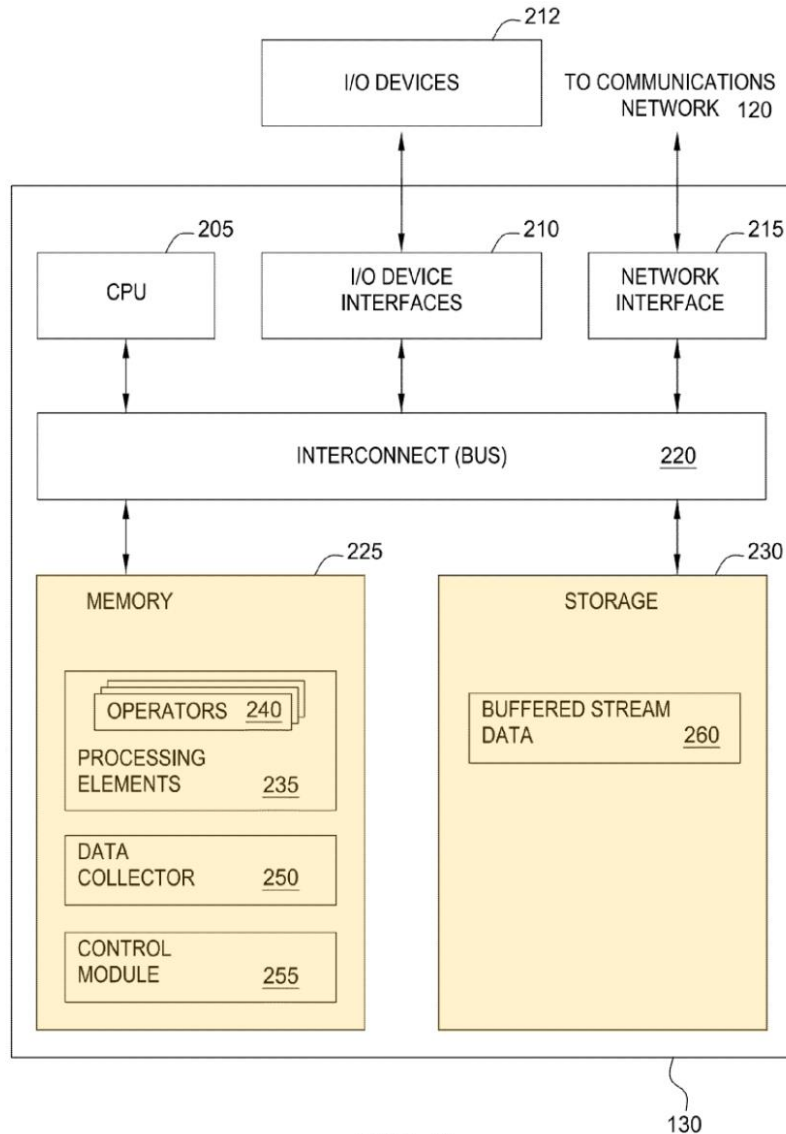


FIG. 2

EX1004, FIG. 2 (highlighting added). In my opinion, the data stored in each compute node's memory 225 and storage 230, including the PEs, the data collector module, the control module, and the buffered stream data, is the *respective second data*.

149. Thus, respective "data" stored in each respective memory of Barsness includes a *respective second data*.

(b) 4[b] – wherein the respective second data represents at least a respective portion of the distributed computational graph,

150. In my opinion, Barsness discloses limitation 4[b]. Each compute node 130 stores at least the portion of operator graph 132 corresponding to the PEs configured to run on that compute node. Within each compute node, memory 225 stores the PEs (*respective second data*): “In this example, the memory 225 includes one or more processing elements (PEs) 235.” EX1004, 8:66-9:1; *id.*, FIG. 2. Because each PE is itself a node of operator graph 132, and the PEs running on a given compute node form an interconnected subset of that graph, the data each compute node’s memory stores *represents at least a respective portion* of operator graph 132. EX1004, 7:52-67 (describing the per-compute-node arrangement of PEs in operator graph 132); *id.*, 1:39-41 (“[A] plurality of processing elements executing on one or more compute nodes.”); *id.*, 1:55-57 (same); *id.*, 2:3-4 (“The compute nodes are configured to execute processing elements of a stream application.”); *id.*, 9:1-33 (describing the configuration of PEs and operators on compute nodes, including the data collector 250 and control module 255 supporting that configuration); *id.*, 3:19-22 (“Scalability is achieved by distributing an application across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them.”); *id.*, 10:49-56 (describing how “the operator graph 132 maintains the connectivity configuration of

a plurality of compute nodes and processing elements in a streaming application” and how the “processing element and operator source code 345 is loaded to setup the processing elements 235 and the operators 240 in the compute nodes 130”). For example, Figure 1B of Barsness shows that compute node 130₁ contains PE1, PE2, and PE3, and accordingly stores data representing the portion of operator graph 132 that corresponds to those three PEs and their interconnections. The other compute nodes 130₂₋₄ similarly store data representing their respective portions of operator graph 132 as depicted in Figure 1B. EX1004, FIG. 1B.

