Bulletin of the Technical Committee on

Data Engineering

March 2003 Vol. 26 No. 1

IEEE Computer Society

Letters

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

The Data Engineering Bulletin web page is http://www.research.microsoft.com/research/db/debull.

Letter from the Editor-in-Chief

Ten Years as Editor-in-Chief

In 1992, Rakesh Agrawal, then chair of the TC on Data Engineering, asked me to be the editor-in-chief of the Data Engineering Bulletin. I paused before accepting this offer. I knew that this was a substantial undertaking. But, if I can be permitted to ”toot my own horn” a bit, I think that the effort have been largely successful.

A good way of judging the credibility of the Bulletin is to see who has contributed toward its success. This space is too short to list them, but I am very gratified by the success of our editors in enlisting authors from the best academic and industrial settings of our field, leading practitioners all. However, the space is sufficient to list the associate editors who have served the Bulletin so well during my tenure as editor-in-chief. It is these editors who, issue after issue, plan and solicit the technical content. I am proud to have recruited these outstanding people to serve as editors: Daniel Barbara, Surajit Chaudhuri, Umeshwar Dayal, Amr El Abbadi, Ahmed Elmagarmid, Michael Franklin, Shahram Ghandeharizadeh, Johannes Gehrke, Goetz Grafe, Luis Gravano, Alon Halevy, Joe Hellerstein, Meichun Hsu, Yannis Ioannidis, Christian Jensen, Donald Kossmann, Renee Miller, Eliot Moss, Elke Rundensteiner, Betty Salzberg, Sunita Sarawagi, Gerhard Weikum, Kyu-Young Whang, and Jennifer Widom. In addition, Divyakant Agrawal, Fabio Casati, Sharma Chakravarthy, and Ron Obermarck served as single issue editors. Thank you all for consistently doing a fine job.

Finally, I want to thank people who have contributed in other ways. Mark Tuttle twice produced latex style files and prototype issue frameworks that enable us to generate the Bulletin both electronically and in hardcopy. The Bulletin could not be published in its current form without Mark’s efforts. Rakesh Agrawal, Betty Salzberg, and Erich Neuhold, as TCDE chairs, provided unwavering support, and have worked to assure that the Bulletin can continue to be successfully published. The IEEE Computer Society staff has been helpful in many respects, especially in providing membership lists and coordinating the hardcopy publication of the Bulletin.

The Current Issue

Web services are rapidly becoming a part of our collective landscape. We needn’t wonder whether we can purchase something on-line. The answer is clearly “yes” almost all the time. B2C e-commerce is florishing. Certain forms of B2B e-commerce are likewise commonplace. Less common, however, is the integration of web services across enterprises, programs interacting with programs as opposed to programs interacting with people. Further, there are many potential ”gotcha’s” to deal with, security, privacy, reliability, availability, etc.. Also, robust web services have been notoriously difficult to implement, with problems including the preceding ”ities”, simplifying application programming, providing standards for metadata and protocols, and integrating specific applications with generic web services provided by other vendors.

This issue discusses many of the problems associated with web services and their implementation and exploitation. As is characteristic of articles by technical professionals, be they product developers or researchers, the emphasis is on frameworks, architectures and generic approaches as opposed to specific web service instances. This should not be a surprise given that the authors are from universities or software providers, not application specialists. And, indeed, the real technical leverage is frequently via such a generic focus. Nonetheless, the current issue ends with examples of useful web services. Fabio Casati, as guest editor, and Umesh Dayal have thus captured a broad snapshot of the current state of web services. Bringing this information together in one place and in a timely fashion is a significant contribution to the dissemination of this technology. So I want to thank Fabio and Umesh for their thorough and timely editorial work. This area, an evolution of transaction processing, can be expected to remain important for years. The current issue should thus be a worthwhile source of information for a long time.

David Lomet
Microsoft Corporation

Letter from the Special Issue Editor

Traditional Database Management Systems (DBMS) software is built on the concept of persistent data sets, that are stored reliably in stable storage and queried several times throughout their lifetime. For several emerging application domains, however, data arrives and needs to be processed continuously, without the benefit of several passes over a static, persistent data image. Potential applications for data stream management range from network monitoring, sensor networks, financial data, to the real-time enterprise.

In January 2003, a group of researchers met at Stanford University for an informal workshop to discuss ongoing and open research questions in query processing for streaming data, and to ponder about the killer application for a data stream management system. While this special issue is not an official report from the workshop, at least one of the authors of each invited papers were present at the workshop, and the selection of papers mirrors some of the exciting work that was presented at the meeting. I encourage the interested reader to visit `http://telegraph.cs.berkeley.edu/swim/` for more information.

Research in data stream processing spans many communities with different research foci. This issue concentrates on some of the ongoing work in the database community. We start this issue with overview articles from four groups that are currently building prototype data stream processing systems. Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur C etintemel, Magdalena Balazinska, and Hari Balakrishnan survey the Aurora and Medusa Projects at MIT, Brandeis University, and Brown University. Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel R. Madden, Fred Reiss, and Mehul Shah give us an update of the latest status of the TelegraphCQ Project at Berkeley. The Stanford STREAM Group — at the writing of this article consisting of Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas, Rohit Varma, and Jennifer Widom — describes the status of their system and ongoing research. Chuck Cranor, Theodore Johnson, Oliver Spatscheck, and Vladislav Shkapenyuk from AT&T Research give an industry perspective; they survey challenges in the design and implementation of the Gigascope stream database system specialized for network monitoring.

The issue continues with two articles that cover algorithmic issue in data stream processing. Peter Tucker, David Maier, and Tim Sheard describe how punctuations — predicates that hold about a data stream — can be used to improve data stream queries. Yanlei Diao and Michael Franklin describe an XML publish-subscribe query processor that matches streaming XML data with a large number of continuous queries. We conclude this issue with an article by Alberto Lerner and Dennis Shasha which discusses a novel architecture of a data stream system based for data from the financial industry on integrating online stream processing with support for historical queries over a data archive.

I hope that you find this special issue as enjoying to read as I did, and I thank the authors for their contributions.

Johannes Gehrke
Cornell University
Ithaca, NY 14853

The Aurora and Medusa Projects

Stan Zdonik
Brown University
sbz@cs.brown.edu

Michael Stonebraker
MIT
stonebraker@lcs.mit.edu

Mitch Cherniack
Brandeis University
mfc@cs.brandeis.edu

Uğur Çetintemel
Brown University
ugur@cs.brown.edu

Magdalena Balazinska
MIT
mbalazin@lcs.mit.edu

Hari Balakrishnan
MIT
hari@lcs.mit.edu

Abstract

This document summarizes the research conducted in two interrelated projects. The Aurora project being implemented at Brown and Brandeis under the direction of Uğur Çetintemel, Mitch Cherniack, Michael Stonebraker and Stan Zdonik strives to build a single-site high performance stream processing engine. It has an innovative collection of operators, workflow orientation, and strives to maximize quality of service for connecting applications. A further goal of Aurora is to extend this engine to a distributed environment in which multiple machines closely co-operate in achieving high quality of service.

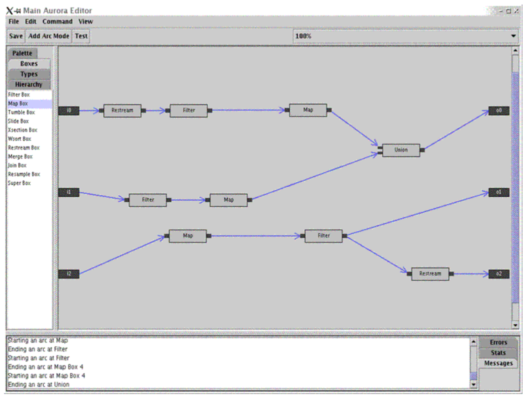
In contrast, the Medusa project being investigated at M.I.T. under the direction of Hari Balakrishnan and Michael Stonebraker is providing networking infrastructure for Aurora operations. In addition, Medusa is stressing distributed environments where the various machines belong to different organizations. In this case, there can be no common goal, and much looser coupling is needed. Medusa is working on an innovative agoric infrastructure in which various participants can co-operate in distributed streaming operations.

Finally, some have questioned whether specialized stream processing software is necessary. They speculate that conventional data base systems can adequately deal with the needs of stream-oriented applications. In order to answer this question one way or the other, both groups are working on a stream-oriented benchmark, called the Linear Road benchmark, which we intend to run on both specialized and conventional system infrastructure.

1 Introduction

Many people have pointed out the rationale for stream-oriented storage systems. Rather than repeat their observations, we want to note just two points here. Business intelligence is typically performed by extracting relevant data from operational systems, transforming it into a common representation, and then loading it into a data warehouse. Business analysts can then run data mining queries against the warehouse to find information of interest, on which to base changes to business practices. Business analysis through warehousing and Extract-Transform-and Load (ETL) techniques is widely used in large enterprises at the current time. However,

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering



```
Starting an arc at Map
Ending an arc at Filter
Starting an arc at Filter
Ending an arc at Map Box 4
Starting an arc at Map Box 4
Ending an arc at Union
```

Figure 1: The Aurora GUI

warehouses are out-of-date by 1/2 of the refresh interval on average, typically 24 hours. Hence, business analysis is similarly out-of-date.

A widespread goal is ”the real-time enterprise”, through which business analysis could be done in real time. Such tactical analysis requires streams of data to be captured from operational systems, combined and then acted on in real time. Supporting the real time enterprise is one goal of specialized stream processing engines such as Aurora and Medusa.

A second observation is that micro-sensor technology is declining in cost precipitously. Micro-sensors that reflect their state to a nearby active device are about the size of a United States dime, and are rapidly converging on a price less than $0.10. In fact, Gillette just placed an order for 500,000,000 ”dimes”, presumably to put on each package of razor blades. This and other sensor technology will allow most any object of value to report its state (including its geographic position) in real time. Such technology will enable a new class of application to collect and act on real-time streams of state information from objects. We will call these monitoring applications, and a major focus of stream processing engines is to support such applications.

In the remainder of this paper we discuss two stream processing prototypes, Aurora and Medusa. We begin in Section 2 with the Aurora engine, then turn in Section 3 to the Medusa network infrastructure. Section 4 continues with the two different approaches to distributed processing. The first, Aurora*, assumes a tight coupling between the various machines, while the second, Medusa, is appropriate when the various systems belong to different organizations. Also discussed in Section 4 is a co-operative study on high availability in stream processing systems. Section 5 then turns to the Linear Road benchmark and offers a brief rationale for this work. The paper concludes in Section 6 with the current state of the two prototypes.

2 The Aurora Stream Processing Engine

The Aurora stream processing engine has five features that distinguish it from other proposals in the same general space such as [5, 6, 10]. They are a workflow orientation, a novel collection of operators, a focus on efficient scheduling, a focus on maximizing quality of service, and a novel optimization structure. We discuss each of these aspects in turn.

2.1 Workflow Orientation

There are two main considerations that led us to build Aurora as a workflow system. First, most monitoring applications contain a component that performs either sensor fusion or data cleansing. Many sensors (especially mobile ones) are low power, and therefore noisy. Hence, signal processing must be performed on the resulting signal. In addition, it is often necessary to triangulate observations from two different sensors in order to define the position of an object. Such front-end signal processing can either be part of the stream processing system or contained in a separate subsystem. Aurora supports signal-processing through powerful programmable operators that allow users to specialize their behavior on either a tuple-by-tuple or an aggregate (window) basis.

The feeling of the Aurora designers is that both data storage, and real-time processing should be in the same system, in order to provide optimized system performance. In this way, the number of boundary crossings between the application layer and the storage/processing layer can be minimized. To facilitate this combination of function, one must adopt a more general processing model than just a query language.

The second reason has to do with query optimization. When an application designer wishes to add new capabilities to an Aurora workflow, he merely locates the correct place in the workflow diagram to add the needed functionality. In contrast, if he submitted one or more SQL commands to an Aurora system, then Aurora would have to do multiple query optimization in order to construct an optimized composite query plan. Common subexpression elimination is at the heart of multiple query optimization and is known to be a very hard problem [12]. The Aurora designers wished to avoid this particular ”snake pit”.

For these reasons, the Aurora application designer is presented with a ”boxes and arrows” GUI through which he can build his workflow. The boxes in the diagram represent primitive Aurora operators, and the arrows indicate data flow. A screen shot of the Aurora GUI is shown in Figure 1.

2.2 The Aurora Operators

The primitive Aurora operators (boxes) have gone through several iterations, and will doubtless go through a few more before they ultimately stabilize. The current collection is defined precisely in [1], and includes the standard filtering, mapping, windowed aggregate, and join boxes. These operators are found in many other stream-oriented systems. Aurora extends this collection with four aspects of novel functionality. First, windowed operations have a timeout capability. If a windowed operation is expecting to operate on three messages, then it must block until all three are received. It is possible for Aurora to block indefinitely in this situation, for example if the generator of the missing message has become disconnected from the Aurora system. In order to avoid this undesirable behavior, Aurora windowed operators can timeout, and produce an output message based on less than the required number of input messages.

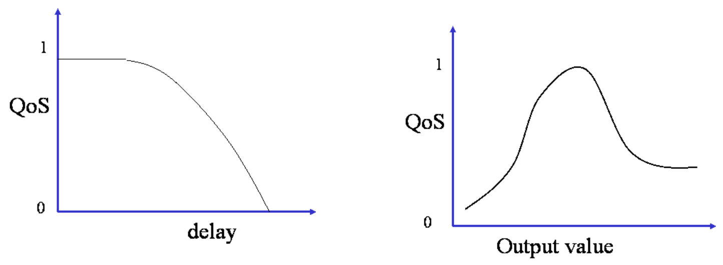
The second feature deals with out-of-order input messages. When there is significant network delay, messages may arrive in an order not intended by the application designer. Further, since Aurora allows windows on any attribute, monotonic attribute values cannot be guaranteed. If a windowed operator is expecting to process all the 9AM messages, then it must wait a little while after receiving a 9:01 message before ”closing” the 9AM window. A slack parameter on each windowed operator allows this sort of controlled waiting. A simpler (and less functional) mechanism is presented in [10] to deal with the same problem.

The third novel feature of Aurora operators is extendability. Aggregates, filters, and mapping operators can all be user-defined. Hence, Aurora is not restricted to a built-in collection of aggregates, as in most SQL systems. Instead, if an Aurora application designer wishes to have ”third largest” as an aggregate, then he is free to define it. Aurora closely follows the extendability features in POSTGRES [14] this regard.

Lastly, Aurora implements a novel resample operator. This operator allows interpolation between two values of a first stream, thereby constructing a message that matches the timestamp of a message in a second stream.

2.3 The Aurora Scheduler

A common misperception about scheduling algorithms is that the cost of running them does not need to be considered. On the contrary, our initial experimental results from the Aurora engine implementation is that the CPU cost of running the scheduler is comparable to or exceeds the cost of executing an Aurora operation on an incoming message [4]. Put differently, if the inner loop of control flow in a stream-based system is:



(a) Drop-based (b) Value-based

Figure 2: Two QoS Graphs

```
Run the scheduler to decide which message to process and then
Cause a single box to the execute that message
```

then the scheduling overheard may exceed the execution cost and an unreasonable (greater than 50%) degradation in performance will be observed. In contrast, the inner loop in Aurora can assemble large trains of messages and then move them collectively through a substantial number of boxes in the workflow diagram. Such train scheduling is the only way to keep the overhead of the scheduler from being onerous.

The objective of our train scheduling algorithms is to maximize quality of service, which is the topic of the next section. We have investigated a collection of heuristic algorithms, having given up long ago on finding algorithms that are provably optimal according to some metric. Our initial scheduling algorithms are presented in [3] and their experimental performance in [4]. We expect to continue our heuristic algorithm development in combination with ongoing experimental studies on our prototype. In particular, we must strive for lower overhead scheduling strategies that increase the percentage of useful work that an Aurora system can deliver.

2.4 Quality of Service

A fundamental tenet of Aurora is that Quality of Service (QoS) should be maximized in a stream processing system. Since Aurora is fundamentally a real-time engine, it is obvious that some applications should get better service than others. In military applications, for example, the task that tracks an incoming missile should get all possible resources and other less important tasks should face temporary resource starvation. Differential resource allocation has long been a tenet in real-time systems [11], but has not been utilized in current DBMSs.

The Aurora approach is for each application that receives output messages from the Aurora engine to specify a QoS graph. Two example graphs are shown in Figure 2. Figure 2(a) shows the utility that the application receives from varying response times. QoS graphs can also be based on metrics other than response time. For example, in a heart monitoring application, ”normal” output is much less important than messages that represent abnormal behavior. In this case, a value-based QoS graph is appropriate as noted on Figure 2(b).

The basic Aurora idea is to have human-specified QoS graphs at the outputs of an Aurora workflow, and then to infer approximate QoS curves for all interior workflow nodes by ”pushing” the QoS curves ”upstream” through the operator boxes. The details of this operation for value-based QoS are indicated in [15].

With a derived QoS curve on every arc of a workflow, two activities are possible. First, the scheduler can make intelligent decisions based on these curves, as noted in the previous section. In this way, unimportant portions of the workflow can be ”starved” to allow more important ones to get the majority of the resources.

Secondly, in certain Aurora applications, it is acceptable to discard messages in overload conditions. The rationale is that this behavior is desired in real-time systems. Also there may be lost messages in the sensor network when an object goes out of range. Hence, ACID properties may be impossible to achieve anyway. As such, Aurora can optionally be instructed to shed messages when overloaded. This pruning is accomplished by adding temporary ”drop” boxes. The purpose of a drop box is to filter messages, either based on value or randomly, in order to reduce load and provide better overall quality of service at the expense of accuracy.

One assumption that Aurora must make is that applications connected to different outputs in an Aurora workflow have a common notion of quality of service. As such, it is essential that they come from a single organization, in which context it is possible to have a common notion of QoS.

2.5 Aurora Optimization

An Aurora workflow is a dynamic object. When new applications get connected to the workflow, the number of boxes and arcs changes. Also ad-hoc queries that are run just once and then discarded have a similar effect. In addition, an Aurora workflow may become quite large; after all it is the stream processing activity of all applications that deal with a collection of sensor inputs.

Contrary to current DBMSs which perform optimization at compile-time by examining most of the possible ways to execute a given query, Aurora takes a radically different approach. First, it is implausible to exhaustively examine all (or most of) the possible permutations of the boxes in an Aurora workflow; the computational complexity is just too high in a large workflow. Second, every time the workflow is changed, this process would have to be repeated, leading to prohibitive overhead. As a result, Aurora does not examine all possible box rearrangements, and performs only run-time optimization.

An Aurora workflow has the notion of connection points. These are marked arcs in a workflow to which additional pieces of workflow can be connected. In addition, connection points also keep a user-specified amount of history, so that newly added Aurora boxes can examine historical messages if they so choose. Since new boxes can be added to connection points, they mark places of invariance in an Aurora workflow. The optimizer cannot reorder boxes by pushing a downstream box through a connection point. This reduces the complexity of the optimization process. In addition, not all Aurora operators commute as noted in [1]; again limiting the possible changes to an Aurora network.