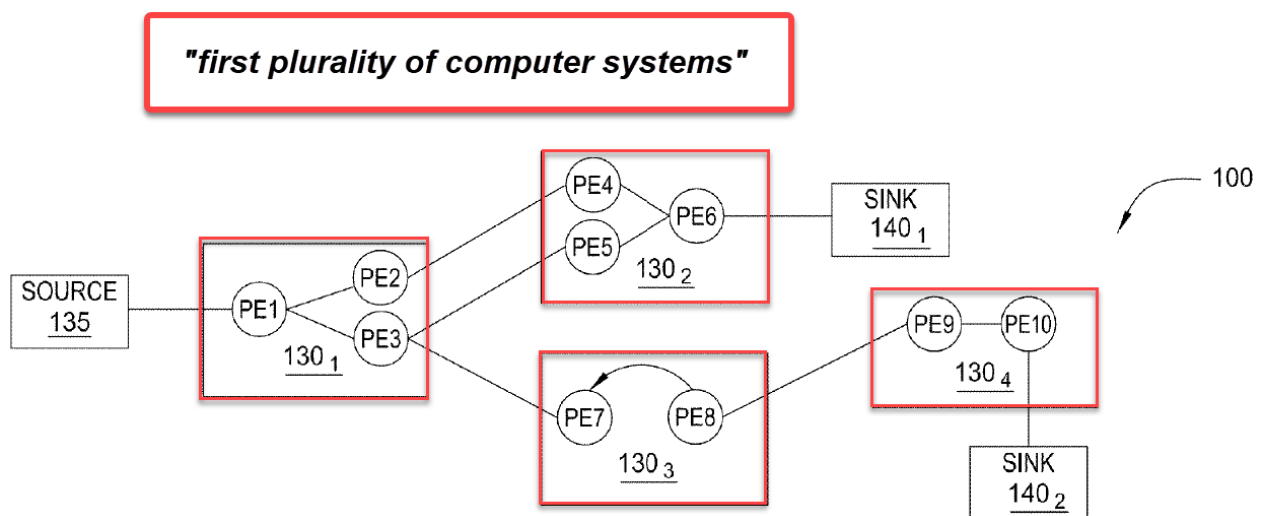


FIG. 1B

EX1004, FIG. 1B (annotations added).

151. As discussed regarding limitation 1[d][i], operator graph 132 *represents ... the distributed computation graph.*

- (c) 4[c] – and wherein the first software instructions transmit the first pipeline output messages according to the flow of the distributed computational graph.

152. In my opinion, Barsness discloses limitation 4[c]. Compute node 130₁ *transmits* the output messages produced by PE2 and PE3 to compute node 130₂ in accordance with the flow described in operator graph 132 (the *distributed computational graph*). EX1004, 3:16-31 (describing how operators in a stream application are connected by data flows and how processing elements can be inserted, removed, and replicated across nodes); *id.*, 6:43-56 (describing the computing infrastructure 100 in which compute nodes are connected via communications network 120); *id.*, 7:38-8:29 (describing the operator graph 132 example of FIG. 1B, the inter-process communication mechanisms including network sockets, the per-compute-node arrangement of PEs, and the data tuples flowing between PEs).

153. As I explained in connection with limitation 1[b][ii], compute node 130₁'s PE1, PE2, and PE3 apply the *first transformation pipeline*, which produces the *first pipeline output messages*. Figure 1B of Barsness, as annotated, depicts the first transformation pipeline:

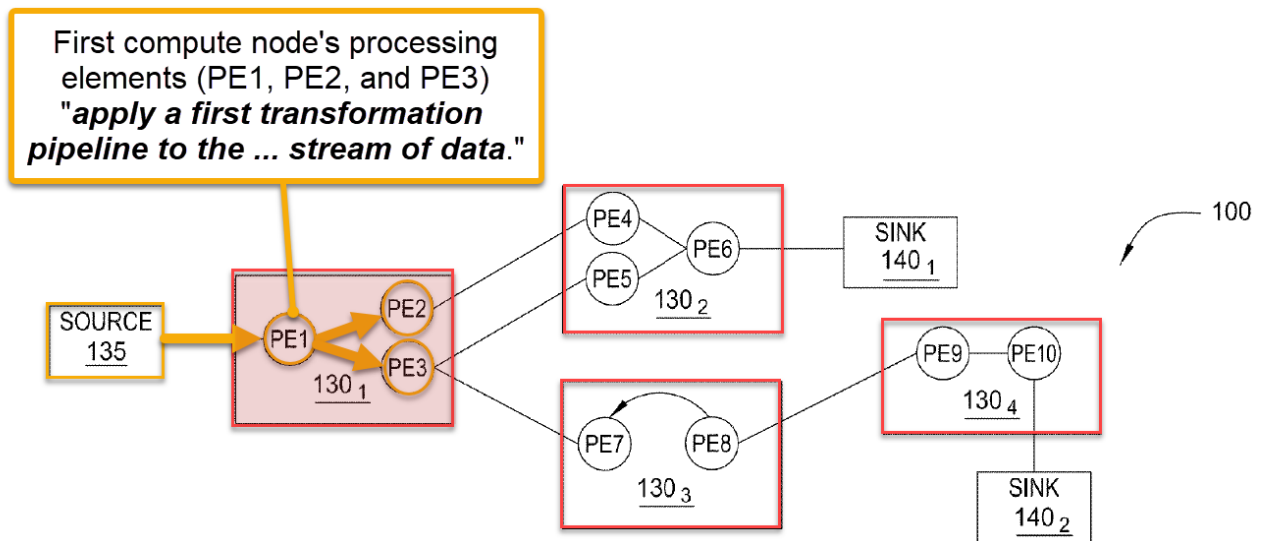


FIG. 1B

EX1004, FIG. 1B (annotations added).

154. As I explained in connection with limitation 1[b][iii], PE2 and PE3 emit data tuples (the *first pipeline output messages*) that compute node 130₁ *transmits* to compute node 130₂'s PE4 and PE5. Figure 1B of Barsness, as annotated, depicts that inter-node transmission:

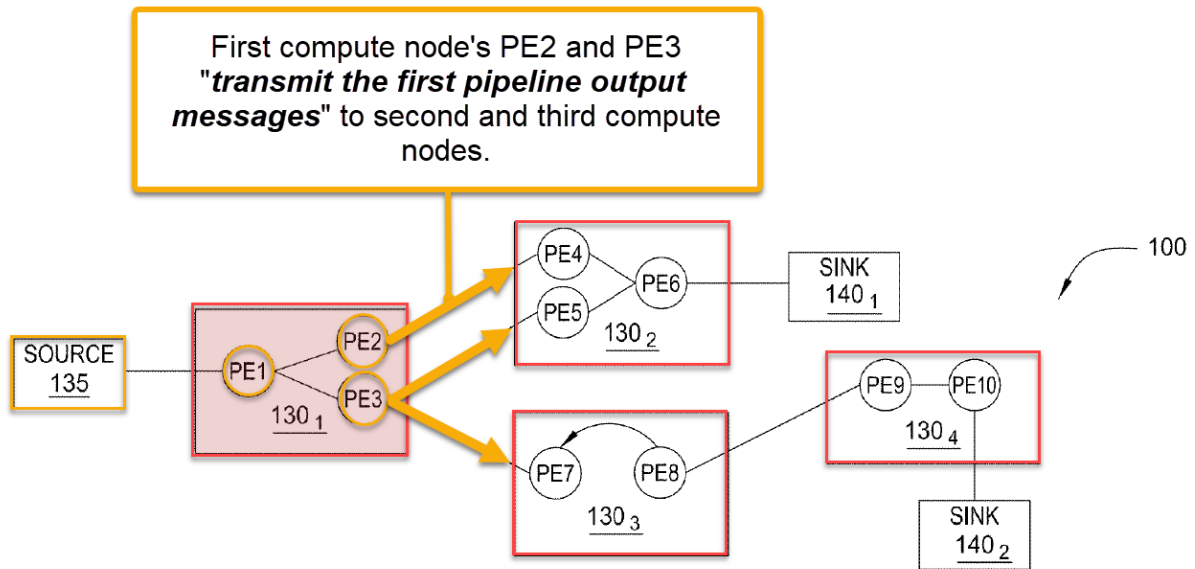


FIG. 1B

EX1004, FIG. 1B (annotations added).

155. As I explained in connection with limitation 1[b], Barsness teaches that each compute node's CPU executes the executable code, namely the *software instructions*, associated with the operators of the PEs running on that compute node. The CPU of compute node 130₁ "retrieves and executes programming instructions stored in the memory 225," and that memory stores the PEs and their constituent operators. EX1004, 8:53-54; *id.*, 8:66-9:14 (describing how memory 225 stores the PEs 235 with their collection of operators 240, and how each operator is "a small chunk of executable code configured to process data flowing into a processing element"); *see also id.*, 7:13-15 (describing how programming instructions are loaded onto the compute nodes for execution). In my opinion, the executable code that compute node 130₁'s CPU runs to operate PE1, PE2, and PE3, and that the CPU

uses to *transmit* the output messages from PE2 and PE3 to compute node 130₂, accomplishes the transmission *according to the flow of the distributed computational graph*. The flow of the operator graph dictates which downstream PEs receive the output messages from PE2 and PE3, and the executable code running on compute node 130₁ effectuates that flow by directing the data tuples to the corresponding inter-node connections (PE2 → PE4 and PE3 → PE5).

5. **Claim 5 – The system of claim 1, wherein the third software instructions are further configured to monitor at least a portion of the execution of at least one of the first software instructions and the second software instructions, to identify a transformation pipeline that causes at least one of the first computer system and the second computer system to transmit information that is outside established data stability boundaries.**

156. In my opinion, Barsness discloses Claim 5. As I explained in connection with limitation 1[f], Barsness teaches that monitoring system 110 monitors the tuple processing rates that measure the input and output of each PE in order to identify when those rates fall “outside a pre-configured range based on the historical tuple processing rate averages.” EX1004, 14:26-29. The pre-configured range Barsness describes, defined relative to the historical tuple processing rate averages and used by monitoring system 110 as the threshold for triggering corrective action, is the *data stability boundaries*. A POSITA would have recognized this as a well-known and conventional solution for a known problem

encountered with streaming data systems at the time. *See* EX1010, 110 (“[A] single bottleneck in the pipeline (congested link or an overloaded operator) can drastically impact the system throughput.”); *id.*, 75 (“To detect and diagnose temporally varying pipeline-bottlenecks, SEs need a feedback-driven control loop.”).

6. Claim 6 – The system of claim 5,

(a) 6[a] – wherein the third computer system of the second plurality of computer systems is further configured to execute fifth software instructions that, upon identifying transmitted information that is outside established data stability boundaries:

6[b] – determine changes to at least one of the first transformation pipeline and the second transformation pipeline in response to the identifying,

6[c] – and cause at least one of the first computer system and the second computer system to execute sixth software instructions that modify a transformation pipeline in response to the determined changes.

157. In my opinion, Barsness discloses Claim 6. As I explained in connection with limitation 1[f] and Claim 5, monitoring system 110 (*the third computer system*) identifies when the tuple processing rates fall outside the pre-configured range based on the historical tuple processing rate averages (*data stability boundaries*). EX1004, 14:26-29. The changes monitoring system 110 directs to the PEs and operators of compute node 130₁ or 130₂ are the *fifth software*

instructions, and they are operations distinct from the spawning or duplicating of compute nodes I addressed earlier in connection with limitation 1[f].

158. Barsness teaches that monitoring system 110 determines the corrective actions it directs (*changes... in response to the identifying*) by analyzing the performance issues identified through the per-PE tuple processing rates. Barsness states:

the monitoring application 435 may interact with the selected processing element to determine if adding more operators 240 may remedy the issue. Alternatively, the monitoring application 435 may interact with the computer node 130 in which the selected processing element is running to determine if adding more processing elements to process the same type of data as being processed by the selected processing element may remedy the issue. The monitoring application 435 may also interact with the management system 105 to determine if rearranging some processing elements on computer nodes 130 may remedy the issue.

EX1004, 12:56-13:3.

159. Barsness further discloses that monitoring system 110 directs corrective actions to the relevant PEs that include “automatic adjustment of an operator within compute nodes 130, automatic fusion or separation of operators, automatic duplication of an operator,...spawning new processing elements,” “add[ing] one operator in the processing element in question,” and “reconfigur[ing] the processing modules” of the stream application. EX1004, 14:51-56; *id.*, 13:24-29; *id.*, 12:31-36; *see also id.*, 14:35-39 (describing how “[t]he corrective measures may include adjusting the run-time priority of a particular processing job in the stream application to bring the current tuple processing rates within the pre-configured range”). These changes are implemented at compute node 130₁ or compute node 130₂ by the control module 255, which Barsness describes as the mechanism by which monitoring system 110 effectuates per-compute-node configuration changes. The control module 255 is the *sixth software instructions*. Barsness explains:

The control module 255 may be included in the compute node 130 to enable the monitoring module 110 to configure the computer node 130 based on the occurrence and detection of pre-configured conditions in the computer node 130. For example, during the monitoring of a stream application, if the monitoring system 110 determines that more processing elements 235 or operators 240 need to be

added, the monitoring system 110 may send a message to the control module 255 to effect the desired change in the compute node 130.

EX1004, 9:34-42. In my opinion, the chain of activity Barsness describes—monitoring system 110 detects that tuple processing rates have fallen outside the pre-configured range; identifies the corrective action needed; sends a message to the control module 255 on compute node 130₁ or 130₂; and the control module implements the change to the PEs or operators on that compute node—satisfies the modification-in-response-to-identification of Claim 6. The kind of feedback-driven topology modification Barsness describes was conventional in distributed stream processing systems known to a POSITA at the relevant time. *See, e.g.*, EX1010, 75-76.

B. Ground 2: Claims 1-6 Are Obvious Over Barsness in View of Chakradhar

160. I provide a summary of this ground as follows. As explained above, Barsness alone renders claims 1-6 obvious. These claims are further obvious in view of Chakradhar, which expressly teaches using route-maps and explains the benefits of doing so.

1. Motivation to Combine and Reasonable Expectation of Success

161. In my opinion, a POSITA would have been motivated, with a reasonable expectation of success, to combine the computing infrastructure for real-time stream processing applications taught by Barsness with the route-map teachings of Chakradhar to achieve “accelerated processing of stream processing engines” and to “address[] bottlenecks in stream engines.” EX1005, [0006]; EX1006, 4/22.