While an Aurora workflow is executing, it is the job of the Aurora optimizer to examine small subnetworks for the possibility of a box rearrangement with better performance. If such a rearrangement is found, then messages are held on the upstream arcs while the subnetwork is ”drained”. Then the rearrangment can be put in place and processing resumed.

Notice that this results in an optimizer architecture that implements a ”greedy” algorithm. It has been clearly shown that such an approach may not produce a globally optimal plan, but we don’t see a way to do better on large workflows.

3 Medusa Network Infrastructure

Many stream processing applications run on multiple computers in a network. In contrast to several other projects which have focused on single machine environments, both Aurora and Medusa are actively pursuing distributed architectures. To facilitate both projects, a common networking infrastructure has been written by the Medusa group at M.I.T.

This infrastructure performs several functions, as discussed in detail in [7]. First, it supports a distributed naming scheme so that the output of an Aurora workflow at one site can be named and connected to a named input node on a second Aurora workflow at a second site. This allows local Aurora workflows to be assembled into larger distributed workflows. In addition, the networking layer multiplexes messages from different streams onto a much smaller number of TCP/IP connections. This multiplexing cuts down on the number of

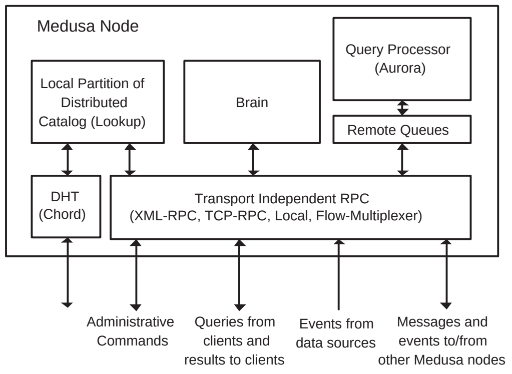

Medusa Node
Query Processor (Aurora)
Local Partition of Distributed Catalog (Lookup)
Brain
Remote Queues
DHT (Chord)
Transport Independent RPC (XML-RPC, TCP-RPC, Local, Flow-Multiplexer)
Administrative Commands
Queries from clients and results to clients
Events from data sources
Messages and events to/from other Medusa nodes

Figure 3: High-level architecture of a Medusa node. Components asynchronously process messages that arrive through their respective input queue(s). *Brain* processes all administrative commands such as creation of new schemas and streams. It receives ad-hoc queries which it partitions and distributes across other nodes. It also monitors and manages load at runtime by moving operators to and from other nodes. The local query processor runs an instance of Aurora. The remote queues component is a wrapper around the local query processor. It forwards events to and from other Medusa nodes and routes events coming from data sources to appropriate locations. *Lookup* holds one fraction of the catalog running over a distributed hash-table (DHT) such as Chord [13]. The transport independent RPC allows clients and Medusa nodes to seamlessly communicate with each other using various transport mechanisms. Within a node, components also communicate through that layer, but the communication is then a light-weight message exchange between threads.

physical connections between sites and improves efficiency. Similar connection pooling is widely performed by application servers and current commercial DBMSs. A report on the flow multiplexer is in preparation [9].

A second piece of infrastructure functionality is a collection of distributed catalogs that store the pieces of a distributed workflow and their interconnections. We are investigating using Chord [13] as underlying distributed hash-table to assign distinct catalog partitions to Medusa nodes.

A last piece of infrastructure functionality is transport. Basically once a distributed workflow is executing, this component efficiently delivers messages from an output arc on one machine to an input arc on a second machine. Figure 3 illustrates the architecture of a Medusa node and the components described above.

4 Distributed Stream Processing

There are two aspects of distributed workflow being investigated. Both groups are refining different processing models, and they are co-operating on a study of high-availability in a stream-based world. This section discussed both classes of efforts.

4.1 Distributed Workflow

Both groups are investigating different notions of distributed workflow. The Aurora project is pursuing Aurora*, which will connect multiple Aurora workflows together in a distributed environment. The basic idea is to push QoS graphs across site boundaries in a similar manner to the way they are pushed ”upstream” on a single host. This will allow a collection of local schedulers to collectively maximize QoS. In addition, the Aurora optimizer will be extended so that the process of quiescing, changing and restarting a subnetwork can cross a site boundary. This will allow cross-site optimization, as well as load balancing across sites. The specific approach is discussed in [7]. The Aurora* architecture is appropriate where all the hosts on which a distributed workflow is executing belong to the same organization. In such a single domain it is reasonable to assume that a common notion of QoS exists across machine boundaries.

In contrast the Medusa group is focused on environments where the hosts belong to different organizations and no common QoS notion is feasible. This multiple domain situation would be found when multiple organizations are providing services to an entity, as would be common in a Web services world. It is also typical in current cell phone environments where multiple providers co-operate to provide roaming.

The Medusa approach to distributed workflow is based on an agoric model. Each participant (host) is assumed to be an economic entity, that is interested in maximizing local profit. As such, it can enter into contracts to provide stream processing services for other participants. For example, one participant might have a geographic database of hotels and a second a similar database of restaurants. A third participant might collect sensor data from cars moving around a metropolitan area. The third participant could contract with each of the first two to provide a service to its mobile consumers to find on demand the hotels within a mile of the consumer with a first-class restaurant on the premises.

The Medusa approach requires a few assumptions in order to avoid economic chaos. For example, messages must have positive value, and applying an Aurora workflow to a message must increase its value. The Medusa group is then focused on refining its contracting model and on finding properties of the model that are enabled by certain additional assumptions. One specific focus is on finding what assumptions are necessary to guarantee that all participants avoid overload situations. A report on the agoric model and its properties is in preparation [2].

4.2 High Availability

Although some stream processing systems are non-transactional as noted above, there are others, especially in the realm of the real-time enterprise that require ACID properties. For example, a large financial institution is interested in exploring the use of a stream processing system to route trades for financial securities to the best ”desk” for execution. In this world, losing messages is unthinkable. For this reason, Medusa and Aurora are co-operating in a study of high availability (HA) in stream systems.

The gold standard of HA is the so-called process-pairs model originated by Tandem. Here, an application (process) is allotted a backup on a second site. Whenever the process reaches a checkpoint, it sends a message to its partner with state information. If the primary fails, then the partner resumes computation from the most recent checkpoint, and uninterrupted operation is assured in the presence of a single failure.

If one assumes a distributed workflow of Aurora operators, then one need not adopt a process-pair model. Instead each site can buffer messages for its downstream neighbor. Only when the message has assuredly exited from the downstream neighbor, can the buffer be truncated. Hence, a small collection of ”queue trimming” messages must flow upstream to sites buffering messages. If a site fails then the upstream neighbor can notice and restart the piece of the Aurora network on a backup site and send a copy of the queued messages to it. When the dead site resumes operation, the reverse switch can take place. This will achieve resilience from a single site failure with a minimum of messages. Also, HA operation is ”lazy”, i.e. messages are not actively sent to a backup site until a failure is observed.

As one can readily imagine, a stream-based HA scheme will have slower recovery time than a process-pair approach because of the lazy nature of the backup. However, run time overhead is noticeably lower. As software and hardware becomes more reliable, we believe that this tradeoff of longer recovery time for lower run-time overhead is worth exploring. As such, we have written a simulation model that can explore both kinds of schemes. We are now exploring the operation of the simulation, and a paper on our results is in preparation [8].

5 Linear Road Benchmark

Several researchers have doubted the need for specialized stream processing engines, speculating instead that commercial DBMSs will work just fine on stream applications. To resolve this issue, we have designed a benchmark in co-operation with other stream projects called Linear Road. It simulates collecting tolls on a collection of expressways, based on the amount of traffic that is using each segment of the expressway. It is driven by real-time reports of vehicle locations from assumed in-vehicle sensors. During the current semester we plan on using this benchmark to test performance of our prototypes against a popular commercial DBMS. We have invited other stream projects to participate, and Stanford has agreed to also run the benchmark. A report on this activity is planned for midyear [16].

6 The Prototypes

The Aurora engine consists of about 75,000 lines of C++ code and 30,000 lines of Java for the design-time user interface. It is operational at this time. We are adding the remainder of the planned features and working on improving performance in preparation for the Linear Road benchmarking activity. The Medusa infrastructure is also operational and its integration with the Aurora engine is underway. The Medusa economic model is fairly robust, and a primitive implementation exists. When integration with Aurora is complete, we will switch over to using the Aurora engine for local processing. We are also about to begin serious efforts on Aurora*.

References

[1] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, Submitted.

[2] Magdalena Balazinska, Can Emre Koksal, Hari Balakrishnan, and Michael Stonebraker. The Medusa economic model for load management and sharing. In preparation.

[3] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, August 2002.

[4] Don Carney, Uğur Çetintemel, Alexander Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. Technical Report CS-03-04, Department of Computer Science, Brown University, February 2003.

[5] Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, and Joseph Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[6] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[7] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[8] Jeong-Hyon Hwang, Hiro Iwashima, Alexander Rasin, Magdalena Balazinska, Hari Balakrishnan, Uğur Çetintemel, Mitch Cherniack, Michael Stonebraker, and Stan Zdonik. High-availability in stream-based processing systems. In preparation.

[9] Hiro Iwashima, Jon Salz, and Hari Balakrishnan. Flexible bandwidth sharing through application-level TCP flow multiplexing. In preparation.

[10] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, January 2003.

[11] Gultekin Ozsoyoglu and Richard T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 7(4):513–532, 1995.

[12] Timos K. Sellis and Subrata Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[13] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *Proc. of the ACM SIGCOMM Conference*, August 2001.

[14] Michael Stonebraker and Lawrence A. Rowe. The design of POSTGRES. In *Proc. of the 1986 ACM SIGMOD International Conference on Management of Data*, 1986.

[15] Nesime Tatbul, Uğur Çetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. Technical Report CS-03-03, Department of Computer Science, Brown University, February 2003.

[16] Richard Tibbetts and Michael Stonebraker. Linear Road benchmark: Performance evaluation of stream-processing engines. In preparation.

TelegraphCQ: An Architectural Status Report

Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande
Michael J. Franklin, Joseph M. Hellerstein, Wei Hong[†], Samuel R. Madden
Fred Reiss, Mehul A. Shah

University of California, Berkeley, CA 94720
[†]Intel Research Laboratory, Berkeley, CA 94704
http://telegraph.cs.berkeley.edu

Abstract

We are building TelegraphCQ, a system to process continuous queries over data streams. Although we had implemented some parts of this technology in earlier Java-based prototypes, our experiences were not positive. As a result, we decided to use PostgreSQL, an open source RDBMS as a starting point for our new implementation. In March 2003, we completed an alpha milestone of TelegraphCQ. In this paper, we report on the development status of our project, with a focus on architectural issues. Specifically, we describe our experiences extending a traditional DBMS towards managing data streams, and an overview of the current early-access release of the system.

1 Introduction

Streaming data is now a topic of intense interest in the data management research community. This has been driven by a new generation of computing infrastructure, such as sensor networks, that has emerged because of pervasive devices. At Berkeley, we are building TelegraphCQ, a system for continuous dataflow processing. TelegraphCQ aims at handling large streams of continuous queries over high-volume highly variable data streams. In this paper, we focus on the architectural aspects of TelegraphCQ in its current release; please see [3] for the details of the novel query execution techniques that underlie the system.

As part of our earlier work [2, 6, 7, 4] in the Telegraph project, we built a system for adaptive dataflow processing in Java [1]. Although our research on adaptivity and sharing showed significant benefits, our experience in using Java in a systems development project was not positive [10]. After considering a few alternatives, we decided to use the PostgreSQL open source database system as a starting point for our new implementation. A continuous query system like TelegraphCQ is quite different from a traditional query processor. We found, however, that a significant amount of PostgreSQL code was easily reusable. We also found the extensibility features of PostgreSQL very useful, particularly the ability to load user-defined code and the rich data types such as intervals.

Challenges: As discussed in [3], sharing and adaptivity are our main techniques in implementing a continuous query system. Doing this in the codebase of a conventional database posed a number of challenges:

- **Adaptivity:** While the TelegraphCQ query executor uses a lot of existing PostgreSQL functionality, the actual interaction between query processor operators is significantly different. We rely on our prior work on adaptive query processing with operators such as *eddies* for tuple routing and operator scheduling, as well as *fjords* for inter-operator communication.

- **Shared continuous queries:** TelegraphCQ aims at amortizing query-processing costs by *sharing* the execution [6] of multiple long-running queries. This requirement resulted in a large-scale change to the conventional process-per-connection model of PostgreSQL.

- **Data ingress operations:** Traditional federated database systems fetch data from remote data sources using an operator in a query plan. Typically, such an operator uses user-defined (often "in the factory") *wrapper* functions that fetch data across the network. For a shared query processing system however, it is vital that all operators are non-blocking, and it is not possible to guarantee this with traditional wrappers.

Status: At the time of writing, we have just completed an alpha milestone – an early access release of TelegraphCQ. In its current state, TelegraphCQ supports a DDL statement, `CREATE STREAM`, that can be used to create archived or unarchived streams. Sources can register themselves with TelegraphCQ and push data to specific streams. This data is used to produce tuples using stream- and source- specific *wrapper* functions registered in the system. In addition, continuous queries can be registered that work over these streams. The queries can involve individual streams, joins over multiple streams and joins over streams and tables. The DML for these queries includes primitive sliding window syntax that supports grouped aggregate operations over these sliding windows. Please visit `http://telegraph.cs.berkeley.edu` for more information.

The rest of this paper is organized as follows. We start with a description of how a user interacts with TelegraphCQ in Section 2. In Section 3 we present an overview of TelegraphCQ. Next, in Section 4 we show how user-defined wrappers can be created in the system to interface with external data sources. We conclude with remarks on future work in Section 5.

2 Using TelegraphCQ

In this section we describe how users and applications can interact with TelegraphCQ. In addition to the full features of PostgreSQL, interactions with TelegraphCQ involve the following:

- Creating archived and unarchived streams.
- Creating sources that stream data to TelegraphCQ.
- Creating user-defined wrappers.
- Issuing continuous queries.

The streams that are created in the system identify objects that may be queried and data that may be processed, and possibly archived. The data sources are independent programs that continually send data to a specific named stream in TelegraphCQ. For each stream, there is a user-defined wrapper that is registered with TelegraphCQ and is designed to understand the data sent by the source. Finally, users and applications connect to TelegraphCQ so that they can issue continuous queries over these streams.

Creating streams: In TelegraphCQ, streams can be created and dropped using the new DDL statements, `CREATE STREAM` and `DROP STREAM` respectively. These statements are similar to those used to create and drop tables. A requirement in defining a stream is that there *must* exist exactly one column of a time type which serves as a timestamp for the stream. This column is specified with a new `TIMESTAMPCOLUMN` constraint. For example, an archived stream of readings of average speeds measured by traffic sensor stations in freeways may be defined as:

```
CREATE STREAM measurements (tcqtime TIMESTAMP TIMESTAMPCOLUMN,
                            stationid INTEGER,
                            speed REAL) TYPE ARCHIVED
```

Creating sources: TelegraphCQ expects that a data source will initiate a network connection with it and advertise the name of the stream for which it undertakes to provide data. More than one source can simultaneously push data to the same stream. The stream name is identified in the first few bytes sent to TelegraphCQ after establishing the network connection. There are no other restrictions on the format of the stream data, as all subsequent network operations on the connection are the responsibility of the user-defined wrapper functions associated with the stream.

Creating wrappers: Each wrapper consists of three user-defined functions (`init`, `next` and `done`) that are called by TelegraphCQ to process data from external sources. This is described in more detail in Section 4. The functions are registered in TelegraphCQ using the standard PostgreSQL `CREATE FUNCTION` statement. For example the `init` function of the `measurements` wrapper should be named `measurements_init` and can be declared using the following DDL.

```
CREATE FUNCTION measurements_init(INTEGER) RETURNS BOOLEAN
AS 'libmeasurements.so',measurements_init' LANGUAGE 'C';
```

Continuous queries: Once streams have been created in the system, users can issue long-running continuous queries over them. When data streams in, the results get sent to the appropriate issuers. A query is identified as being continuous if it operates over one or more streams. Such queries do not end until cancelled by the user. The queries can be simple SQL queries (without subqueries) with an optional *window* clause. The interval that is specified as a string in the window clause may be anything that PostgreSQL can convert automatically into an *interval* type. The window clause serves to restrict the amount of data that participates in the query over time. This is particularly important for join and aggregate queries since streams are unbounded. An example query that involves a stream and a table is:

```
SELECT   ms.stationid, s.name, s.highway, s.mile, AVG(ms.speed)
FROM     measurements ms, stations s
WHERE    ms.stationid = s.stationid
GROUP BY ms.stationid, s.name, s.highway, s.mile
WINDOW   ms ['10 minutes']
```

This query joins `measurements`, a stream of sensor readings with `stations`, a normal relation. The query continually reports average speeds recorded by each station in a sliding 10 minute window.

These queries can be submitted to TelegraphCQ using `psql`, the interactive client. Other programmatic interfaces such ODBC and JDBC can also be used, but the applications using those must submit continuous queries by declaring named cursors. Otherwise the PostgreSQL call-level interface buffers the results of a query and blocks until all results have been received.

3 Overview of the system

In this section we present a short overview of the TelegraphCQ system, and show how we leverage the functionality of PostgreSQL.

Figure 1 shows the basic process structure of PostgreSQL. PostgreSQL uses a process-per-connection model. Data structures shared by multiple processes, such as the buffer pool, latches, etc. are located in shared memory. A *Postmaster* process forks new server processes in response to new client connections. Amongst the different components in each server process, the *Listener* is responsible for accepting requests on a connection and returning data to the client. When a new query arrives it is parsed, optimized, and compiled into an access plan that is

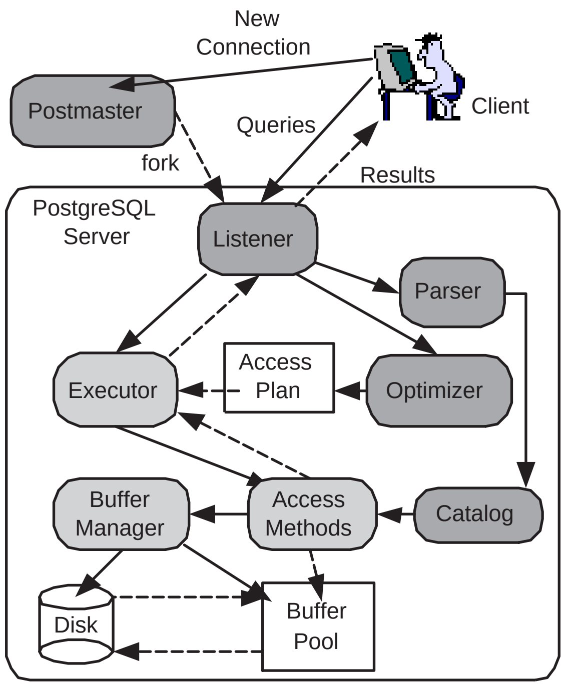


Figure 1: PostgreSQL Architecture

then processed by the query *Executor* component. The components that have only been changed minimally in TelegraphCQ are shaded in dark gray in Figure 1. These include: Postmaster, Listener, System Catalog, Query Parser and Optimizer[1]. Components shown in light gray (the Executor, Buffer Manager and Access Methods) are being leveraged with significant changes. In addition, by adopting the front-end components of PostgreSQL we also get access to important client-side call-level interface implementations (not shown in the figure) such as ODBC and JDBC.

Our chief challenge in using PostgreSQL is supporting the TelegraphCQ features it was not designed for: streaming data, continuous queries, shared processing and adaptivity. The biggest impact of our changes is to PostgreSQL's process-per-connection model. In TelegraphCQ we have a dedicated process, the TelegraphCQ Back End (BE), for executing shared long-running continuous queries. This is in addition to the per-connection TelegraphCQ Front End (FE) process that fields queries from, and returns results to, the client. The FE process also serves non-continuous queries and DDL statements. TelegraphCQ also has a dedicated Wrapper Clearing-House process that ensures that data ingress operations do not impede the progress of query execution.

Figure 2 depicts a bird's eye view of TelegraphCQ. The figure shows (as ovals) the three processes that comprise the TelegraphCQ server. These processes communicate using a shared memory infrastructure. The TelegraphCQ Front End contains the Listener, Catalog, Parser, Planner and "mini-Executor". The actual query processing takes place in a separate process called the TelegraphCQ Back End. Finally the TelegraphCQ Wrapper ClearingHouse is used to host the data ingress operators which make fresh tuples available for processing, archiving them if required.

As in PostgreSQL, the Postmaster listens on a well-known port and forks a Front End (FE) process for each fresh connection it receives. The listener accepts commands from a client and based on the command, chooses where to execute it. DDL statements and queries over tables are executed in the FE process itself. Continuous

[1]Currently we use the PostgreSQL optimizer to create the plan data structures; we are planning to bypass this step for continuous queries in future.

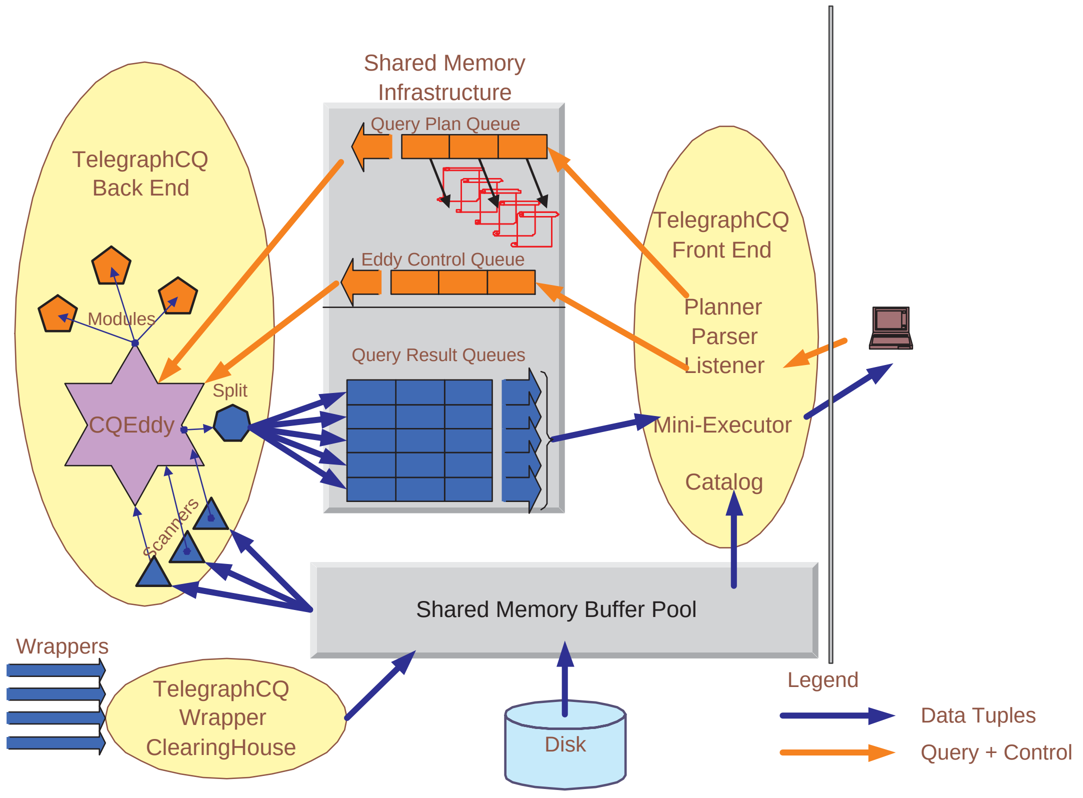


Figure 2: TelegraphCQ Architecture

queries, those that involve streams as well as streams and tables, are “pre-planned” and sent via the Query Plan queue (in shared memory) to the Back End (BE) process. The BE executor continually dequeues fresh queries and *dynamically* folds them into the current running query. Query results are in turn, placed in the client-specific Query Result queues. Once the FE has handed a query off to the BE, it produces an FE-local minimal query plan that the mini-executor runs to continually dequeue results from its Query Result queue and send back to the connected client.

Since a continuous query never ends, clients should submit such queries as part of named cursors. We have added a continuous query mode to `psql`, the standard PostgreSQL interactive client. In this mode, `SELECT` statements are automatically converted into named cursors which can then be iterated over to continually fetch results.

4 Wrappers

In this section we describe how data from external sources can be streamed into TelegraphCQ. We plan to support the following kinds of sources:

- Pull sources, as found in “traditional” federated database systems.
- Push sources, where connections can be initiated either by TelegraphCQ (TelegraphCQ-Push) or by the data source itself (Source-Push).

With pull and TelegraphCQ-push sources, TelegraphCQ connects to the source. In contrast, Source-Push sources connect to a well-known port of TelegraphCQ, advertise the stream they are “supplying” and commence data delivery. In our current release we only support Source-Push sources.

As with traditional federated systems, we use user-defined wrapper functions to massage data into a TelegraphCQ format (essentially that of PostgreSQL). In our current implementation, there is a one to one correspondence between a wrapper and a stream, although eventually we will ease this restriction. In TelegraphCQ, however, the system handles the network interface, and the user-defined wrapper limits itself to reading data off a network socket.

Since it is expensive to create a fresh process for each data source, we have a single process, the TelegraphCQ Wrapper ClearingHouse (WCH) that manages data acquisition. The WCH is responsible for the following operations:

- Accepting connections from external sources, and obtaining the stream name they advertise.
- Loading the appropriate user defined wrapper functions.
- Calling these wrapper function when data is available on a connection.
- Processing result tuples returned by a wrapper according to the stream type.
- Cleaning up when a source ends its interaction with TelegraphCQ.

The WCH accepts connections from new data sources, and based on the stream name the source advertises, loads and executes the appropriate user-defined wrapper code. Since the single WCH process handles multiple sources, each network socket, and hence each wrapper function, must be non-blocking.

The wrapper code is called either in response to a data-ready indication on the wrapper's network socket, or if there is any data as yet unprocessed by the wrapper. The latter condition can arise, as the wrapper only returns one tuple at a time, in a manner akin to the classical iterator [5] model. Each call to a wrapper may not even result in a fresh tuple, as the data required to form a tuple might not be available all at once (e.g. a large XML document that streams in across many network packets). So user defined wrappers should read data from the network, buffering it if necessary.

Once the wrapper has enough data to form a tuple, it converts it into the PostgreSQL data types expected by the target stream. Such conversion often relies on calls to PostgreSQL's own data conversion and construction functions, and so the wrapper writers are expected to have at least minimal familiarity with PostgreSQL datatypes and data manipulation functions. In particular, the wrapper must return data to the system as an array of PostgreSQL `Datum` items that can be used to create a data tuple in the PostgreSQL format. In future the WCH will be extended to provide both mapping and data conversion services to wrappers to make their implementation easier, perhaps based on the Potter's Wheel [8] system.

A wrapper consists of three separate user defined functions: `init` for initialization, `next` to produce new records, and `done` for cleanup. They are loaded from a shared library using the PostgreSQL function manager. Each of these functions take a single argument that is treated as a pointer to a `WrapperState` structure, and return a boolean value to indicate success or fatal error. The `next` function returns processed tuples through a field in the `WrapperState`. Most wrapper functions will need to maintain their own additional state. This is done through another private field of the `WrapperState`. Wrapper functions will also typically allocate and free memory respectively in their `init` and `done` functions using the PostgreSQL region-based memory management infrastructure.

The WCH and wrappers use other fields to communicate state information and return values. For instance the wrapper needs to indicate to the WCH that it has as yet unprocessed data and must be called even if its network connection remains idle.

The final responsibility of the WCH is to make freshly created tuples available to the rest of the system. This is done by placing the tuples in buffers within the standard PostgreSQL buffer pool (analogous to the way buffers would be allocated for insertions to a traditional table). In the case of archived streams these are flushed to disk as part of the normal buffer manager operation. Subsequently, normal scan operators in the TelegraphCQ executor fetch these archived tuples and process them. Unarchived streams can be handled in a similar way, the difference being that dirty buffers are not written out to stable storage. In the current release, however, we use a separate shared memory queue for unarchived streams.

5 Conclusion and future work

This paper describes the current status of TelegraphCQ, a streaming data management system that processes continuous queries. This corresponds to an early access release of TelegraphCQ that we completed in March 2003. After our experience building the first version of Telegraph in Java, we chose to extend PostgreSQL, a proven open source database system that is implemented in the C programming language and runs on a wide variety of platforms. Starting off with PostgreSQL has helped us quickly ramp up the development of TelegraphCQ. We have benefited from very many useful features that already exist in PostgreSQL. The main message we have at this point in the TelegraphCQ project is that it is feasible to build a highly adaptive streaming dataflow system with shared query processing that uses building blocks from a conventional relational data base. An important next step is to analyze the performance of our system.

TelegraphCQ is very much a work in progress, and we are actively working on adding more features to the system. As part of this effort, we are addressing a number of open questions that have arisen. Some of these are:

- **Adaptive adaptivity:** Adaptivity clearly has a benefit in the hostile environments that streaming dataflow systems must operate in. These benefits, however, probably come with a concomitant cost. What are these costs, and is it possible for our system to adapt the *amount* of adaptivity dynamically ?

- **Storage system:** In a streaming system, data continually floods in, in a fashion unrelated to the queries in the system and how they are being processed. When data is no longer *orchestrated* through the system, can we revisit the role of the storage manager ?

- **Interactions between continuous and historical queries:** In TelegraphCQ we currently only support continuous queries over data streams. In future we plan to support historical queries by running them separately, say in the TCQ Front End process. However, it is not clear how to process queries that are combinations of a stream's newly arriving data and historical data (e.g. combine the last hour's traffic sensor data with the average data for this hour over the last year).

- **Scalability and Availability:** The Flux operator [9] is designed to enhance TelegraphCQ with partitioned parallelism, adaptive load-balancing, high availability and fault tolerance. The load-balancing work (Lux) has been validated in simulations from the prior version of Telegraph; the full-featured Flux implementation in TelegraphCQ is underway.

- **Signal/noise streams:** To handle overload situations, a number of research groups are considering schemes that downsample (drop) or synopsize (lossily compress) items in a stream. In this vein, we are exploring a new algebra that combines both the fully-propagated items and the synopsized components. Our intent is for this combined approach to provide a characterization of both the signal and noise produced when downsampling or synopsizing the stream. We are also exploring opportunities for joint optimization of the fully-propagated query plan and the synopsis plan.

6 Acknowledgements

We are grateful for comments and suggestions from other members of the database systems research group at Berkeley, as well as researchers who are building similar systems in other groups. This work has been supported by National Science Foundation awards IIS-0208588, IIS-0205647, IIS-0086057 and SI0122599 as well as funding from Microsoft, IBM, Intel and the UC MICRO program.

References

[1] ARNOLD, K., GOSLING, J., AND HOLMES, D. *The Java Programming Language*, 3rd ed. Addison-Wesley, 2000.

[2] AVNUR, R., AND HELLERSTEIN, J. M. Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD Conference* (2000), pp. 261–272.

[3] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M. J., HELLERSTEIN, J. M., HONG, W., KRISHNAMURTHY, S., MADDEN, S. R., RAMAN, V., REISS, F., AND SHAH, M. A. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of Conference on Innovative Data Systems Research* (2003).

[4] CHANDRASEKARAN, S., AND FRANKLIN, M. J. Streaming queries over streaming data. In *Proceedings of VLDB Conference* (2002).

[5] GRAEFE, G. Query evaluation techniques for large databases. *ACM Computing Surveys 25*, 2 (June 1993), 73–170.

[6] MADDEN, S. R., SHAH, M. A., HELLERSTEIN, J. M., AND RAMAN, V. Continuously adaptive continuous queries over streams. In *Proceedings of ACM SIGMOD Conference* (2002).

[7] RAMAN, V., DESHPANDE, A., AND HELLERSTEIN, J. M. Using state modules for adaptive query processing. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[8] RAMAN, V., AND HELLERSTEIN, J. M. Potter's wheel: An interactive data cleaning system. In *Proceedings of VLDB Conference* (2001), pp. 381–390.

[9] SHAH, M. A., HELLERSTEIN, J. M., CHANDRASEKARAN, S., AND FRANKLIN, M. J. Flux: An adaptive partitioning operator for continuous query systems. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[10] SHAH, M. A., MADDEN, S. R., FRANKLIN, M. J., AND HELLERSTEIN, J. M. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record 30*, 4 (December 2001), 103–114.

STREAM: The Stanford Stream Data Manager

The STREAM Group*

Stanford University
`http://www-db.stanford.edu/stream`

Abstract

The STREAM *project at Stanford is developing a general-purpose system for processing continuous queries over multiple continuous data streams and stored relations. It is designed to handle high-volume and bursty data streams with large numbers of complex continuous queries. We describe the status of the system as of early 2003 and outline our ongoing research directions.*

1 Introduction

The *STanford stREam datA Manager (STREAM)* project at Stanford is developing a general-purpose *Data Stream Management System (DSMS)* for processing continuous queries over multiple continuous data streams and stored relations. The following two fundamental differences between a DSMS and a traditional DBMS have motivated us to design and build a DSMS from scratch:

1. A DSMS must handle multiple continuous, high-volume, and possibly time-varying *data streams* in additional to managing traditional stored relations.

2. Due to the continuous nature of data streams, a DSMS needs to support long-running *continuous queries*, producing answers in a continuous and timely fashion.

A high-level view of STREAM is shown in Figure 1. On the left are the incoming *Input Streams*, which produce data indefinitely and drive query processing. Processing of continuous queries typically requires intermediate state, which we denote as *Scratch Store* in the figure. This state could be stored and accessed in memory or on disk. Although we are concerned primarily with the online processing of continuous queries, in many applications stream data also may be copied to an *Archive*, for preservation and possible offline processing of expensive analysis or mining queries. Across the top of the figure we see that users or applications register *Continuous Queries*, which remain active in the system until they are explicitly deregistered. Results of continuous queries are generally transmitted as output data streams, but they could also be relational results that are updated over time (similar to materialized views).

*Active members as of March 2003: A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom. This work was supported by NSF Grants IIS-0118173 and IIS-9817799, by Stanford Graduate Fellowships from 3Com, Rambus, and Sequoia Capital, by a Siebel Scholarship, and by support from the Okawa Foundation, Microsoft, Stanford Networking Research Center, and Veritas. Nishizawa is visiting Stanford from Hitachi, Ltd.

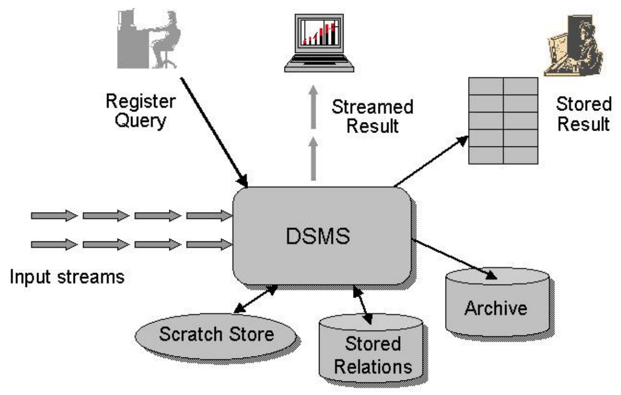


Figure 1: Overview of STREAM

Currently STREAM offers a Web system interface through direct HTTP, and we are planning to expose the system as a Web service through SOAP. Thus, remote applications can be written in any language and on any platform. Applications can register queries and receive the results of a query as a streaming HTTP response in XML. To allow interactive use of the system, we have developed a Web-based GUI as an alternative way to register queries and view results, and we provide an interactive interface for visualizing and modifying system behavior (see Section 4).

In Sections 2 (Query Language and Processing), 3 (Operator Scheduling), and 4 (User Interface) we describe the most important components of STREAM. In Section 5 we outline our current research directions. Due to space limitations this paper does not include a section dedicated to related work. We refer the reader to our recent survey paper [BBD$^+$02], which provides extensive coverage of related work.

2 Query Language and Processing

We first describe the query language and semantics for continuous queries supported by STREAM. The latter half of this section describes STREAM's query processing architecture.

2.1 Query Language and Semantics

We have designed an abstract semantics and a concrete declarative query language for continuous queries over data streams and relations. We model a *stream* as an unbounded, append-only bag of ⟨*tuple*, *timestamp*⟩ pairs, and a *relation* as a time-varying bag of tuples supporting updates and deletions as well as insertions. Our semantics for continuous queries over streams and relations leverages well-understood relational semantics. Streams are converted into relations using special *windowing* operators; transformations on relations are performed using standard relational operators; then the transformed relational data is (optionally) converted back into a streamed answer. This semantics relies on three abstract building blocks:

1. A relational query language, which we can view abstractly as a set of relation-to-relation operators.

2. A *window specification language* used to extract tuples from streams, which we can view as a set of stream-to-relation operators. In theory these operators need not have anything to do with "windows," but

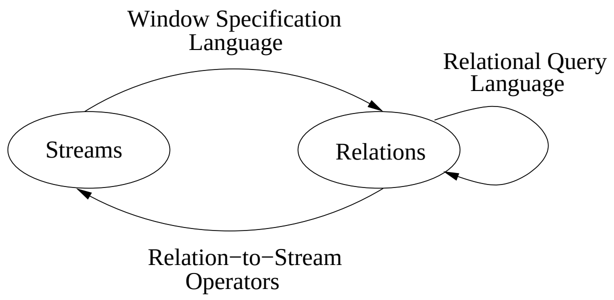

Window Specification Language
Relational Query Language
Streams
Relations
Relation−to−Stream Operators

Figure 2: Mappings used in abstract semantics

in practice windowing is the most common way of producing bounded sets of tuples from unbounded streams [BBD$^+$02].

3. A set of relation-to-stream operators.

The interaction among these three building blocks is depicted in Figure 2.