162. A POSITA implementing a system like the one Barsness teaches would have been motivated to incorporate Chakradhar’s route-map mechanism. Barsness requires operators to emit output to downstream PEs according to operator graph 132, but Barsness does not prescribe any specific internal data structure for locally storing and updating routing information at each operator. EX1004, 9:4-9 (describing how each operator emits data “to other operators 240 in a PE and to other PEs in the stream application as described by the operator graph 132,” without specifying the local routing data structure). Chakradhar addresses this design need by explicitly teaching operator-local storage and processing of routing data structures that represent at least portions of the distributed topology. EX1005, [0122]; EX1006, 9/22. In my opinion, a POSITA would have recognized that Chakradhar’s route-maps provide a natural and efficient way to implement the routing behavior Barsness requires.

163. The benefits Chakradhar identifies align directly with the goals and challenges Barsness describes. Chakradhar's ability to update routing behavior on the fly and atomically across operators supports fast reconfiguration, scalability, and fault tolerance, which Barsness identifies as important characteristics for stream processing systems that must rebalance load or move processing elements between machines. EX1004, 3:56-60; *id.*, 10:28-39; EX1005, [0054]; EX1006, 5/22. In my opinion, a POSITA would therefore have expected that incorporating Chakradhar's route-maps into Barsness's computing infrastructure would complement and improve the scalability, adaptability, and robustness of Barsness's system. EX1004, 3:3-6 (describing the "desired results with regards to scalability, efficiency, and real-time data processing"); *id.*, 3:13-15 (describing the "desired results where fault detections need to be fixed in real-time"); *id.*, 3:19-22 (describing how scalability is achieved by distributing the application across nodes and replicating processing elements with load balancing).

164. Applying Chakradhar's route-maps to Barsness would have allowed each compute node and monitoring system 110 to store respective data representing a portion of a distributed computational graph, and to process that data locally to determine the routing of output messages to downstream PEs. Because Chakradhar's route-maps are expressly designed for distributed operator-based stream processing systems, in my opinion a POSITA would have had a reasonable expectation of

-124-

success in making this modification. EX1005, [0043], [0056]; EX1006, 4/22, 6-7/22.

The combination would have predictably reduced the overhead of managing scalable stream applications and enabled efficient, consistent routing updates with minimal disruption to in-flight data streams. EX1005, [0054]-[0055]; EX1006, 5/22.

165. In my opinion, this combination of Barsness and Chakradhar would include the features of the cloud-based cluster for distributed stream data processing applications Barsness teaches, with the operators storing and maintaining route-maps that, in Chakradhar's words, "encode tuple-routing information and also the topology link structure." EX1005, [0054]; *see also id.*, [0009]-[0010], [0053], [0122]; EX1006, 5/22, 9/22. The kind of feedback-driven, on-the-fly routing modification that the Barsness-Chakradhar combination would produce was the conventional approach to dynamic optimization in distributed stream processing systems known to a POSITA at the relevant time. *See, e.g.*, EX1010, 75-76, EX1014, 250 (describing how operators in distributed stream processing systems use FIFO semantics for routing data items between operator output and input ports). So implemented, in my opinion the Barsness-Chakradhar combination renders the challenged claims obvious.

2. Claim 1

166. In my opinion, the Barsness-Chakradhar combination satisfies this claim. As discussed *supra* Section VII.A, claim 1 is obvious in view of Barsness alone. Claim 1 is further obvious in view of the Barsness-Chakradhar combination.

(a) 1[a] – a distributed computing network comprising a first plurality of computer systems, a second plurality of computer systems, and a third plurality of computer systems,

167. In my opinion, the Barsness-Chakradhar combination discloses limitation 1[a]. Chakradhar describes stream processing engines such as Storm, which it characterizes as “a distributed, real-time computation system” that “makes it possible to process unbounded streams of data.” EX1005, [0042], [0056]; *id.*, FIG. 13 (depicting a cloud-based stream processing system with multiple incoming data streams); *id.*, [0134]; EX1006, 4/22. Chakradhar explains that “[o]n a Storm cluster, topologies are executed and process streams of tuples (i.e., data).” EX1005, [0056]; *see also id.*, [0046] (“The orchestration of cluster resources among these systems is handled by a global cluster scheduler such as Mesos.”), [0069] (describing how route-maps are installed in a consistent manner on a running cluster), [0098] (referring to “nodes in a cluster”), [0148] (describing an evaluation of the techniques conducted on a 30-node cluster); EX1006, 4/22, 6-7/22, 11/22. The clusters Chakradhar describes are composed of distributed computer systems that

communicate over a network. EX1005, [0069]. In my opinion, the Barsness-Chakradhar combination accordingly includes a *distributed computing network*.