We have developed a concrete declarative query language, *CQL* (for *Continuous Query Language*), which instantiates our abstract semantics. Our language uses SQL as its relational query language, its window specification language is derived from SQL-99, and it includes three relation-to-stream operators. The CQL language also supports syntactic shortcuts and defaults for convenient and intuitive query formulation. The complete specification of our query semantics and CQL is provided in an earlier paper [ABW02]. The interested reader is referred to our *Stream Query Repository* [SQR], which contains queries from many realistic stream applications, including a large number and variety of queries expressed in CQL. A significant fraction of CQL has been implemented to date, as described in the next section.

2.2 Query Processing

When a continuous query specified in CQL is registered with STREAM, it is compiled into a *query plan*. The query plan is merged with existing query plans whenever possible, in order to share computation and memory. Alternatively, the structure of query plans can be specified explicitly using XML. A query plan in our system runs continuously and is composed of three different types of components:

1. Query *operators* correspond to the three types of operators in our abstract semantics (Section 2.1). Each operator reads tuples from a set of input queues, processes the tuples based on its semantics, and writes its output tuples into an output queue.

2. Inter-operator *queues* are used to buffer the output of one operator that is passed as input to one or more other operators. Incoming stream tuples and relation updates are placed in *input queues* feeding leaf operators.

3. *Synopses* maintain run-time state associated with operators.

STREAM supports the standard relational operators (including aggregation and duplicate elimination), *window* operators that compute time-based, tuple-based, and partitioned windows over streams [ABW02], three operators that convert relations into streams, and *sampling* operators for approximate query answering. Note that the queues and synopses for the active query plans in the system comprise the *Scratch Store* depicted in Figure 1.

A synopsis stores intermediate state at some operator in a running query plan, as needed for future evaluation of that operator. For example, a *sliding-window join operator* [KNV03] must have access to all the tuples that are

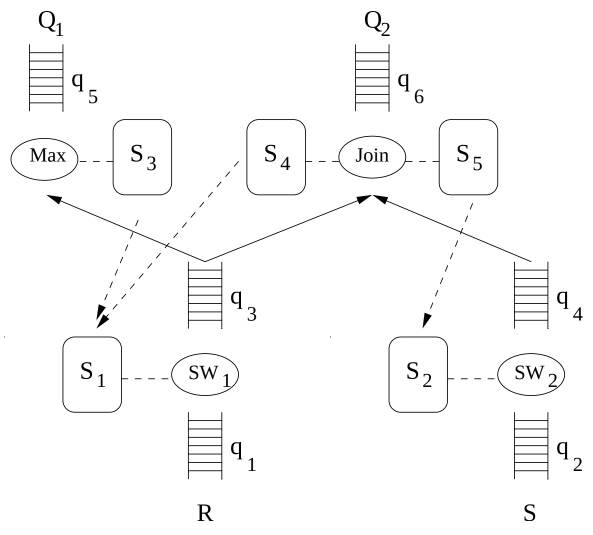


Figure 3: STREAM query plans

part of the current window on each of its input streams, so we maintain one *sliding-window synopsis* (typically a hash table) for each of these streams. On the other hand, simple filter operators, such as selection and duplicate-preserving projection, do not require a synopsis since they do not need to maintain state. The most common use of a synopsis in our system is to materialize a relation or a view (e.g., a sliding window). Synopses can also be used to store a summary of the tuples in a stream or a relation for approximate query answering. For this reason we have implemented *reservoir samples* [Vit85] over streams, and we will soon add *Bloom filters* [MW+03].

Figure 3 illustrates plans for two queries, Q_1 and Q_2, over input streams R and S. Query Q_1 is a windowed-aggregate query: it maintains the maximum value of attribute $R.A$ over a sliding window on stream R. Query Q_2 is a sliding-window join query over streams R and S. Together the plans contain four operators `SW`$_1$, `SW`$_2$, `Max`, and `Join`, five synopses S_1–S_5, and six queues q_1–q_6. `SW`$_1$ is a sliding-window operator that reads stream R's tuples from queue q_1, updates the sliding-window synopsis S_1, and outputs the inserts and deletes to this sliding window into queue q_3. Thus, queue q_1 represents stream R, while queue q_3 represents the relation that is the sliding-window on stream R. Similarly, `SW`$_2$ processes stream S's tuples from q_2, updating synopsis S_2 and queue q_4. The `Max` operator maintains the maximum value of $R.A$ incrementally over the window on R, using the inserts and deletes from the window maintained by `SW`$_1$. Whenever the current maximum value expires from the window, `Max` will potentially need to access the entire window to compute the new maximum value. Thus, `Max` must materialize this window in its synopsis S_3. However, since S_3 is simply a time-shifted version of S_1, we can share the data store between S_1 and S_3, as indicated by the dotted arrow from S_3 to S_1. Similarly, the sliding-window synopsis S_4 maintained by the join operator `Join` can be shared with S_1 and S_3, and S_5 can be shared with S_2. Also, queue q_3 is shared by `Max` and `Join`, effectively sharing the window-computation subplan between queries Q_1 and Q_2.

The *Aurora* system [CCC+02] supports shared queues, used to share storage for sliding windows on streams. Our system goes a step further in synopsis-sharing, including the ability to share the storage and maintenance overhead for indexes over the synopses as well. For example, if `Max` in Figure 3 computed `Group By` $R.B$, `Max` $R.A$, and `Join` used the join predicate $R.B = S.B$, then it would be useful to maintain a hash-index over $R.B$ in synopsis S_1 which both `Max` and `Join` could use. We currently support shared windows over streams where all the window specifications need not be the same, and shared materialized views, which are effectively common subexpressions in our query plans [CDTW00]. We use novel techniques to eliminate from synopses tuples that will not be accessed in the future, for example using reasoning based on constraints on the input streams [BW02].

Execution of query operators is controlled by a global *scheduler*, discussed next in Section 3. The operators have been implemented in such a way that they make no assumptions about the scheduling policy, giving the scheduler complete flexibility to adapt its scheduling strategy to the query workload and input stream characteristics.

3 Operator Scheduling

The execution of query plans is controlled by a global scheduler running in the same thread as all the operators in the system. (The I/O operations are handled by a separate thread.) Each time the scheduler is invoked, it selects an operator to execute and calls a specific procedure defined for that operator, passing as a parameter the maximum amount of time that the operator should run before returning control to the scheduler. The operator may return earlier if its input queues become empty.

The goals of a scheduler in a continuous query system are somewhat different than in a traditional DBMS. Some traditional scheduling objectives, such as minimizing run-time resource consumption and maximizing throughput, are applicable in the context of continuous queries, whereas other objectives, such as minimizing query response time, are not directly relevant in a continuous query setting, though they may have relevant counterparts (e.g., minimizing average latency of results). One objective that takes on unusual importance when processing data streams is careful management of run-time resources such as memory. Memory management is a particular challenge when processing streams because many real data streams are irregular in their rate of arrival, exhibiting burstiness and variation of data arrival rate over time. This phenomenon has been observed in networking [FP95], web-page access patterns, e-mail messages [Kle02], and elsewhere. When processing high-volume and bursty data streams, temporary bursts of data arrival are usually buffered, and this backlog of tuples is processed during periods of light load. However, it is important for the stream system to minimize the memory required for backlog buffering. Otherwise, total memory usage can exceed the available physical memory during periods of heavy load, causing the system to page to disk and limiting system throughput. To address this problem, we have developed an operator scheduling strategy that minimizes the memory requirement for backlog buffering [BBDM03]. This strategy, called *Chain scheduling*, is near-optimal in minimizing run-time memory usage for single-stream queries involving selections, projections, and foreign-key joins with stored relations. Chain scheduling also performs well for queries with sliding-window joins over multiple streams, and multiple queries of the above types.

The basic idea in Chain scheduling is to break up query plans into disjoint chains of consecutive operators based on their effectiveness in reducing run-time memory usage, favoring operators that “consume” a large number of tuples per time unit and “produce” few output tuples. This metric also determines the scheduling priority of each operator chain. Chain scheduling decisions are made by picking the operator chain with highest priority among those that have operators that are ready to execute and scheduling the first ready operator in that chain. Complete details of Chain, proofs of its near-optimality, and experimental results demonstrating the benefits of Chain with respect to other scheduling strategies, are provided in an earlier paper [BBDM03].

While Chain achieves run-time memory minimization, it may suffer from starvation and poor response times during bursts. As ongoing work, we are considering how to adapt our strategy to take into account these additional objectives.

4 User Interface

We are developing a comprehensive interactive interface for STREAM users, system administrators, and system developers to visualize and modify query plans as well as query-specific and system-wide resource allocation while the system is in operation.

4.1 Query Plan Execution

STREAM will provide a graphical interface to visualize the execution of any registered continuous query. Query plans are implemented as networks of *entities*, each of which is an operator, a queue, or a synopsis. The query plan execution visualizer will provide the following features.

1. View the structure of the plan and its component entities.
2. View specific attributes of an entity, e.g., the amount of memory being used by a synopsis in the plan.
3. View data moving through the plan, e.g., tuples entering and leaving inter-operator queues, and synopsis contents growing and shrinking as operators execute. Depending on the scope of activity individual tuples or tuple counts can be visualized.

4.2 Global System Behavior

'The query execution visualizer described in the previous section is useful for visualizing the execution and resource utilization of a single query, or a small number of queries that may share plan components. However, a system administrator or developer might want to obtain a more global picture of DSMS behavior. The STREAM system will provide an interface to visualize system-wide query execution and resource utilization information. The supported features include:

1. View the entire set of query plans in the system, with the level of detail dependent on the number and size of plans.
2. View the fraction of memory used by each query in the system, or in more detail by each queue and each synopsis.
3. View the fraction of processor time consumed by each query in the system.

4.3 Controlling System Behavior

Visualizing query-specific and system-wide execution and resource allocation information is important for system administrators and developers to understand and tune the performance of a DSMS running long-lived continuous queries. A sophisticated DSMS should adapt automatically to changing stream characteristics and changing query load, but it is still useful for "power users" and certainly useful for system developers to have the capability to control certain aspects of system behavior. STREAM does or will support the following features:

1. Run-time modification of memory allocation, e.g., increasing the memory allocated to one synopsis while decreasing memory for another.
2. Run-time modification of plan structure, e.g., changing the order of synopsis joins in a query over multiple streams, or changing the type of synopsis used by a join operator.
3. Run-time modification of the scheduling policy, choosing among several alternative policies.

5 Directions of Ongoing Research

This section outlines the problems that we are addressing currently in the STREAM project, in addition to implementing the basic prototype as described above. These problems fall broadly into the areas of efficient query processing algorithms, cost-based optimization and resource allocation, operator scheduling, graceful degradation under overload, and distributed stream processing.

Efficient query processing: Our system needs efficient query processing algorithms to handle high-volume data streams and large numbers of complex continuous queries. Some of the issues we are addressing in this area include techniques for sharing computation and memory aggressively among query plans, algorithms for multi-way sliding-window joins over streams, tradeoffs between incremental computation and recomputation for different types of continuous queries, and strategies for processing continuous queries efficiently while ensuring correctness in the absence of time-synchronization among stream sources and the DSMS.

Cost-based optimization and resource allocation: Although we have implemented support for a significant fraction of CQL in STREAM to date, our query plan generator is fairly naive and uses hard-coded heuristics to generate query plans. We are now moving towards one-time and dynamic cost-based query optimization of CQL queries. Since CQL uses SQL as its relational query language, we can leverage most of the one-time optimization techniques used in traditional relational DBMSs. Our unique optimization techniques include relocating window operators in query plans, exploiting stream constraints to reduce window sizes without affecting result correctness, and identifying opportunities for sharing computation (e.g., common subexpression computation, index maintenance) and memory (synopses and queues). Apart from choosing plans shapes and operators, a query optimizer must allocate resources such as memory within and across queries. One of the problems we are addressing in this area is how to allocate resources to query plans so as to maximize result precision whenever resource limitations force approximate query results. We are also exploring dynamic and adaptive approaches to query processing and resource allocation. Our adaptive query processing is less fine-grained than *Eddies* (as used in the *Telegraph* project [CC+03]). Our approach relies on two interacting components: a *monitor* that captures properties of streams and system behavior, and an *optimizer* that can reconfigure query plans and resource allocation as properties change over time.

Scheduling: As described in Section 3, the Chain scheduling strategy achieves run-time memory minimization, but it may suffer from poor response times during bursts. As ongoing work, we are adapting Chain to minimize total run-time memory usage for queries under the constraint that the latency of any query-result tuple must not exceed a given threshold. Another planned extension needed for a complete scheduling strategy for a DSMS is the intelligent handling of query workloads where synopses and queues do not all fit into the physical memory available in the DSMS.

Graceful degradation under overload: There could be large intervals of time when input stream arrival rates exceed the maximum rate at which the DSMS can process its query workload over these streams. As shown by the *Aurora* system [CCC+02], a general approach to handle such overload situations is *load shedding*. The system load is reduced to manageable levels by dropping input tuples selectively so that the overall *quality-of-service* given by the system degrades as little as possible [CCC+02]. Ongoing work in our project adopts a similar approach, using sampling-based techniques to drop input tuples with the goal of minimizing the overall weighted error in query results incurred during overload situations.

Distributed stream processing: A final important aspect of our long-term research agenda is to incorporate distributed data stream processing techniques into the STREAM system. Data stream sources are frequently geographically dispersed, and our experiments and simulations show that processing strategies that take this fact into account can result in significant savings in computation and communication costs [OJW03, BO03]. We plan to modify STREAM to function in a distributed environment, incorporating specialized distributed data processing strategies.

References

[ABW02] A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-57.

[BBD$^+$02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[BBDM03] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[BO03] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[BW02] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-52.

[CC$^+$03] S. Chandrasekaran, O. Cooper, et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[CCC$^+$02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[FP95] S. Floyd and V. Paxson. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[Kle02] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. of the 2002 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, August 2002.

[KNV03] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, March 2003.

[MW$^+$03] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, January 2003.

[OJW03] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. (To appear).

[SQR] SQR – A Stream Query Repository. http://www-db.stanford.edu/stream/sqr.

[Vit85] J.S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, 1985.

The Gigascope Stream Database

Chuck Cranor, Theodore Johnson, Oliver Spatscheck
AT&T Labs – Research
{chuck,johnsont,spatsch}@research.att.com

Vladislav Shkapenyuk
Department of Computer Science, CMU
vshkap@cs.cmu.edu

Abstract

Managing a large scale network requires a network monitoring infrastructure. However, network monitoring is a difficult application. In response to shortcomings in the readily available tools, we have developed Gigascope, a stream database system specialized for network monitoring. In this article, we discuss some of the constraints we faced when developing the Gigascope architecture, Gigascope applications, and how Gigascope is used.

1 Introduction

Modern communications systems are very complex, involving large amounts of expensive equipment running complex protocols, and which must be continuously available. Network monitoring is necessary for many tasks, from finding network bottlenecks to diagnosing network attacks. Researchers at AT&T Labs — Research perform extensive studies using data derived from network monitoring. These experiences showed that existing methods of network monitoring and analysis had serious shortcomings.

One common method of network analysis is to gather data from a network tap using tools such as tcpdump or its underlying library pcap. While this method can provide a great amount of detail for subsequent analysis, there are many problems (as is discussed in [4]). Because so much data is gathered so quickly, the data gathering can be performed only for short periods of time. To extend the sampling period, usually only the network protocol headers are stored, which frustrates analyses which use the application layer headers of the packets (e.g. web or database transactions on a VPN). The data collection is often lossy (i.e., because of buffer overflows) necessitating data munging strategies at analysis time. The resulting data set is collected on a server in thousands to millions of files, and analyzed with hand-crafted programs (for example, Perl scripts). Managing these very large data sets is difficult — data quality problems are common and metadata is quickly lost. As a result, data analyses are often not repeatable. There are some other perhaps less obvious problems. For one, moving these data sets to a central analysis server is very expensive relative to the value of the data. For another, collecting and storing detailed information about network traffic creates privacy and security problems.

A method at the opposite end of the spectrum is to use one of the commercially available network monitoring tools. This approach also presents problems, the most significant of which is the lack of flexibility. Commercially available systems are usually closed, creating data only for their own reporting systems. Making changes to the reports generally requires negotiation with the vendor and winds up being too slow and very expensive (especially for high-speed link monitors). Also, these systems are generally expensive (and AT&T has a *very*

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

large network) and run on computers provided by the vendor. However these boxes are often not the most appropriate devices for the task.

An intermediate method uses aggregated data sets collected by the router. Two common types of data are SNMP, which collects and distributes a variety of statistics about router usage, and netflow, which is a summary of all flows (packets from a source to a destination) traversing the router. Because these data sets are highly aggregated, they present fewer problems with data management and fewer concerns about privacy and security. However, their coarse nature greatly reduces the range of analyses that can be made. Furthermore, post-collection data analysis is still made difficult by the loss of metadata and the size and unstructured nature of the collected data sets.

To improve our ability to perform various network monitoring tasks, we have developed *Gigascope*, a lightweight stream database specialized for network monitoring applications. Our first deployment of Gigascope occurred in October 2002, and currently (March 2003) we have seven deployments with many more in negotiations. In this article, we describe the application constraints which guided our choice of architecture, a brief discussion of the Gigascope system, and a discussion of Gigascope applications and use.

2 Application Constraints

In our design of the Gigascope architecture, we faced a number of constraints which guided our decisions.

2.1 Data Reduction

The single most critical function of a network monitor is to reduce the amount of data flowing through the system. The earlier the data can be reduced, the less load is placed on the computing system, and therefore the higher the data rate (or the more complex the query set) that can be supported without data loss. As a side benefit, the sooner that irrelevant data can be discarded, the less risk of privacy and security problems. Ideally, the output of the query system is reduced to a small enough size that it can be loaded into a conventional data warehouse, or even viewed directly.

2.2 Flexibility

To a person in the database community, the need for flexibility is self-evident. But in an application-specific domain such as network monitoring, flexibility can be a problem. A more flexible system is generally harder to use, harder to optimize, and easier to abuse than a less flexible system. For example, some systems, e.g. the FLAME architecture [1], propose distributing executable code modules to accomplish network monitoring, but this level of flexibility leads to a system which is very difficult to manage. We have no opportunity to perform critical optimizations and the metadata is lost, leading to problems in the post-collection analysis.

A declarative query language, such as SQL, provides a great deal of flexibility while providing enough structure to enable allow a wide variety of optimizations. However, for network applications SQL is often insufficiently flexible, leading to user frustration and to very complex and inefficient expressions for common network monitoring applications (for example, see the network traffic management queries in [7]). We found that sacrificing the purity of the query language for increased flexibility is sometimes a good tradeoff (as discussed below).

2.3 Query Language

It might be surprising to the readers of a database publication but the choice of query language is an issue. Should it be a standard database language, e.g. SQL, or a special purpose language designed to succinctly and efficiently express network monitoring queries. Consider the examples of Tribeca [8] and Hancock [2], both of which are mostly procedural. In the end we decided that the advantage of using a well-known and well-researched query language outweighed the advantages to be had from a special purpose language.

2.4 Real Time Processing

A stream database system is inherently “push-based” rather than “pull-based”, meaning that the system must handle each new packet of data when it is offered, typically by buffering the packet until it can be processed. If the buffer overflows, the system loses data. If the query system is searching for, e.g. multimedia session initiations, losing even a single packet can significantly skew the query results. Therefore the system must incorporate traditionally real-time concerns such as scheduling and buffer sizing.

2.5 Computing Device Selection

Network equipment centers are usually more severe environments than data centers. They are generally small, so that equipment space is at a premium. The network center might be in a remote location and rarely visited by a technician. Even in large network centers, access is generally restricted to qualified technicians. In either case, the equipment must run reliably with only remote maintenance for years. Network centers often have certification requirements for their equipment, which limits the selection of vendors and models. In many network centers, only 48 volt power is available (a telephony standard).

2.6 Reliability

The network monitors are often installed in remote or otherwise difficult to access locations, and must operate for months at a time. One way to ensure reliability is to use reliable server-quality hardware. However, the software system must also be highly reliable, and as a result we generally have a preference for simplicity in our software architecture.

2.7 Efficiency

A critical requirement is efficiency. In part, this requirement is dictated by limitations on devices. Space is usually very limited in network centers: we simply cannot install large computer systems. Cost is another factor. AT&T has a very large network and very many customers. Saving $10,000 on an host computer becomes important when you make hundreds or thousands of installations. Also, it is very convenient to be able to monitor a 100 Mbit/sec network with a laptop. We developed Gigascope to monitor high-speed optical networks, which can carry millions of packets per second at peak usage. Although modern CPUs operate at gigahertz frequencies, there are distressingly few CPU cycles per network bit.

2.8 User Resistance

The target user base (network analysts) are often skeptical of using database technology. In part, they are skeptical because they have not used database tools in the past and are reluctant to change. Also, many attempts to use databases to query network monitor data have failed in the past. A system that is slow, unreliable, or inflexible will be quickly discarded. Instead, we need to show that complex applications can be very quickly developed and that they will be more reliable and have better performance than a hand-crafted system.

3 Gigascope Architecture

We describe the Gigascope architecture more fully in another paper [3], so we give a brief description here and relate it to the application constraints.

Gigascope is a lightweight stream database specialized for network monitoring applications. Gigascope manages streams, or indefinite sequences of tuples. Currently, relations and continuous tables are not supported, and there are no immediate plans to incorporate them (although there are ways to perform certain foreign key joins with tables). Applications receive results by subscribing to the stream which is the output of a query.

Stream database systems must incorporate some notion of time in order to unblock operators such as aggregation and join. One method is to make the dbms a continuous query system, so that all results are continuous tables defined as the result of a query on a recent time window of the input data streams. This approach is well suited to many continuous monitoring queries, but introduces complications for network monitoring (more complex query semantics and less efficient query evaluation). The other common method is to make the dbms a pure stream database – that is, all operators transform input streams into output streams. With a pure-stream dbms, some explicit notion of time is generally needed. Network analysis queries almost always make explicit references to timestamps, so this restriction is not a serious complication. Network data generally contains many types of timestamps and sequence numbers, so Gigascope provides a mechanism for schema annotations to indicate which stream fields behave as timestamps and how.

By pushing timestamp notations into the base stream schemas and using timestamp-ness imputation (similar to type imputation) queries such as

```
Select tb, DestAS, count(*)
From IPV4
Group By time/60 as tb, getlpmid(destIP, 'asn.tbl') as destAS
```

are meaningful, as long as time (in this example, a second-granularity timestamp) has been annotated as being a timestamp.

The GSQL query language is in most respects a restriction of SQL. Some of these restrictions are related to programming effort – for example, only two-way joins are supported. Other restrictions are related to the need to find time windows in which to evaluate blocking operators such as aggregation and join. There are also some language extensions, for example the *merge* operator, which is similar to a *union* except that it merges two input streams in a timestamp order.

In order to provide the flexibility required for network analysis, GSQL supports user-written functions and operators. For example, the getlpmid function performs *longest prefix matching*, i.e., to find the most specific subnet matching an IP address. Routers use longest prefix matching when doing route lookup, so many network monitoring queries must use this function. The source data is the table asn.tbl, which is downloaded from a router. The getlpmid function will read this file when the query starts, build a search data structure, and use this data structure to perform lookups (we have built in the support for this kind of behavior). Network analysts have written highly tuned and very fast C-language modules to do longest prefix matching, one of which is now part of the Gigascope runtime library. The ability to use these special functions is a key part of Gigascope's flexibility. We note that longest prefix matching is a special kind of join, but its expression in relational operators would be complex and inefficient.

Many network analyses require that a network protocol be simulated in part or in whole. Examples include IP defragmentation and multimedia monitoring. To accommodate user-written software, we provide a view mechanism to incorporate user-defined operators. The operator exports a schema, has source queries, and a specification of how selections and projections can be pushed into the operator and its source queries. We have recently added this functionality to Gigascope; and are still experimenting with it.

Gigascope is not a monolithic dbms, instead it operates as a set of cooperating processes. A run-time system evaluates low-level queries on raw packet sources (this important performance optimization is described in [3]). The low level queries generate derived data streams that are consumed by one or more processes, which can be further query processing nodes or application programs. A process which is a query processing node also generates a data stream, and the level of nesting can continue indefinitely deep. A *registry* process contains a mapping from query names to the processes which supply the corresponding stream. To read data from a query, a process (application program or query processing node) finds the data stream source from the registry, then contacts the supplying process to subscribe to the data stream. We used this flexible organization to incorporate user-defined operators.

Gigascope is a compiled query system. When a user submits a set of queries to Gigascope, they are analyzed and optimized, and a set of C and C++ language modules are generated. These are compiled and linked into a run time system, and executed. At this point, the user can access the output data streams. Although the approach of compiled queries places some limits on flexibility, we have found that this approach produces the most efficient system. In a recent application, Gigascope processed 1.2 million packets per second using a moderately priced dual 2.4 Ghz rack mount server.

4 Modes of Use

We have made several Gigascope deployments in the AT&T network and at AT&T customer sites, and we have noticed typical patterns of use. Typically, we define a set of queries, then backhaul the results to a centralized server, where we ingest the data into a data warehouse. We use the data warehouse to generate reports and to populate web pages. Gigascope generally reduces the data volumes to a point where data transfer and data warehouse ingestion costs are quite moderate.

In some occasions, the data volumes are so large, or the backhaul network is so slow, that it is much better to keep the data warehouse on the network monitor. In this case, we just need to spend a little more money on the server's disk drives. The monitor serves web pages, and on demand transfers reports back to an centralized analysis server.

The procedure of gathering Gigascope output and periodically loading it into a data warehouse has worked well for our applications. However there is some delay between data generation and data loading. It would be an interesting experiment to use a continuous query system such as STREAM [6] to monitor Gigascope output.

Another mode of use is to use Gigascope to search for events in a packet stream. We have written a translator to convert Snort [5] rules into GSQL queries, and have written a trigger processing application to interpret the results of the queries. However we have made only preliminary experiments with this type of system.

4.1 Applications

We have used Gigascope in a number of different applications, which present their own challenges.

- **Network analysis:** We have developed a standard query suite to be used for analyzing the health and status of a network. This data is typically backhauled to a data warehouse then correlated with other network data, especially BGP (Border Gateway Protocol) data. The resulting reports give a picture on network health and usage, and correlation with BGP (router) reports often indicates the source of network problems. We have applied this type of analysis to AT&T's network, and to current or potential customer networks.

- **Protocol analysis:** Applications do not just run on a single machine anymore, these days there are many instances of applications which run across large numbers of diverse machines. In these kind of applications, the (wide-area) network is no longer just a dumb pipe, it is a critical part of the properly functioning system. In a recent application of Gigascope, we helped a customer debug performance problems by correlating Gigascope's protocol-level measurements with user experience and with router reports.

- **Research:** The highly flexible nature of Gigascope makes it well-suited as a research tool. Experimental analysis code can be quickly grafted into a high speed monitoring system. Currently we are working with a research team which has developed new algorithms for monitoring the quality of video streams.

- **Intrusion detection:** Network intrusion detection can be accomplished by expressing intrusion rules as GSQL queries and feeding the result streams into a trigger processing application. We have made only preliminary experiments with this kind of application.

5 Conclusions

Gigascope is an applied industrial research project, and as a result has developed in ways different than other recent stream database research projects. The GSQL query language is less expressive and the Gigascope architecture and query processing algorithms are less sophisticated that of, e.g. [6, 9]. However, we have been able to develop a stream database network monitor which operates at very high speeds, is stable enough for unattended operation, and which has been deployed for widely diverse applications. To do so, we have made several query language and architectural innovations, which are more fully described in [3].

References

[1] K. G. Anagnostakis, S. Ioannidis, S. Miltchev, J. Ioannidis, M. B. Greenwald, and J. M. Smith. Efficient packet monitoring for network management. In *Proc. IFIP/IEEE Network Operations and Management Symposium (NOMS)*, 2002.

[2] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, and F. Smith. Hancock: A language for extracting signatures from data streams. In *Proc. Sixth Intl. Conf. on Knowledge Discovery and Data Mining*, pages 9–17, 2000.

[3] C. Cranoe, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *Proc. ACM SIGMOD Conf.*, 2003.

[4] V. Paxton. Some not-so-pretty admissions about dealing with internet measurements. Invited talk, Workshop on Network-Related Data Management, 2001.

[5] snort.org. Snort home page. http://www.snort.org/.

[6] Stanford Stream Data Manager. Stream home page. http://www-db.stanford.edu/stream/.

[7] Stanford Stream Data Manager. Stream query repository. http://www-db.stanford.edu/stream/sqr/.

[8] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proc. USENIX Annual Technical Conf.*, 1998.

[9] The Telegraph project. Telegraph home page. http://telegraph.cs.berkeley.edu/.

Applying Punctuation Schemes to Queries Over Continuous Data Streams

Peter A. Tucker, David Maier, Tim Sheard
OGI School of Science & Engineering at OHSU
{ptucker,maier,sheard}@cse.ogi.edu

Abstract

In-stream punctuations can be used in a variety of ways to enable applications over data streams. In previous work, we focused on using punctuations to improve specific query operators, namely blocking operators and unbounded stateful operators. We report here on our ongoing work in exploiting punctuation semantics. We discuss three applications that illustrate how punctuations can be used. We also discuss our current focus – deciding when punctuations can be used to improve entire queries. We use a notion of compactness to characterize the utility of a given punctuation scheme.

1 Introduction

Our previous work has shown that punctuations make a wider range of query operators applicable to data streams [19]. Queries that involve blocking operators such as group-by and difference can be unblocked using punctuations. Further, punctuations can reduce the state requirements for stateful operators such as join and duplicate elimination. Punctuations allow users to pose more kinds of queries over data streams.

Much of the feedback on this work can be reduced to two questions: How do punctuations get embedded into data streams? and What kinds of applications can take advantage of punctuations? We discuss three applications that address both questions: a temperature monitoring system, a network traffic monitoring system, and an online auction system. Each application uses punctuations in different ways to answer user queries.

To date, our work has focused on individual query operators. A more significant issue is the effects of punctuations on an entire query. Certainly, understanding the effects of punctuations on individual operators helps, but we need a formal way of determining if a set of punctuations can enhance a query over data streams (or the dual question of which queries are able to take advantage of a given set of punctuations?) For a set of punctuations to be effective over a stream of data, it must cover designated subspaces of the output domain using a finite number of punctuations. A set of punctuations is said to be *compact* if it has this property for all subspaces of the output domain.

This paper is organized as follows: Section 2 gives a brief background of punctuation semantics and an overview of related work. In Section 3 we describe three applications that take advantage of punctuations in a data stream. Section 4 reports on our current effort using compactness to characterize the queries that can exploit particular punctuations. We conclude in Section 5.

2 A Brief Background on Punctuated Streams

In our discussion of data streams, we will use $S[i]$ to mean the first i elements in stream S, and for $j > i$, $S[i \rightarrow j]$ are the elements in S from $i + 1$ to j. Note that $S[i] = S[0 \rightarrow i]$.

Punctuations denote the end of a subset of data. Therefore, a punctuation can be seen as a predicate over data in the stream. A data item t from a stream is said to *match* a punctuation p if t evaluates to `true` for the predicate described by p. We denote this as a function: $match(t, p)$. Further, we say that a data stream S is *grammatical* if, $\forall i, j \in \omega . j > i, p \in S[i] \wedge t \in S[i \rightarrow j] \Rightarrow \neg match(t, p)$. That is, a stream S is grammatical if no data items in S that match punctuation p come after p. In our work, we expect streams to be grammatical.

One can imagine many different formats for punctuations. For example, since each is a predicate over data, it might be represented as a function. We instead represent a punctuation as a series of patterns over attributes of data items in the stream, as it allows easier manipulation of punctuation. Our patterns are listed in Table 1. A data item matches a punctuation if the value of each attribute in the data item matches the corresponding pattern in the punctuation. For example, given a punctuation $p =< *, [1, 6] >$, if t_1 and t_2 are data items such that $t_1 =< 3, 5 >$ and $t_2 =< 3, 7 >$, then $match(t_1, p) = true$ and $match(t_2, p) = false$. This scheme is simple, but has the useful property that the logical 'and' of two punctuations is itself a punctuation.

ψ	Empty – does not match any values	$*$	Wildcard – matches any value
c	Constant – matches only c	$\{c_1, c_2, c_3\}$	List – matches values in the list
(c_1, c_2)	Exclusive Range – matches values strictly in the range	$[c_1, c_2]$	Inclusive Range

Table 1: Patterns for punctuations

In many cases, the output of a relational operator is determined by values of specific attributes of input data items. For example, the join operator compares values of join attributes, and group-by arranges data items by the values of the grouping attributes. It is therefore useful to be able to discuss punctuations that only focus on specific attributes. For a schema R over a data stream, we say that a punctuation *describes* a set of attributes $X \in R$ if, for all attributes $a \in R - X$, $p(a) = *$. Consider the group-by operator as an example. Punctuations that describe the grouping attributes are most useful, since they ensure that all data items for a particular group have arrived.

2.1 Additional Behavior Due to Punctuations

We initially considered punctuations simply as a way to unblock blocking operators and decrease the amount of state required by unbounded stateful operators. This vague intuition needed to be formalized into a notion of correct behavior for operators in the presence of punctuations. To this end, we formulated *punctuation behaviors* for individual operators. There are three kinds of behaviors: *Pass behavior* describes the additional data items that can be output as a result of punctuations. *Keep behavior* describes the state required for the operator to continue to process incoming data correctly. (Alternatively, one could talk about purge behavior that describes the state that is no longer required.) Finally, *propagation behavior* describes what kinds of punctuation can be output from an operator. These behaviors define operation beyond normal operator behavior.

Consider for example the following three operators: select, duplicate elimination, and group-by. The select operator is not blocking – data items that pass the selection predicate are immediately output. Therefore, select does not have additional pass behavior. Further, select does not maintain any state, and therefore has no additional keep behavior. The select operator should propagate punctuations, though. In this case, select can simply output punctuations as they arrive. If the input stream is grammatical, then the output from select will also be grammatical.

Duplicate elimination is a bit harder. Like select, it does not have additional pass behavior. However, it does maintain state. In a simple implementation, unique data items are maintained in a hash table. When new tuples arrive, they are used to probe the hash table to filter out duplicate values. This hash table will grow without bound on unbounded input, in the absence of punctuation. When a punctuation arrives, we know that no more data items will arrive that match it, so any data items in the hash table that match that punctuation can be purged. The keep behavior here is to retain only data items that have not matched incoming punctuation. Finally, we want duplicate elimination to propagate punctuations. Like select, punctuations are output as they arrive.

Finally, we consider group-by. Since group-by is a blocking operator, we need additional pass behavior to output correct results as soon as possible. If the input stream contains punctuations that describe the grouping attribute set, then group-by can pass results for group that match the punctuation. Further, since groups that have been output are complete, we no longer need to keep state for those groups. Thus, the keep behavior is to retain groups that do not match incoming punctuation describing the grouping attribute set. Finally, group-by can propagate punctuation stating that no more data items for groups that have been output will appear. Notice that group-by can only be improved by with punctuations that describe the grouping attribute. Other punctuations are not generally useful (unless they can be combined into useful punctuations).

2.2 Building on Punctuation Behaviors

Punctuation behaviors give us a good, high-level view of how individual operators should process punctuations. We use punctuation behaviors in two ways: First, we have defined three kinds of *punctuation invariants* [19] based in set theory to define various behaviors: pass invariants, keep invariants, and propagation invariants. These invariants define when a punctuation-aware operator is a reasonable counterpart to a relational operator.

The second use for punctuation behaviors is in our framework for stream iterators. Our framework is implemented in the functional programming language Haskell [10], a lazy-evaluation language. We have defined the general behavior of stream iterators using two generic functions, and provide specific helper functions to the generic function to define specific iterators. Each punctuation behaviors appears as a specific function to be executed on each data item iteratively. Our model allows us to prove that our implementations of punctuation-aware stream iterators obey the invariants. Proofs for two implementations are available elsewhere [19].

2.3 Related Work

Querying over continuous data streams has become a popular research topic. The Tangram stream query processing system by Parker et al. [17] is an early investigation attempting to use database technology on a data stream. They include in their discussion blocking relational operators such as sort and difference, but do not discuss how these behave in the presence of continuous streams. Parker also details a model for stream transducers [16] very similar to our model of stream iterators, though he limits the discussion to unary transducers.

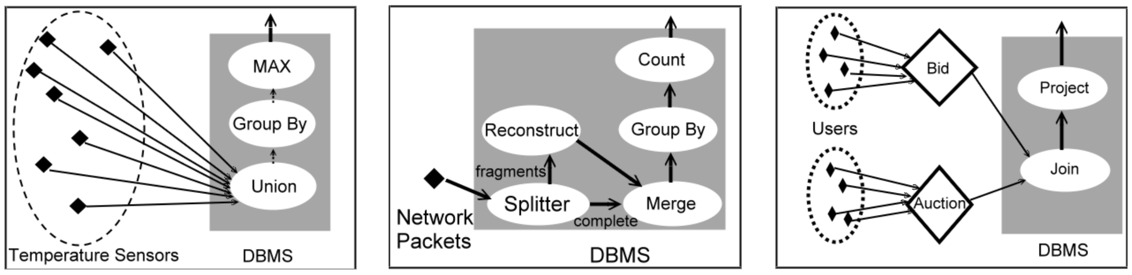
Many data stream systems [2, 3, 7, 8, 14, 21] use *windows* over data streams to bound the amount of memory considered during query execution. Windows define a range of data items to process, where the range is normally defined by the number of data items or by a timespan. Typically, windows come in two flavors: *fixed* (or *landmark*) and *sliding* (or *moving*). In fixed windows, a landmark is defined in the data stream (e.g. every 100^{th} data item or at the beginning of each hour), and the query is processed over data items from that landmark up to the current data item. Sliding windows are defined to be a certain size (e.g. the last 100 data items or data items in the past 10 minutes), and queries are processed over data items from the current one back to the size of the window. Kang et al. [11] evaluate different join algorithms over sliding windows based on a number of external factors. Zhu and Shasha [21] add a third kind of window called *damped* windows, where previous sliding windows are evaluated with the current sliding window at exponentially decreasing weights. Chandrasekaran et al. [3] discuss a more general window model, in which the start-point and end-point of each window processed by a query can change over time.