(b) 1[d][i] – wherein the first data represents a portion of a distributed computational graph,

168. In my opinion, the Barsness-Chakradhar combination discloses limitation 1[d][i]. Chakradhar explains that “stream engines code an application workflow as a directed acyclic graph (DAG) of operators, referred to as a topology.” EX1005, [0043]; EX1006, 4/22. Chakradhar further teaches that stream data is “processed in a pipelined fashion as they traverse through the topology.” EX1005, [0043]; EX1006, 4/22. Chakradhar describes the topology as graph-structured: “The logic for realtime applications is packaged in a topology ... [e]ach topology is represented by a graph including spouts (which produce tuples) and bolts (which transform tuples).” EX1005, [0056]. Chakradhar elsewhere reinforces that this topology is, structurally, a directed graph: “a topology is a graph whose nodes are operators (spouts and bolts) and edges are virtual connections among operators.” EX1005, [0056]; *see also id.*, [0106] (describing “a complete directed graph, where nodes denote machines hosting learner operators and edges represent potential overlay links”); EX1006, 5/22, 8/22. In my opinion, the topology Chakradhar describes is a *distributed computational graph* within the meaning of the limitation: it is a graph because its components are nodes and edges; it is computational because

each node represents a transformation that operates on incoming tuples; and it is distributed because the operators are hosted on separate compute nodes that communicate over a network.

3. Claim 4

169. In my opinion, the Barsness-Chakradhar combination discloses Claim 4. Chakradhar teaches that each operator stores routing data that specifies how output messages flow to downstream operators. Chakradhar explains: “Every operator chooses a destination operator for its outgoing tuples based on route-maps.” EX1005, [0069]; *id.*, [0070] (describing tuple-routing between operators); EX1006, 5-6/22, 14/22. Chakradhar further explains that “[r]oute maps contain information on the type and proportion of traffic for each destination operator.” EX1005, [0069]; EX1006, 6/22. Most directly relevant to the limitation, Chakradhar discloses that “[e]ach operator maintains a route-map that specifies the routes on which output messages should be sent after processing incoming messages.” EX1005, [0122]; EX1006, 9/22. The route-maps Chakradhar teaches accordingly “represent[] topology link structure” of the distributed computational graph. EX1005, [0053]; EX1006, 5/22. When Chakradhar’s route-maps are updated, they are atomically injected into the operators “for fast topology re-optimization.” EX1005, [0010], [0054]; EX1006, 5/22, 14/22. In my opinion, the per-operator route-maps Chakradhar describes are the *respective second data*.

170. In the Barsness-Chakradhar combination, each compute node stores both (a) the operators of the PEs configured to run on that compute node, namely the executable code corresponding to the tasks assigned to those PEs as Barsness teaches (EX1004, 9:4-9), and (b) the route-maps specifying where “output messages should be sent after processing incoming messages” as Chakradhar teaches (EX1005, [0122]; EX1006, 9/22). In my opinion, the data each compute node stores in the combination, comprising the operators plus the route-maps for those operators, *represents at least a respective portion of a distributed computational graph*: the operators are the graph nodes (with their associated transformations), and the route-maps encode the graph edges (the connections from each operator’s output to the inputs of downstream operators). The combination accordingly satisfies Claim 4.

C. Ground 3: Claims 1-6 Are Obvious Over Barsness in View of Siripurapu

171. As explained above, Barsness alone renders claims 1-6 obvious. These claims are further obvious in view of Siripurapu, which provides further details of how the claimed monitoring and provision of additional computer systems was known in the art. Specifically, Siripurapu describes monitoring data flows within a distributed computing system and adding computer nodes to accommodate increased computational requirements, additional data streams, and increased data stream flows.

1. Motivation to Combine and Reasonable Expectation of Success

172. In my opinion, a POSITA would have been motivated, with a reasonable expectation of success, to implement the real-time stream processing applications taught by Barsness with the scalability teachings of Siripurapu to “accommodate increased computational requirements, additional data streams, increased data stream flows, and the like.” EX1007, [0453]. I explain my reasoning in the paragraphs that follow.

173. Barsness expressly teaches scaling a distributed stream processing application to address data processing needs. EX1004, 3:19-22 (“Scalability is achieved by distributing an application across nodes by creating many small executable pieces of code (operators), as well as replicating processing elements on multiple nodes and load balancing among them.”); *id.*, 6:53-55 (“The operator graph 132 is configurable to add or remove one or more PEs based on the data processing needs and scalability.”). Siripurapu teaches the same type of scaling in the same stream processing context, and provides additional details on how and when to scale by dynamically adding worker nodes: “This distributed implementation provides scalability. Wherein one or more additional worker nodes may be added to perform additional map and update operations and to accommodate increased computational requirements, additional data streams, increased data stream flows, and the like.”

EX1007, [0453]. In my opinion, a POSITA would have recognized Siripurapu's teachings as directly aligning with, and easily integrated into, the scalable infrastructure of Barsness, which distributes processing elements across compute nodes and adds or removes processing elements based on "data processing needs and scalability." EX1004, 6:53-55.

174. The scalability teachings of Siripurapu reinforce the scalability and monitoring teachings of Barsness regarding "replicating processing elements on multiple nodes and load balancing among them" (EX1004, 3:19-22) and spawning or provisioning additional compute nodes based on monitoring (EX1004, 6:19-42, 10:26-39). Like the monitoring system 110 of Barsness, Siripurapu's conductor "may have visibility into the performance of the worker nodes and may perform load balancing by reassigning tasks to the worker nodes." EX1007, [0448]. And like the spawning of new compute nodes in Barsness, Siripurapu teaches that the conductor "may assign stream processing operations to the new worker nodes" provisioned to "accommodate increased computational requirements, additional data streams, increased data stream flows, and the like." EX1007, [0453]. It is my opinion that these teachings would have motivated a POSITA to combine the Siripurapau and Barsness references to increase efficiency and processing rates.