Another common approach to querying over data streams is to maintain a summary of the dataset, and output approximate results to queries based on that summary. Gilbert et al. [8] use wavelets to approximate data streams from cellular telephones. Gehrke et al. [7] use focused histograms to compute approximate, correlated aggregate values over data streams in a single pass. Dobra et al. [5] also look at “sketch” summaries of data streams to approximate aggregate queries.

There have been a number of other system issues addressed. The system discussed by Madden and Franklin [12] implements operators that efficiently combine data from stored (file-based) and streaming sources. Additionally, they have enhanced their adaptive query processing system [1] to support queries over data streams [13]. Finally, work on the Aurora system [2] investigates optimization tactics for queries over data streams and ways to intelligently shed load when the system is about to run out of memory.

3 Implementations of Punctuated Streams

Punctuations can be added to applications in various ways. The following examples illustrate three different methods for embedding punctuations in data streams, and different ways to exploit those punctuations. The applications are shown in Figure 3.



(a) (b) (c)

Figure 1: (a) Query plan for temperature monitoring in a warehouse. (b) Query plan for monitoring network packets. (c) Query plan for online auction system.

3.1 Temperature Monitoring in a Warehouse

Consider a warehouse that contains temperature-sensitive merchandise. Temperature sensors are deployed throughout the warehouse, reporting temperatures at regular intervals to a main system. A query the warehouse manager might want to pose to the data stream is, “Report the maximum temperature at any sensor each hour.” This simple query can be expressed easily in SQL:

```
SELECT MAX(tmp)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorn)
GROUP BY hour;
```

That is, union data from all streams and output the maximum temperature for each hour. Unfortunately, this query cannot execute over an infinite stream, since group-by will block until it has read all of its input. As we have discussed, punctuations that mark the end of each group unblock the group-by operator. The query in this case is grouping on values in the `hour` attribute, which is non-decreasing. The sensor itself can be designed to emit punctuations at the end of each hour, and these punctuations will unblock the query.

We have simulated this scenario using the Niagara Query Engine [15]. Temperature reports are generated by stream source applications built in-house, and sent to Niagara to evaluate the above query. A stream source is smart enough to know that, when it begins reporting temperatures for a new hour, no reports for the previous hour will be output from that sensor. Therefore, each stream source embeds a punctuation at the end of every hour. In Niagara, we modified the union operator to output punctuations only when equivalent punctuations have arrived from each input. For example, when a punctuation matching hour 4 arrives for some source, union holds that punctuation in state until a punctuation matching hour 4 has arrived from all sources. When that happens, the punctuation can be output. The group-by operator maintains a hash table for each group. We modified group-by to look for punctuations that describe the grouping attribute. When punctuations arrive that describe the grouping attribute, it outputs its results for that group and removes the group from the hash table.

3.2 Gigascope

The Gigascope application [4] is a flexible network monitor that executes relational-style queries over streams of network packets. One example query is to report the count of IP packets every 60 seconds. The SQL for this query is (where `time` is in seconds):

```
SELECT tb, count(*)
FROM IP
GROUP BY time/60 AS tb
```

Query operators in Gigascope can take advantage of non-decreasing attributes in the IP stream. When the `tb` attribute moves to the next minute, group-by knows it can output results for the previous minute since the time attribute is non-decreasing. However, there is still a problem. Packets can occasionally be fragmented by a router in the network path. It is the responsibility of the receiver to put fragmented packets back together. Fragmented packets are split out from the complete packets and reconstructed. When a packet is completely reconstructed, it is merged back with the complete packets. However, the merge operation must maintain the order on time. The problem lies in the rates of the two streams. Fragmented packets are rare, so the fragmented stream has a much lower rate than the unfragmented one. Since the merge is order-preserving, it cannot simply output a packet from the faster stream until it is sure there is nothing from the slower stream that should go before it.

In this example, the operator receiving packets is designed to embed punctuation, marking the end of some time period based on the most recent timestamp received. The splitter passes the punctuation to both the merge and the reconstruct operators. The reconstruct operator outputs the punctuation as soon as all fragments with a timestamp less than the punctuation have been output. Then merge can safely output results from the faster stream. (We thank Ted Johnson and the Gigascope group for this example.)

3.3 Online Auction Monitoring

For the final example, consider an online auction system such as eBay [6]. This system monitors bids over items available for auction. One can imagine two streams in this system: one containing new items to sell (the *auction* stream) and one containing bids for items for sale (the *bid* stream). One possible query in this system outputs legal bids (bids that have not arrived past the end of the auction) for a particular item. The SQL for such a query is as follows:

```
SELECT A.itemname, B.price
FROM Auction A, Bid B
WHERE A.id=B.auctionid AND A.expires > B.datetime
```

Since `A.id` uniquely determines data for a given auction, we can use an extension join [9] to join items in the bid stream with items in the auction stream, and only output bids that are for non-completed auctions. The join uses a hash table for items from the auction input, and so requires unbounded state. Punctuations that denote the end of a specific time period can allow us to purge state from the auction hash table when we know that no more bids made before the auction expire time will arrive. However, since our system uses the Internet to pass bids, we cannot rely on values in the `datetime` field being in order. In fact, bids may arrive much later than the auction expiration, even if their datetime value is less than the expires value for the auction. It is acceptable to adopt a timeout policy for late bids. Bids that were posted before the end of an auction but arrive beyond some slack time passed the end of the auction are ignored.

We use a new operator called the *punctuator* for this case, and place it in the query plan between the join and the bid stream. The punctuator embeds punctuations in the bid stream stating that no more bids will arrive later than a certain time, where time is the current system time plus the allowable slack time. Since bids may arrive later than slack time, the punctuator also enforces its punctuations. Here the punctuator filters out all late-arriving bids. We are currently working on a simulation of this system using the Niagara Query Engine.

4 Defining Compact Covers using Punctuation Schemes

Clearly, some queries can take advantage of particular punctuations and others cannot. In the warehouse example we group on the hour attribute. We know that eventually the end of an hour will occur, and can embed punctuations marking the end of each hour. Punctuations cannot help a query that groups on sensorid, since there will be no end of reports from a sensor generally. For example, the following query cannot make use of hourly punctuations:

```
SELECT sensorid, AVG(temperature)
FROM (SELECT * FROM sensor1
      UNION
      SELECT * FROM sensor2
      UNION
      ...
      UNION
      SELECT * FROM sensorn)
GROUP BY sensorid;
```

Such cases led us to the notion of compact sets [18]. The definition of compactness in analysis suggested a similar notion for punctuations (though they are not exactly equivalent notions). We say that the *interpretation* $\mathcal{I}(D,p)$ of a punctuation p in a dataspace D is the subspace $S \subseteq D$ containing data items that match the punctuation. Clearly, $t \in D \wedge match(t,p) \Leftrightarrow t \in \mathcal{I}(D,p)$. We shorten $\mathcal{I}(D,p)$ to $\mathcal{I}(p)$ when D can be inferred from context. Figure 4(a) shows one possible interpretation of a punctuation.

We first define the notion of a cover in terms of subspaces. Let $\Sigma = \{S|S \text{ is a subspace of } D\}$. Given some subspace $Q \subseteq D$, we say that Σ *covers* Q if $\forall t \in Q, \exists S \in \Sigma | t \in S$. Further, Q is *compact* in Σ if $\exists \Sigma' \subseteq \Sigma$ such that $|\Sigma'|$ is finite and Σ' covers Q. Now, let $\Theta = \{Q|Q \text{ is a subspace of } D\}$. We say that Θ is compact in Σ if $\forall Q \in \Theta$, Q is compact in Σ.

A *punctuation scheme* $\mathcal{P} = \{p_1, p_2, \ldots\}$ is the set of all punctuations that will be emitted from a stream iterator or stream source. Then $\Sigma_{\mathcal{P}} = \bigcup_{p \in \mathcal{P}} \mathcal{I}(p)$ is the subspace defined by the collection of interpretations of punctuations in $\mathcal{P}$. Figure 4(b) shows subspaces corresponding to two queries with different groupings. If $\mathcal{P}$ is by hour, by sensor punctuations, then the query that is grouping on hour is compact for $\mathcal{P}$. However, the query that is grouping on sensorid is not compact for $\mathcal{P}$. Our goal is to use compactness of subspaces of the output dataspaces (or their *preimages* in the input dataspace) to characterize when a punctuation scheme aids a particular operator or query.

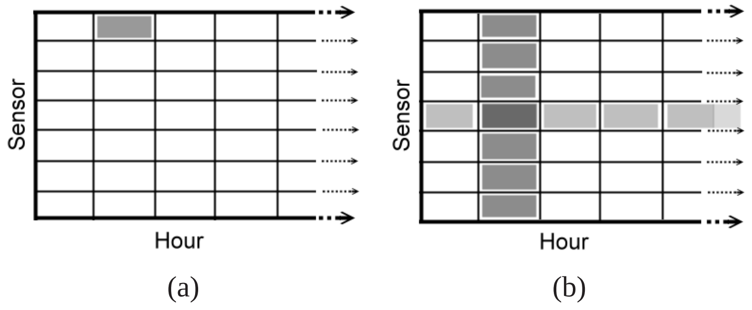


Figure 2: (a) Interpretation of a single punctuation – all reports from a single sensor in one hour (b) The darker rectangles, containing all data items for a particular hour, can be covered by a finite set of punctuations, and thus is compact. The lighter rectangles, containing all reports from a specific sensor, cannot be covered by a finite set of punctuations, and thus is not compact.

5 Conclusions and Future Work

We have shown how punctuations can be used in various applications. We illustrated how punctuations can be embedded by the stream source, or generated inside the query plan. The warehouse example illustrates how punctuations can be embedded by the stream source, unblocking queries involving grouping operators. The Gigascope example illustrates how punctuations can be used to allow progress in the absence of data items on an input. The auction example illustrates how a punctuation can be used to purge unnecessary state, and how a punctuation operator can enforce its punctuations in the presence of unreliable inputs. We have also discussed our current focus of evaluating entire queries relative to a punctuation scheme, using the notion of compactness. There is much to do to formalize this notion.

There are a number of other directions we can take related to punctuated data stream. One question is, are there other operator implementations that are normally inappropriate for data streams that can be aided with punctuations? For example, our work with join has mostly focused on the symmetric hash join [20]. Are there other join implementations that might perform better with punctuations? Another interesting question involves tree-structured data such as XML. Our focus so far has been on relational data. There are many interesting issues in trying to punctuation elements that themselves have elements.

6 Acknowledgements

The authors thank Kristin Tufte, Vassilis Papadimos, Jin Li, and Lois Delcambre for the discussions they had, Ted Johnson for discussions regarding Gigascope, Jennifer Widom and Samuel Madden for their comments, Raghu Ramakrishnan for his pointers to related work, and David DeWitt for his advice on the direction of this work. More information on punctuated streams can be found on http://www.cse.ogi.edu/˜ptucker/PStream/. Funding for this work was provided by DARPA through NAVY/SPAWAR Contract No. N66001-99-1-8908 and by NSF ITR award IIS0086002.

References

[1] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 261–272, May 2000.

[2] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. In *Proceedings of the 28th International Conference on Very Large Data Bases*, Aug. 2002.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*, pages 269–280, Jan. 2003.

[4] C. Cranor, T. Johnson, V. Shkapenyuk, and O. Spatscheck. Gigascope: A stream database for network applications. In *2003 ACM SIGMOD International Conference on Management of Data (to appear)*, June 2003.

[5] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *2002 ACM SIGMOD International Conference on Management of Data*, pages 61–72, June 2002.

[6] eBay home page. http://www.ebay.com/.

[7] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continuous data streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 13–24, May 2001.

[8] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. J. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 79–88, Sept. 2001.

[9] P. Honeyman. Extension joins. In *Proceeding of the 6th International Conference on Very Large Data Bases*, pages 239–244, Oct. 1980.

[10] P. Hudak. *The Haskell School of Expression: Learning Functional Programming through Multimedia*. Cambridge University Press, 2000.

[11] J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *Proceedings of the 19th International Conference on Data Engineering*, Mar. 2003.

[12] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering*, pages 555–566, Feb. 2002.

[13] S. Madden, M. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of the ACM Special Interest Group on Management of Data*, pages 49–60, June 2002.

[14] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximations in a data stream management system. In *Proceedings of the 2003 Conference on Innovative Data Systems Research*, pages 245–256, Jan. 2003.

[15] J. Naughton, D. DeWitt, D. Maier, J. Chen, L. Galanis, K. Tufte, J. Kang, Q. Luo, N. Prakash, and F. Tian. The Niagara query system. *The IEEE Data Engineering Bulletin*, 24(2):27–33, June 2000.

[16] D. S. Parker. Stream data analysis in Prolog. In L. Sterling, editor, *The Practice of Prolog*, chapter 8. MIT Press, 1990.

[17] D. S. Parker, R. R. Muntz, and L. Chau. The Tangram stream query processing system. In *Proceedings of the 5th International Conference on Data Engineering*, Feb. 1989.

[18] W. Rudin. *Principles of Mathematical Analysis*. McGraw-Hill, Inc., 1964.

[19] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Enhancing relational operators for querying over punctuated data streams. *Transactions on Knowledge and Data Engineering (to appear)*, May 2003.

[20] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proceedings of the 1st International Conference of Parallel and Distributed Information Systems*, pages 68–77, Dec. 1991.

[21] Y. Zhu and D. Shasha. StatStream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 358–369, Aug. 2002.

High-Performance XML Filtering: An Overview of YFilter

Yanlei Diao, Michael J. Franklin

University of California, Berkeley, CA 94720
{diaoyl, franklin}@cs.berkeley.edu

Abstract

We have developed YFilter, an XML filtering system that provides fast, on-the-fly matching of XML-encoded data to large numbers of query specifications containing constraints on both structure and content. YFilter encodes path expressions using a novel NFA-based approach that enables highly-efficient, shared processing for large numbers of XPath expressions. In this paper, we provide a brief technical overview of YFilter, focusing on the NFA model, its implementation, and its performance characteristics.

1 Introduction

Today, it is widely agreed that in distributed computing scenarios such as Web Services, data and application integration, and personalized content delivery, XML is the way that data to be exchanged will be encoded. This use of XML has spawned significant interest in techniques for filtering XML data. In an XML filtering system, continuously arriving streams of XML documents are passed through a filtering engine where documents are matched to query specifications representing data interests of users or applications, and the matched documents are delivered accordingly. Queries in these systems are expressed in a language such as XPath [4], which is used to specify constraints over both structure (using path expressions) and content (using value-based predicates).

An earlier project, XFilter [1], pioneered the use of event-based parsing and *Finite State Machines* (FSMs) for fast structure-oriented XML filtering. In XFilter, XPath expressions are converted in to FSMs by mapping location steps of the the expressions to machine states. Arriving XML documents are then parsed with an event-based (SAX) parser, and the events raised during parsing are used to drive the FSMs through their various transitions. A query is determined to match a document if during parsing an accepting state for that query is reached. In XFilter, a separate FSM is created for each distinct path expression and a sophisticated indexing scheme is used during processing to locate potentially relevant machines and to execute those machines simultaneously. The indexing scheme and several optimizations provide a substantial performance improvement over a more naive approach. The drawback, however, is that by creating a separate FSM for each distinct query, XFilter fails to exploit commonality among the path expressions, and thus, may perform redundant work.

Based on this insight, we have developed YFilter, an XML filtering system aimed at providing efficient filtering for large numbers (e.g., 10's or 100's of thousands) of query specifications. The key innovation in YFilter is an Nondeterministic Finite Automaton (NFA)-based representation of path expressions which combines all queries into a *single* machine. YFilter exploits commonality among path queries by merging the common prefixes of the paths so that they are processed at most once. The resulting shared processing provides tremendous improvements in structure matching performance over algorithms that do not share such processing or exploit sharing to a more limited extent. The NFA-based implementation also provides additional benefits including a relatively small number of machine states, incremental machine construction, and ease of maintenance.

An important challenge that arises due to the shared structure matching approach of YFilter is the handling of value-based predicates that address contents of elements. We have developed two alternative approaches to handling such predicates. One approach evaluates predicates as soon as the addressed elements are read from a document, while the other delays predicate evaluation until the corresponding path expression has been entirely matched. A further complication that arises in this regard is that of “nested paths”. Since predicates may also reference other elements in an XML document, we employ a query decomposition scheme to take advantage of the shared path processing, and use special post-processing to return the final query evaluation results.

The remainder of this paper is organized as follows. The logical model of the NFA-based shared path processing approach is presented in Section 2. The implementation of the approach and techniques for predicate evaluation are described in some detail in Section 3. Section 4 discusses related work, and Section 5 presents conclusions and future work.

2 An NFA-based Model for Shared Path Processing

The basic path matching engine of YFilter handles query specifications that are written in a subset of XPath.[1] XPath allows parts of XML documents to be addressed according to their logical structure. A query path expression in XPath is composed of a sequence of location steps. Each location step consists of an axis, a node test and zero or more predicates. An axis specifies the hierarchical relationship between the nodes. We focus on two common axes: the parent-child operator ’/’, and the ancestor-descendent operator ”//”. We support node tests that are specified by either an element name or the wildcard operator ’*’ (which matches any element name). Predicates can be applied to address contents of elements or to reference other elements in the document

2.1 An NFA-based Model with an Output Function

Any single path expression written using the axes and node tests described above can be transformed into a regular expression. Thus, there exists a Finite State Machine (FSM) that accepts the language described by such a path expression [9]. In YFilter, we combine all of the path queries into a single FSM that takes a form of *Nondeterministic Finite Automaton* (NFA). All common prefixes of the paths are represented only once in the NFA.

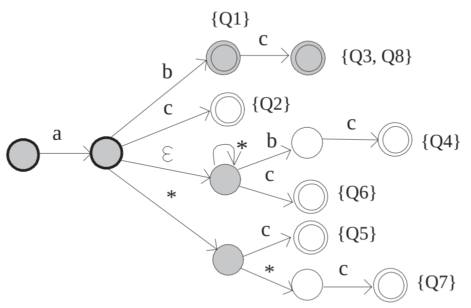
Figure 1 shows an example of such an NFA, representing eight queries (we describe the process for constructing such a machine in the following section). A circle denotes a state. Two concentric circles denote an accepting state; such states are also marked with the IDs of the queries they represent. A directed edge represents a transition. The symbol on an edge represents the input that triggers the transition. The special symbol ”*” matches any element. The symbol ”ϵ” is used to mark a transition that requires no input. In the figure, shaded circles represent states shared by queries. Note that the common prefixes of all the queries are shared. Also note that the NFA contains multiple accepting states. While each query in the NFA has only a single accepting state, the NFA represents multiple queries. Identical (and structurally equivalent) queries share the same accepting state (recall that at the point in the discussion, we are not considering predicates).

This NFA can be formally defined as a Moore Machine [9]. The output function of the Moore Machine here is a mapping from the set of accepting states to a partitioning of identifiers of all queries in the system, where each partition contains the identifiers of all the queries that share the accepting state.

[1] In more recent work we show how to use YFilter to handle more sophisticated queries written in XQuery [6].