175. In my opinion, to the extent Barsness alone did not expressly teach doing so, a POSITA would have been motivated in view of Siripurapu to configure

-131-

Barsness's infrastructure to scale the processing nodes by adding additional or duplicate nodes to handle additional data streams from other applications based on monitoring those existing nodes. The two teachings are complementary, and in my opinion a POSITA would have recognized the computational efficiency improvements resulting from having combined them to create an infrastructure capable of monitoring the per-node performance metrics of a streaming application and dynamically adapting the compute resources to meet the desired performance of the system.

176. Barsness already discloses a scalable, cloud-based computing infrastructure in which compute nodes can be dynamically provisioned and duplicated to handle data processing needs. EX1004, 6:16-42 (describing the cloud computing infrastructure for the "provision of scalable computing resources"), 3:19-22. In my opinion, a POSITA would accordingly have had a reasonable expectation of success in combining Barsness's teachings with Siripurapu's. The combination would include the cloud-based cluster for distributed stream data processing applications Barsness teaches, including scalability through the duplication and provisioning of nodes, with Siripurapu's teachings on scaling and adding worker nodes to accommodate additional data streams based on monitoring the load of those nodes. EX1007, [0453], [0496]. In my opinion, this Barsness-Siripurapu combination renders the challenged claims obvious.

177. The kind of monitoring-driven, elastic scaling that the Barsness-Siripurapu combination would produce was the conventional approach to capacity management in distributed stream processing systems known to a POSITA at the relevant time. *See* EX1008, 240 (“Elasticity describes the behavior of a system to dynamically scale in or out based on a changing workload.”), 240 (describing cloud-based stream processing systems as “designed to dynamically scale to hundreds of computing nodes and automatically cope with varying workloads”); EX1012, 173 (describing the standby pool of nodes in S4 from which spawned nodes can be drawn); EX1010, 95 (describing the feedback-driven control loop required to detect and address bottlenecks in distributed stream processing systems).

2. Claims 1-6

178. In my opinion, the Barsness-Siripurapu combination discloses Claims 1-6. As I explained earlier in this declaration, Claims 1-6 are obvious in view of Barsness alone. Claims 1-6 are further obvious in view of the Barsness-Siripurapu combination because, in my opinion, a POSITA would have found it obvious to implement Barsness’s monitoring system 110 in a way that causes the provisioning of additional compute nodes to handle increased data volume or processing needs at a given compute node or transformation pipeline, based on monitoring of those nodes, in light of Siripurapu’s scalability teachings. EX1007, [0453] (“[O]ne or more additional worker nodes may be added to perform additional map and update

operations and to accommodate increased computational requirements, additional data streams, increased data stream flows, and the like ... [t]he conductor may assign stream processing operations to the new worker nodes and may transmit stream events to the new worker nodes.”), [0448] (“The conductor 3612 may have visibility into the performance of the worker nodes and may perform load balancing by reassigning tasks to the worker nodes.”). The combination implements exactly the kind of monitoring-driven, elastic provisioning that Claims 1-6 describe: monitoring system 110 (in the Barsness role) or the conductor (in the Siripurapu role) detects per-node performance issues, identifies the need for additional capacity, and causes the provisioning of additional compute nodes to address that need.

(a) 1[e] – wherein the third computer system is configured to execute third software instructions that:

1[e][i] – process the first data to determine information about at least one of the first transformation pipeline and the second transformation pipeline,

179. In my opinion, the Barsness-Siripurapu combination discloses these elements. Siripurapu teaches that the conductor maintains per-node metadata that includes the kind of operational, status, and load information that Barsness’s monitoring system 110 likewise tracks. Specifically, Siripurapu explains: “The conductor 3612 may maintain metadata about the available worker nodes in the system including, but not limited to, the computational specifications of the worker nodes, the current status of each worker node (e.g., whether it is operational, is in a

failed state, etc.), the current processing load at each worker node, the data streams from which each worker node are subscribed to receive stream events, [and] the data streams to which each worker node publishes events.” EX1007, [0449]. The metadata Siripurapu’s conductor maintains is the *first data* within the meaning of the limitation. Siripurapu further teaches that the conductor periodically determines load data for the worker nodes by polling them. EX1007, [0470] (“[T]he conductor may periodically determine load data for the worker nodes, e.g., by polling the worker nodes.”), [0496] (describing how the conductor receives current computational load information by polling the worker nodes “at pre-defined intervals of time”), [0509] (describing how the conductor “may poll the worker nodes at pre-determined intervals of time for load data and to detect failures”). In my opinion, a POSITA would have found it obvious to implement the per-node status and load tracking that Siripurapu’s conductor performs in the monitoring system 110 of Barsness, so that monitoring system 110 monitors the processing load at each compute node 130 in the same manner Siripurapu describes for worker nodes, while continuing to perform Barsness’s tuple-rate-based monitoring of operational PEs.