Q1=/a/b
Q2=/a/c
Q3=/a/b/c
Q4=/a//b/c
Q5=/a/*/c
Q6=/a//c
Q7=/a/*/*/c
Q8=/a/b/c

(a) XPath queries



(b) A corresponding NFA

Figure 1: XPath queries and their representation in YFilter

Some Comments on Efficiency. A key benefit of using an NFA-based approach is the tremendous reduction in machine size it affords. It is reasonable to be concerned that using an NFA-based model could lead to performance problems due to (for example) the need to support multiple transitions from each state. A standard technique for avoiding such overhead is to convert the NFA into an equivalent DFA [9]. A straightforward conversion could theoretically result in severe scalability problems due to an explosion in the number states. But, as pointed out in [8], this explosion can be avoided in many cases by placing restrictions on the set of DTDs (i.e., document types) and queries supported, and lazily constructing the DFA.

Our experimental results (reported in [5], however, indicate that such concerns about NFA performance in this environment are unwarranted. In fact, in the YFilter system, path evaluation (using the NFA) is sufficiently fast, that it is in many cases not the dominant cost of filtering. Rather, other costs such as document parsing and result collection are often more expensive than the basic path matching. Thus, while it may in fact be possible to further improve path matching speed, we believe that the substantial benefits of flexibility and ease of maintenance provided by the NFA model outweigh any marginal performance improvements that remain to be gained by even faster path matching.

2.2 Constructing a Combined NFA

Having presented the basic NFA model used by YFilter, we now describe an incremental process for NFA construction and maintenance. The shared NFA shown in Figure 1 was the result of applying this process to the eight queries shown in that figure.

The four basic location steps in our subset of XPath are ”/a”, ”//a”, ”/*” and ”//*”, where ’a’ is an arbitrary symbol from the alphabet consisting of all elements defined in a DTD, and ’*’ is the wildcard operator. Figure 2(a) shows the directed graphs, called NFA fragments, that correspond to these basic location steps. Note that in the NFA fragments constructed for location steps with ”//”, we introduce an ϵ-transition moving to a state with a self-loop. This ϵ-transition is needed so that when combining NFA fragments representing ”//” and ”/” steps, the resulting NFA accurately maintains the different semantics of both steps (see the examples in Figure 2(b) below). The NFA for a path expression, denoted as NFA_p, can be built by concatenating all the NFA fragments for its location steps. The final state of this NFA_p is the (only) accepting state for the expression.

NFA_ps are combined into a single NFA as follows: There is a single initial state shared by all NFA_ps. To insert a new NFA_p, we traverse the combined NFA until either: 1) the accepting state of the NFA_p is reached, or 2) a state is reached for which there is no transition that matches the corresponding transition of the NFA_p. In the first case, we make that final state an accepting state (if it is not already one) and add the query ID to the query set associated with the accepting state. In the second case, we create a new branch from the last state reached in the combined NFA. This branch consists of the mismatched transition and the remainder of the

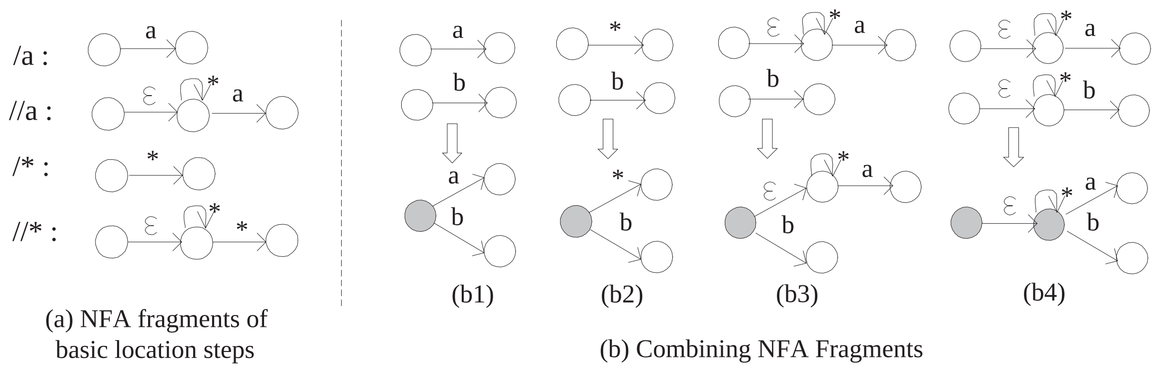

/a :
a
//a :
ε
*
a
/* :
*
//* :
ε
*
*
(a) NFA fragments of basic location steps
a
b
*
b
ε
*
a
b
ε
*
a
ε
*
b
(b1)
(b2)
(b3)
(b4)
(b) Combining NFA Fragments

Figure 2: NFA fragments for location steps, and examples of merging NFA fragments

NFA_p. Figure 2(b) provides four examples of this process.

It is important to note that because NFA construction in YFilter is an incremental process, new queries can easily be added to an existing system. This ease of maintenance is a key benefit of the NFA-based approach.

3 Implementation of the NFA-based Path Processing

The previous section described YFilter's NFA model and its logical construction. In this section, we present the implementation of the NFA approach and describe its execution.

3.1 Implementing the NFA

For efficiency we implement the NFA using a hash table-based approach. Such approaches have been shown to have low time complexity for inserting/deleting states, inserting/deleting transitions, and actually performing the transitions [12]. In this approach, a data structure is created for each state, containing: 1) The ID of the state, 2) type information (i.e., if it is an accepting state or a //-child as described below), 3) a small hash table that contains all the legal transitions from that state, and 4) for accepting states, an ID list of the corresponding queries.

The transition hash table for each state contains [symbol, stateID] pairs where the symbol, which is the key, indicates the label of the outgoing transition (i.e., element name, '*', or 'ϵ') and the stateID identifies the child state that the transition leads to. Note that the child states of the 'ϵ' transitions are treated specially. Recall that such states have a self-loop marked with '*' (see Figure 2(a)). For such states, (called "//-child" states) we do not index the self-loop. As described in the next section, this is possible because transitions marked with 'ϵ' are treated specially by the execution mechanism.

3.2 Executing the NFA

Having walked through the logical construction and physical implementation we can now describe the execution of the machine. Following the XFilter approach, we chose to execute the NFA in an event-driven fashion. As an arriving document is parsed, the events raised by the parser drive the transitions in the NFA. The nesting of XML elements requires that when an "end-of-element" event is raised, NFA execution must backtrack to the states it was in when the corresponding "start-of-element" was raised. A stack mechanism is used to enable the backtracking. Since many states can be active simultaneously in an NFA, the run-time stack mechanism must be capable of tracking multiple active paths. Details are described in the following.

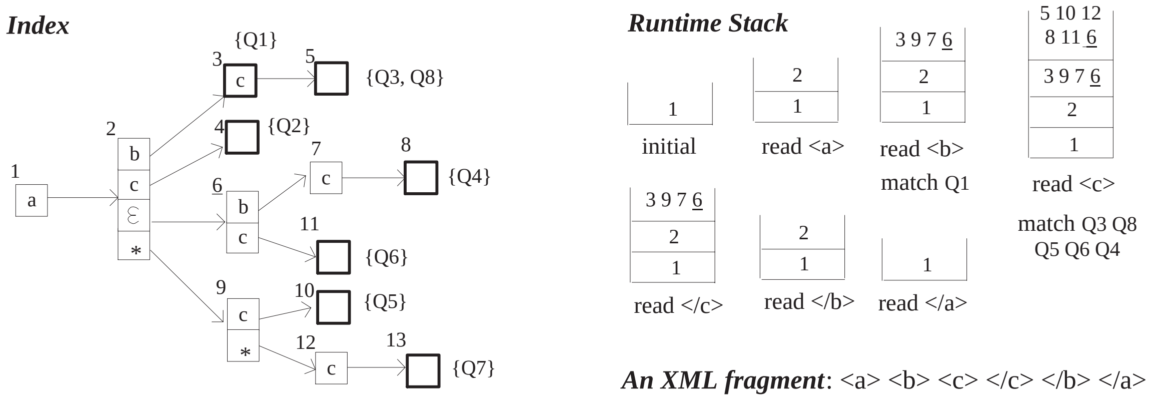


Figure 3: An example of the NFA execution

Start of Document. When an XML document arrives to be parsed, the execution of the NFA begins at the initial state. That is, the common initial state is pushed to the runtime stack as the active state.

Start of Element. When a new element name is read from the document, the NFA execution follows all matching transitions from all currently active states, as follows. For each active state, four checks are performed: 1) The incoming element name is looked up in the state's hash table. If it is present, the corresponding stateID is added to a set of "target states". 2) The '*' symbol is also looked up in the hash table. If it exists, its stateID is also added to the set of target states. Since the '*' symbol matches any element name, a transition marked by it is always performed. 3) Then, the type information of the state is checked. If the state itself is a "//-child" state, then its own stateID is added to the set, which effectively implements a self-loop marked by the '*' symbol in the NFA structure. 4) Finally, to perform an ϵ-transition, the hash table is checked for the 'ϵ' symbol, and if one is present, the //-child state indicated by the corresponding stateID is processed recursively, according to steps 1-3 above.

After all the currently active states have been checked in this manner, the set of "target states" is pushed onto the top of the run-time stack. They then become the "active" states for the next event. If a state in the target set is an accepting state, which means it has just been reached during reading the last element, the identifiers of all queries associated with the state are collected and added to an output data structure.

End of Element. When an end-of-element is encountered, backtracking is performed by simply popping the top set of states off the stack.

Finally, it is important to note that, unlike a traditional NFA, whose goal is to find one accepting state for an input, our NFA execution must continue until all potential accepting states have been reached. This is because we must find all queries that match the input document.

An example of this execution model is shown in Figure 3. On the left of the figure is the index created for the NFA of Figure 1. The number on the top-left of each hash table is a state ID and hash tables with a bold border represent accepting states. The right of the figure shows the evolution of the contents of the runtime stack as an example XML fragment is parsed. In the stack, each state is represented by its ID. An underlined ID indicates that the state is a //-child.

3.3 Predicate Evaluation

The discussion so far has focused on the structure matching aspects of YFilter. In an XPath expression, however, predicates can be applied to address properties of elements, such as their text data, their attributes and their position. We refer to these as *value-based* predicates. In addition, predicates may also include other path expressions, which are called nested paths. Any number of such predicates can be attached to a location step in a path expression. In this section, we briefly describe the techniques used in YFilter to support the evaluation of these predicates.

Given the NFA-based model for path-matching, an intuitive approach to supporting value-based predicates would be to extend the NFA by including additional transitions to states that represent the successful evaluation of the predicates. Unfortunately, such an approach could result in an explosion of the number of states in the NFA, and would destroy the sharing of path expressions, the primary advantage of using an NFA. Instead, in YFilter, we use a separate selection operator that evaluates value-based predicates by interacting with the NFA-based processing of path expressions. Traditional relational query processing uses the heuristic of pushing selections down in the query plan so that they are processed early in the evaluation. Following this intuition, we developed an approach called *Inline*, that processes value-based predicates as soon as the elements in path expressions that those predicates address are matched during structure matching. We also developed an alternative approach, called *Selection Postponed* (SP), that waits until an entire path expression is matched during structural matching, and at that point applies all the value-based predicates for the matched path.

Nested paths are handled by first decomposing queries into their constituent paths and then inserting all of these paths into the path matching engine. These paths are matched using the shared path matching approach described above. Then, we use a separate collection operator to process the matches of constituent paths and return the final query results. A more detailed description of these techniques is provided in the full paper on YFilter [5].

3.4 Overview of the Performance Results

We have performed a detailed performance study of our YFilter implementation [5]. In the study, we compared the performance of the NFA-based path matching approach in YFilter, the FSM-per-query approach used by XFilter, and a hybrid approach that exploits a reduced degree of shared path processing. We also investigated the tradeoffs between the Inline and SP approaches to value-based predicates. The results of the studies can be summarized as follows:

1. YFilter can provide order of magnitude performance improvements over both XFilter and the hybrid approach. In fact, as discussed earlier, path processing using YFilter is sufficiently fast that in many cases it is outweighed by other costs for XML filtering such as document parsing and result collection.

2. The NFA-based approach is robust and efficient under query workloads with varying proportions of “//” operators and ’*’ operators. This is important because it is these operators that introduce non-determinism into the path matching process. The NFA-based approach was also shown to perform well using a number of DTDs with different characteristics.

3. The maintenance cost (i.e., as queries are added and removed) of the NFA structure is small, due to the incremental construction that an nondeterministic version of a FSM enables and due to the sharing of structure inherent in the NFA approach.

4. For value-based predicates, the SP approach was found to perform much better than the Inline approach. The Inline approach suffers because early predicate evaluation cannot eliminate future work of structure matching or predicate evaluation, due to the shared nature of path matching and the effect of recursive elements in the presence of “//” operators in path queries. In contrast, SP uses path matching to prune the set of queries for which predicate evaluation needs to be considered, thus achieving a significant performance gain.

4 Related Work

A number of XML filtering systems have been developed to efficiently match XPath queries with streaming documents. XFilter [1] builds a Finite State Machine (FSM) for each path query and employs a query index on all the FSMs to process all queries simultaneously. XTrie [3] indexes sub-strings of path expressions that only contain parent-child operators, and shares the processing of the common sub-strings among queries using the index. In [8], all path expressions are combined into a single DFA, resulting in good performance but with significant limitations on the flexibility of the approach. YFilter and Index-Filter are compared through a detailed performance study in [2]. MatchMaker [10] is the only published work reporting its performance on shared tree pattern matching. Using disk-resident indexes on pattern nodes and path operators, it labels document nodes with all matching queries. I/O invocations limited its matching efficiency. Other related work includes publish/subscribe systems, such as Xlyeme [7] and Le Subscribe [11]. A common feature of these systems is the use of restricted profile languages, e.g. a set of attribute value pairs, and data structures tailored to them for high system throughput.

5 Conclusions

In this paper we have presented a technical overview of the basic structure and value-based matching approaches of YFilter with a focus on its novel NFA-based approach to shared processing of path expressions. Our work has shown that the NFA-based approach can provide high-performance XML filtering for large numbers of queries that contain both structure-based and value-based constraints.

More recently we have been extending YFilter in two important directions. First, we have investigated the use of YFilter in a more general XML Message Brokering scenario [6]. XML filtering represents the lowest level of functionality required for XML-based data exchange in a distributed infrastructure. In many emerging applications in this environment, however, XML data must also be transformed on a query-by-query basis, in order to provide customized data delivery and to enable cooperation among disparate, loosely coupled services and applications. To support such transformations, we take the NFA-based path matching engine as the basis, and develop alternative techniques that push the work of processing path expressions into the engine and perform efficient post-processing of the remaining portions of queries to generalize customized results. Second, as XML filtering systems are to be deployed in a distributed wide-area environment, our current efforts are aimed at studying the deployment of such systems as the foundation of an overlay network that supports content-based routing of documents and queries and intelligent delivery that allows shared transmission of query results.

6 Acknowledgments

We would like to thank Hao Zhang for the early discussion on the NFA-based path matching algorithm, Peter Fischer and Raymond To for helping us develop YFilter, and Philip Hwang for helping provide insight into XML parsing. We also owe thanks to Joseph M. Hellerstein, Samuel R. Madden, Sirish Chandrasekaran, Sailesh Krishnamurthy, and Ryan Huebsch for their valuable comments and feedback.

This work has been supported in part by the National Science Foundation under the ITR grants IIS0086057 and SI0122599 and by Boeing, IBM, Intel, Microsoft, Siemens, and the UC MICRO program.

References

[1] ALTINEL, M., AND FRANKLIN, M. J. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB Conference* (2000).

[2] Bruno, N., Gravano, L., Koudas, N., and Srivastrava, D. Navigation- vs. index-based XML multi-query processing. In *Proceedings of IEEE Conference on Data Engineering* (2003).

[3] Chan, C. Y., Felber, P., Garofalakis, M. N., and Rastogi, R. Efficient filtering of XML documents with XPath expressions. In *Proceedings of IEEE Conference on Data Engineering* (2002).

[4] Clark, J., and DeRose, S. XML path language XPath - version 1.0. `http://www.w3.org/TR/xpath`, November 1999.

[5] Diao, Y., Altinel, M., Franklin, M. J., Zhang, H., and Fischer, P. Path sharing and predicate evaluation for high-performance XML filtering. Submitted for publication. Available at `http://www.cs.berkeley.edu/~diaoyl/publications/yfilter-public.pdf`, 2002.

[6] Diao, Y., and Franklin, M. J. Query processing for high-volume XML message brokering. Tech. rep., University of California, Berkeley, 2003.

[7] Fabret, F., Jacobsen, H.-A., Llirbat, F., Pereira, J., Ross, K. A., and Shasha, D. Filtering algorithms and implementation for very fast publish/subscribe. In *Proceedings of ACM SIGMOD Conference* (2001).

[8] Green, T. J., Miklau, G., Onizuka, M., and Suciu, D. Processing XML streams with deterministic automata. In *Proceedings of IEEE Conference on Database Theory* (2003).

[9] Hopcroft, J. E., and Ullman, J. D. *Introduction to Automata Theory, Languages and Computation*. Addition-Wesley Pub. Co, 1979.

[10] Lakshmanan, L. V., and Parthasarathy, S. On efficient matching of streaming XML documents and queries. In *Proceedings of International Conference on Extending Database Technology* (2002).

[11] Nguyen, B., Abiteboul, S., Cobena, G., and Preda, M. Monitoring XML data on the web. In *Proceedings of ACM SIGMOD Conference* (2001).

[12] Watson, B. W. Practical optimization for automata. In *Proceedings of International Workshop on Implementing Automata* (1997).

The Virtues and Challenges of Ad Hoc + Streams Querying in Finance[*]

Alberto Lerner[†]
Ecole Nationale Superieure de Telecommunications
Paris - France
lerner@cs.nyu.edu

Dennis Shasha
New York University
New York - USA
shasha@cs.nyu.edu

Abstract

Financial trading strategies are based on queries over time-ordered data. The strategies value very recent data over older data, but require information about older data to avoid making poor decisions. One can imagine a streaming architecture that keeps a synopsis of older information available for many possible queries, but this may be too crude - and too expensive – an approximation of the query requirements. Instead, we show several fundamental query types for which traders would prefer to issue an ad hoc query Q and then allow updates to change the answer to Q over time. To make the ad hoc portion fast, the architecture puts historical data into a well-structured form (organized by security and time) to support rapid querying whereas recent data is ordered by time alone. Periodically, some of the older recent data is moved to the historical data structures. The net effect is to allow queries to look at very recent and less recent data efficiently and only when needed. Several new research issues arise in this setting.