(b) Claim 2

180. In my opinion, the Barsness-Siripurapu combination discloses Claim 2. Siripurapu teaches that the conductor polls each worker node at regular intervals for

both load data and failure detection: “A conductor may poll the worker nodes at pre-determined intervals of time for load data and to detect failures.” EX1007, [0509]; *see also id.*, [0449] (describing how the per-node metadata the conductor maintains includes “the current status of each worker node (e.g., whether it is operational, is in a failed state, etc.)”). In my opinion, a POSITA would have found it obvious to implement Siripurapu’s per-node failure detection in Barsness’s monitoring system 110 to *identify execution that has become unresponsive*. Siripurapu’s conductor and Barsness’s monitoring system 110 perform analogous roles in their respective architectures: each communicates with the per-node components, gathers metrics about per-node performance, and takes action when the metrics indicate a problem. The natural extension of that parallel is to incorporate Siripurapu’s explicit failure-detection mechanism into Barsness’s monitoring system 110, so that monitoring system 110 not only tracks the tuple processing rates of operational PEs but also detects when a compute node has become unresponsive (in a failed state, in Siripurapu’s terms). The kind of polling-based unresponsiveness detection that Siripurapu describes was the conventional approach to fault detection in distributed stream processing systems known to a POSITA at the relevant time. *See* EX1011, 150 (describing how Storm’s Supervisor classifies workers as “timed out” when “the worker did not provide a heartbeat in the specified time frame, and is now assumed

to be dead”); EX1008, 242 (“[F]aults are no more exceptions but rather normal events that must be possibly dealt with during a vast majority of the executions.”).

D. Ground 4: Claims 1-6 Are Obvious Over Barsness in View of Chakradhar and Siripurapu

181. As I explained earlier in this declaration, Barsness alone renders Claims 1-6 obvious. For the same reasons a POSITA would have been motivated to combine the real-time stream processing system of Barsness with the teachings of Chakradhar and Siripurapu individually (each discussed in the corresponding Ground sections of this declaration), in my opinion a POSITA would have applied those teachings together when implementing the Barsness system. The Chakradhar and Siripurapu teachings address different aspects of the distributed stream processing system, are fully compatible with one another, and are complementary. Chakradhar’s teachings relate to storing per-operator routing information so that each compute node can efficiently transmit transformed data to the correct downstream PEs, and Siripurapu’s teachings provide additional details on scaling the infrastructure to accommodate varying workload needs. In my opinion, a POSITA would have been motivated, with a reasonable expectation of success, to implement both sets of teachings simultaneously within the Barsness system. Accordingly, in my opinion the challenged claims are further obvious in view of the combination of Barsness, Chakradhar, and Siripurapu.

VIII. SECONDARY CONSIDERATIONS OF NON-OBVIOUSNESS

182. I am not aware of any secondary considerations, or so-called “objective indicia of non-obviousness” for the challenged claims.

183. Secondary considerations could include things like commercial success of a product due to the merits of the claimed invention; a long felt need for the solution provided by the claimed invention; unsuccessful attempts by others to find the solution provided by the claimed invention; copying of the claimed invention by others; unexpected and superior results from the claimed invention; acceptance by others of the claimed invention as shown by praise from others in the field or from the licensing of the claimed invention; teaching away from the conventional wisdom in the art at the time of the invention; independent invention of the claimed invention by others before or at about the same time as the named inventor thought of it; and other evidence tending to show obviousness.

184. As discussed throughout, the claimed features reflect predictable results of combining known elements according to their known functions, and thus, there are no unexpected and superior results from the claimed invention.

185. Further, even if Patent Owner were to assert that secondary considerations exist, given the strong reasons that the challenged claims are obvious in light, including the motivations to combine the references set forth above, I do not believe any such secondary considerations would rise to the level of overcoming the

invalidity opinions I have expressed.

186. I understand that Patent Owner may address the issue of secondary considerations more fully in the future. I reserve the right to respond to any specific bases for such secondary considerations that Patent Owner may identify.

IX. CONCLUSION

187. For the reasons set forth above, I believe claims 1-6 of the '425 patent are unpatentable in view of the prior art. In signing this declaration, I understand that the declaration will be filed as evidence in a contested case before the Patent Trial and Appeal Board of the United States Patent and Trademark Office. I acknowledge that I may be subject to cross-examination in this case and that cross-examination will take place within the United States. If cross-examination is required of me, I will appear for cross-examination within the United States during the time allotted for cross-examination.

188. I declare that all statements made herein of my knowledge are true, that all statements made on information and belief are believed to be true, and that these statements were made with the knowledge that willful false statements and the like so made are punishable by fine or imprisonment, or both, under Section 1001 of Title 18 of the United States Code.

Declaration of Paul S. Min Ph.D.
U.S. Patent No. 12,143,425

Dated: May 26, 2026

By: 

Paul S. Min, Ph.D.