1 Nature of Financial Data

Consider data coming from trading activities of equities (stocks), in particular electronic trading markets such as the NASDAQ. Such markets generate large volumes of data, in bursts that can achieve up to 4,200 messages per second [3]. Those messages present momentary opportunities to make profits if they can be processed in an appropriate way. Before explaining those strategies, let us consider a slightly simplified example of the mechanics of trading:

Bob decides to move some savings into company A's stocks. He calls his broker and places an order to buy 1000 of A's shares at a maximum price of $12.00. This is called a *bid price*. Bob's broker uses the stock market's trading system to broadcast Bob's bid. When it hits the market, the best offer for A's shares, called its *ask price*, is $12.03. Alice wants to sell some of her shares of A. She instructs her broker to sell 1000 of her A's shares at market value. When her offer meets Bob's bid, a trade is done. Each bid, ask, and trade is called a "tick." There

[*]Work supported in part by U.S. NSF grants IIS-9988636 and N2010-0115586.
[†]This work was done while this author was visiting NYU

are up to about 100 million ticks per six-and-a-half-hour day. The time series in this context involve prices and volumes (number of shares) of bids, asks, and trades.

These series can be obtained by a subscriber in distinct levels of detail [6]. Strategies use each level of detail differently. For instance, applications called *Level I* are real-time displays of best (maximum) bids, best (minimum) asks, and the last trade price. *Level II* displays show more in-depth information by presenting the top 5 best levels of bids and asks with their respective market participants. Such information is useful because it can indicate the level and depth of supply and demand.

Those "displays" present real-time summaries of trading activities. Alarms (continuous queries) can be set to detect very simple conditions, such as when a price is the maximum price of the day. We show in the next section that the technical analysis of the finance streams involves more complex queries in addition to these simple alarms.

2 Technical Analysis Strategies

Technical analysis is the activity of making buy/sell decisions based on the time course of a stock, perhaps with respect to other stocks. Here we give a few techniques used by traders [5].

2.1 Intraday

Intraday trading attempts to uncover momentary opportunities (at most a few minutes old) to make small profits. Here are some example opportunities:

"Scalper" trading tries to find a tight interval within which the ask and bid prices are currently oscillating and buys or sells at mid points. For instance, a "scalper" would try to buy A's share if it were quoted at $11.98, before Bob's had the chance to hit the market, if such an ask price were announced. If the "scalper" were fast enough, he could have even sold these shares to Bob. A human scalper can do a few hundreds of these operations in a day.

Pairs-trading tries to take advantage of shares that are usually impacted by the same effects. Simplifying to the essence of the idea, if two stocks tend to differ by no more than $20 and the higher one goes up whereas the other goes down, then sell the first and buy the second. Sell out when the difference returns to normal.

Note that in the above cases, very recent history (i.e., a few seconds to a few minutes old) is vastly more important than older history. In the next example not-so-recent data is involved.

"Hammer" discovery. Market Makers are large holders of shares and are thus capable of buying or selling heavily. Whenever they do so they can exercise some control (their "hammer") over the quotes of a given security. To avoid calling attention to themselves, they partition the volumes they are moving into a number of small trades to avoid paying more when buying or receiving less when selling. Savvy investors try to infer such moves by trying to identify any hammer-type movement – which can be diluted over a period ranging from minutes to hours – and profit from it by bidding against the hammers (e.g., buying when the hammers are buying).

2.2 Long-Term

At the other end of the continuum of trading strategies come those that consider long periods of price series. Here are a few examples:

Crossing averages. Moving averages are capable of smoothing the volatile price curves and exposing underlying optimistic or pessimistic sentiment. For example, whenever a short term trend curve (a moving average over a few days) climbs above the long term one (a few weeks or months) one, technical analysts will suspect that the stock will move up soon.

Breakout in support-resistance curves. Some view a trading market as a continuous fight between buyers and sellers, and attribute to them the power to control the range within which prices oscillate. The basic idea is that buyers will predominate when the price is cheap enough (at or below its “support level”) and sellers will predominate when the price rises above its “resistance level.” Occasionally, of course, the price exceeds this range, an event that may be of considerable interest to a trader.

2.3 Characterizing Queries in Technical Analysis

Underlying each trading strategy are queries over the time series. Long-term analysis have little need for intraday trading data. A salient aspect about these queries is that fact that they often depend on order (e.g., moving averages, or deltas of prices on time-ordered series). Therefore, long-term queries can be implemented using order-dependent query languages over static data. Our language AQuery [1] is an example of such a language.

In turn, intraday data requires several kinds of queries:

- Continuous: every time an element of data such as a quote or a trade arrives, the query has to be re-evaluated. Typical applications: scalper and pairs trading.
- Periodic queries: Continuous queries over fast-paced streams may overflow an analyst with results that may not be needed as soon as they are processed. Periodic queries allow data to accumulate for a fixed amount of time and then issue the query up to the latest time point. A typical application is a query looking for hammer activity in a given stock. Periodically, perhaps on demand, we want to see if the hammer is active.
- Ad-hoc: just prior to doing a trade, we may want some longer term analysis. For example, we may have heard rumors of a hammer and then verify it.

3 A Detailed Query Example

Suppose a trader believes there might be a Hammer trying to acquire shares of ACME, but is not yet sure. He/she issues an ad hoc query to find out. The query essentially asks whether any market maker has many *inside bids* in the recent past. An inside bid is the highest bid at a given time.

If the query were on purely time ordered data, the ACME data would be mixed in with 10,000 other stocks. This would require a scan of all that data. The query would go much faster, however, if ticks were organized by stock and then by timestamp within each stock. Such data reorganization is the key to answer our query within a reasonable amount of time. However, it may not be feasible to do such a reorganization in real-time. Let us see how a periodic reorganization can be done instead.

Let the ticks’ schema be Ticks(sID, MMID, price, volume, type, timestamp) where sID is a security’s identifier; MMID, the market maker behind this tick; type is either ’bid’, ’ask’, or ’trade’; price and volume are associated with this tick; and timestamp is the precise instant the tick enters the trading system. We will be using AQuery, a dialect of SQL that incorporates the notion of order, to show the queries discussed here. We will be explaining its features at they appear. A more complete reference about that language (its underlying data model and its optimization) can be found at [1]. AQuery’s formulation of the hammer discovery query looks like:

```
WITH
   MaxPrice(insideBidPrice, timestamp) AS
   (SELECT  maxw( range(90,timestamp) , price ), timestamp
   FROM     Ticks
            ASSUMING ORDER timestamp
   WHERE    timestamp > now() - 30 minutes
            AND type = 'bid'
            AND sID = 'ACME')
SELECT  MMID, count(*)
FROM    Ticks t, MaxPrice mp
WHERE   t.timestamp = mp.timestamp
        AND t.timestamp > now() - 30 minutes
        AND type = 'bid'
        AND sID = 'ACME'
        AND price = insideBidPrice
GROUP   BY MMID
```

Figure 1: The “Hammer Discovery” query.

The first half of the query computes ACME’s inside bid prices for the last 30 minutes, considering that a bid has a 90 seconds lifespan[1]. The WITH is a construct borrowed from SQL:1999 that allows a query to define a “local” view [2]. The ASSUMING ORDER clause is AQuery’s and is used to enforce a given sort order after the FROM clause is processed. Thus, in the first half of the query, Ticks can be assumed to be in timestamp order. All subsequent clauses in the WITH query can count on and preserve that ordering. AQuery also provides convenient ways of manipulating date and time data, e.g., the function now() that returns the system’s current timestamp, or the literal ‘30 minutes’ as a way to express a time value. Finally, AQuery’s semantics is column-oriented, meaning that instead of having variables that range over collections, an AQuery variable is bound to an entire collection – an array to be more precise – at once. For instance, the function ‘range(90, timestamp)’ is called only once and it is passed the entire column timestamp as its second argument. Range takes a value v and an ordered vector c whose element type is the same as v’s, and returns a vector range$_{v,c}[i] = w[i]$ where $w[i]$ is the minimum index such that $c[i] - c[i - w[i]] <= v$. In our case, range() returns for each timestamp t, the number of previous ticks the window should contain if one wanted that window to span from t - 90 seconds up to t [2]. The function ‘maxw()’ takes a vector w of window ranges and a vector c and computes maxw$_{w,c}[i] = max(c[i - w[i]]..c[i])$. The main query identifies the Market Makers that matched any inside bid at any given point and counts how many times that happened for each distinct Market Maker.

Now, if one issues the hammer discovery query late in the day, available ticks over the last several hours should be looked at. To make that task efficient, we keep recent and historical ticks in different structures. We describe those structures as (maybe materialized) views over the stream. Recent data is maintained in a “RECENT” portion of Ticks, which is defined as a view over Ticks, as seen in Figure 2.

Data arriving at the system is immediately available at the RECENT view. But the latter necessarily have an “aging” predicate that determines that data must be purged when the aging predicate is false. In TicksRecent, this predicate is ‘timestamp => now() - 5 minutes.’ Aged RECENT data is moved to HISTORICAL views. See Figure 3. Moving the data efficiently is an implementation issue in its own right – when to do it, how to maintain data while it’s being done and so on. We return to this issue later.

Note that RECENT and HISTORICAL views may organize data differently. In this particular case, they differ in the way they sort it: whereas recent Ticks are sorted by timestamp, older ones are organized by security ID and sorted by timestamp for each such ID. Intuitively, to answer the hammer search query, the system would

[1]A bid’s lifespan can vary depending on several factors, including which system was used to place it and even which time of the day it was placed [4], so this is a slight simplification.

[2]Windows in AQuery are mere aggregate function arguments. Some other languages handle them as special language constructs, confined to specific parts of a query (e.g., valid only in the SELECT clause). We argue in [1] for the benefits of the former over the latter.

```
CREATE RECENT VIEW TicksRecent AS
SELECT sID, MMID, price, volume, type,
       timestamp
FROM   Ticks
       ASSUMING ORDER timestamp
WHERE  timestamp => now() - 5 minutes
```

Figure 2: Ticks' RECENT View

```
CREATE HISTORICAL VIEW TicksHistorical AS
SELECT sID, MMID, price, volume, type,
       timestamp
FROM   Ticks
       ASSUMING ORDER sID, timestamp
WHERE  date(timestamp) = today()
       AND  timestamp < now() - 5 minutes
```

Figure 3: Ticks' HISTORICAL View

break that query into two parts. While the recent part of it would have to select out 'ACME' ticks among all others, the historical part would be able to select a contiguous extent of 'ACME' ticks. Because data in the historical part is more voluminous, it is important that such a query not require a scan of it all. Figure 4 shows possible query plans for the situation above.

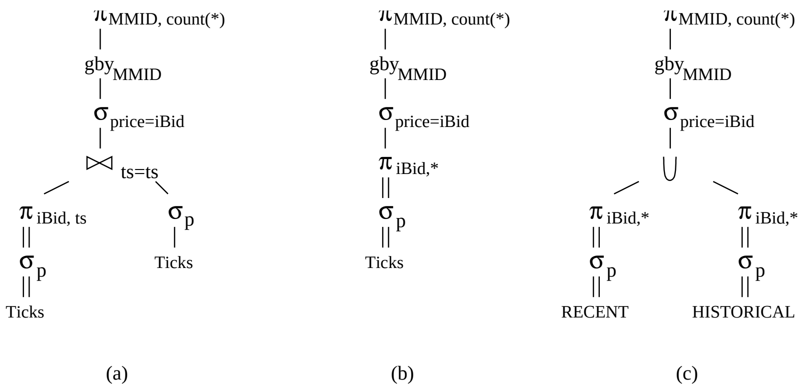


Figure 4: Plans for the Hammer Discovery Query: (a) Initial Plan (b) Optimized Plan over Stream Data (c) Optimized Plan over RECENT + HISTORICAL Data

The plan in Figure 4(a) follows the query's syntax. We abbreviate `date(Ticks.timestamp)=today() AND type='bid' AND sID='ACME'` for p, and `maxw( range( 90, timestamp ), price)`, the inside bid price, for $iBid$. Note that some operators in the plan are connected by double arcs. That signifies that existing order is being maintained by operators. For instance, in the left-side branch of the join, the existing order of Ticks (over timestamp) is preserved. That is necessary, because the calculation of iBid requires data to be in timestamp order. After that computation is done the ordering requirement is drop.

The plan can be simplified. The left-hand side of the join is simply computing a new column, iBid. The join can thus be exchanged for a projection that adds that computed column to the ordered table. The result is depicted in Figure 4(b).

The performance of that plan grows slower as the stream it is acting upon becomes more dense over time (i.e., more ticks per second). The top curve on the graph presented in Figure 5 shows that progression. We are simulating a one-hour-long stream with varying number of ticks per second. The query is required to look at the last 30 minutes ticks only. But, as said before, ACME ticks are mixed within thousands of others. The cost is dominated by the evaluation of predicate p. The reorganization of aged data clearly pays off as the second curve on the graph suggests. But before commenting on those results, let us show how a more elaborate plan can take advantage of the RECENT and HISTORICAL data organizations.

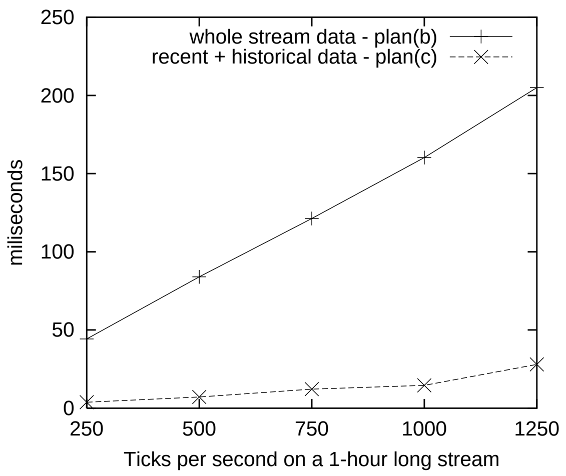


Figure 5: Performance Comparison between Figure 4's plans

The union of RECENT and HISTORICAL ticks contains all the ticks in the original stream. Thus, one can decompose the original query to use the (materialized) views described before. In our present example, it suffices to compute iBid over both the views, take the union of the results and then select which Market Makers matched the inside bids and count how many times per Market Maker. Such a plan is depicted in Figure 4(c).

Such an elaborate plan yields much better performance. Let's see why. The predicate p should be evaluated over both RECENT and HISTORICAL data. RECENT has the same organization as the original stream and so it is expensive to evaluate p over it. But recall that RECENT doesn't keep all the data. Here the aging predicate is discarding data that is 5 minutes old. As a result, there are far fewer rows to look at. Now, those rows go to HISTORICAL when they age but are ordered there by sID and timestamp. Therefore, the conjuncts of p that find ACME's ticks and the desired timespan of ticks can be computed with faster binary searches. That alleviates most of cost of computing p, as those conjuncts are quite selective. The remaining part of the query concatenates the results, computes inside bids and so on, as before. The result: not only does the plan (c) in Figure 4 have a much lower cost than plan (b) – one order of magnitude – but also its time increases slower as a function of stream density.

Of course, this organization has a price, but if the query density is high enough (or the importance of the queries is high enough), it is worthwhile.

This data reorganization rationale can be generalized as we discuss next.

4 A Generalized Streaming Architecture

We now describe an architecture that allows the ad hoc queries to be processed using the two data organizations based on the modules of Figure 6.

Rows arrive as a set of streams. The *dynamic router* taps into those streams and extracts rows' elements that are to be inserted into RECENT views. As its name suggests, the dynamic router is capable of directing a row to its respective RECENT destination. This module and all the remaining ones depicted in Figure 6 have access to metadata (catalog) information.

Eventually rows at RECENT views become aged. Whenever that happens – or at strategic moments (e.g., low system workload) – the *transfer agent* is responsible for purging them from RECENT and inserting them into HISTORICAL views. To avoid concurrent contention, some data may be redundant in the two tables for a time. In that ways, queries will always find all the necessary data.

Ad hoc queries sent to the system are handled by the *ad hoc query processor*. This module is responsible

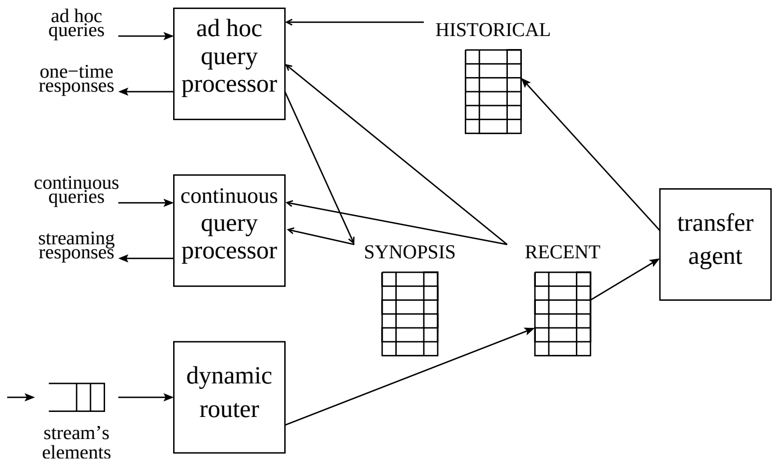


Figure 6: A Generalized Architecture for Ad-Hoc and Continuous Queries

for breaking down an ad hoc query, which is expressed against an integration schema, into component queries that can be executed against RECENT and HISTORICAL data. It then creates a synopsis data structure and passes the query to the *continuous query processor* so the answer to the query can be maintained as updates arrive. When the query ceases to be of interest, then the continuous query processor must be informed and stop processing updates.

5 Summary and Open Problems

Streaming is often viewed as a technology in which one has only one chance to look at data, and limited memory. Thus viewed, streaming is a useful technology for financial databases. Of the kinds of technical queries we've identified, scalping, and breakout are essentially stream queries.

On the other hand, some queries must be done retrospectively after a short term pattern (e.g. of a potential hammer) suggests looking at historical data. Because the data required by such queries is ideally ordered by some other attribute as well as by time, some preprocessing is desirable on most if not all the data. For this reason, we propose a generalized streaming architecture that supports efficient ad hoc queries, but allows classical streaming on recent data. Its main characteristics are:

1. splitting recent data from better-indexed historical data, where the indexing includes data ordering;
2. streaming techniques for the recent data including running synopses to support the maintenance of query responses over time;
3. order-aware query optimization for both recent and historical data.

Such an architecture presents two major challenges that we are actively studying:

- Maintaining the historical data in this useful organization without interrupting concurrent queries requires the avoidance of locks and perhaps the postponement of updates.
- Handing off the processing of continuous queries from the ad hoc query processor to the continuous query processor.

These challenges are real, but far from insurmountable. We foresee the deployment of high performance, order-aware, generalized streaming systems to a trading desk near you in the not-too-distant future.

References

[1] Lerner, A., and Shasha, D., “AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments.” TR2003-836, Courant Institute of Mathematical Sciences, New York University, March 2003.

[2] Melton, J., and Simon, A.R. “SQL:1999 – Understanding Relational Language Components.” Morgan Kaufmann, May, 2001.

[3] NASDAQ. “Peak Message Rates fro NASDAQ SuperMontage Data Feeds in December 2002.” Nasdaq Vendor Alert #2003-4, January 17, 2003.

[4] NASDAQ. “SuperSOES Frequently Asked Questions.” http://www.nasdaqtrader.com

[5] NASDAQ. “ViewSuite Data and NASDAQ SuperMontage: PowerView.” http://viewsuite.nasdaqtrader.com/resources/Level2_pres.ppt.

[6] NASDAQ. “Nasdaq Data Feed Products.” http://www.nasdaqtrader.com/trader/mds/nasdaqfeeds/feeds.stm

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903