Report Number: WUCS-96-27

1996-01-01

Distributed Stream Filtering for Database Applications

William M. Shapiro and Kenneth J. Goldman

Distributed stream filtering is a mechanism for implementing a new class of real-time applications with distributed processing requirements. These applications require scalable architectures to support the efficient processing and multiplexing of large volumes of continuously generated data. This paper provides an overview of a stream-oriented model for database query processing and presents a supporting implementation. To facilitate distributed stream filtering, we introduce several new query processing operations, including pipelined filtering that efficiently joins and eliminates duplicates from database streams and a new join method, the progressive join, that joins streams of tuples. Finally, recognizing that the stream-oriented model results... **Read complete abstract on page 2.**

Distributed Stream Filtering for Database Applications

William M. Shapiro and Kenneth J. Goldman

Complete Abstract:

Distributed stream filtering is a mechanism for implementing a new class of real-time applications with distributed processing requirements. These applications require scalable architectures to support the efficient processing and multiplexing of large volumes of continuously generated data. This paper provides an overview of a stream-oriented model for database query processing and presents a supporting implementation. To facilitate distributed stream filtering, we introduce several new query processing operations, including pipelined filtering that efficiently joins and eliminates duplicates from database streams and a new join method, the progressive join, that joins streams of tuples. Finally, recognizing that the stream-oriented model results in performance tradeoffs that differ significantly from those in traditional databases, we present a new query optimization strategy specifically designed for stream-oriented databases.

Distributed Stream Filtering for Database Applications

William M. Shapiro and Kenneth J. Goldman

WUCS-96-27

October 1996

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Distributed Stream Filtering For Database Applications

William M. Shapiro
wms1@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130-4899

Kenneth J. Goldman
kjg@cs.wustl.edu
Department of Computer Science
Washington University
St. Louis, MO 63130-4899

Abstract

Distributed stream filtering is a mechanism for implementing a new class of real-time applications with distributed processing requirements. These applications require scalable architectures to support the efficient processing and multiplexing of large volumes of continuously generated data.

This paper provides an overview of a stream-oriented model for database query processing and presents a supporting implementation. To facilitate distributed stream filtering, we introduce several new query processing operations, including pipelined filtering that efficiently joins and eliminates duplicates from database streams and a new join method, the progressive join, that joins streams of tuples. Finally, recognizing that the stream-oriented model results in performance tradeoffs that differ significantly from those in traditional databases, we present a new query optimization strategy specifically designed for stream-oriented databases.

1. Introduction

A database is traditionally viewed as a set of data stored at one or more locations. The database is passive, responding to specific queries and updates invoked by the environment. Distributed databases provide the same functionality as the traditional centralized database, but use replication to provide greater reliability and availability without sacrificing database consistency.

A class of distributed database applications has emerged that utilize databases, but these databases are not stored; rather, they are generated streams of data. These applications, which we call *High Volume Data Stream* (HVDS) applications, include real-time stock market analysis systems, next generation weather processing systems[17], wire and news services that generate large volumes of information[14], satellite imaging systems and multimedia database applications[1][7].

A particular example of a HVDS application is the Advanced Weather Interactive Processing System (AWIPS) currently being developed by the National Weather Service[17]. One of the major goals of AWIPS is to provide more timely weather forecasts and warnings. To accomplish this goal, each AWIPS site will use models specific to its location to provide real-time local analysis and forecasting based on the data received from both nearby and distant data sources. AWIPS will consist of over one-hundred information processing sites that are capable of receiving and processing real-time and frequently updated climate data from doppler radars, satellites, surface weather observing stations, and a variety of other data sources. Each of these sources may, in turn, consist of up to several hundred installations that collect and transmit data.

The distinguishing characteristics of a HVDS application are:

- large volumes of continuously updated data from multiple sources,
- multiple recipients of data subsets,
- distributed data processing requirements, and
- higher importance placed on recency of data than database consistency.

For some applications, receiving the newest data possible is more important than receiving a consistent picture of a database at any one moment in time. In the case of a stock market analysis system, knowledge of specific events occurring (e.g. a change in a stock price) may take precedence over knowledge of the prices for all stocks at a moment in time.

Additionally, HVDS applications will often be used on networks with very high communication speeds at the center and much lower speeds at the edges. With the deployment of gigabit speed networks in the next few years, communication costs are expected to fall dramatically at the backbone of networks[2][20][21], yet network speeds at the edges of networks are not expected to increase significantly[13]. Therefore, large amounts of data can be sent across the

high-speed portion of the network, but the quantity of data must be reduced significantly before it is sent to the edges of the network.

This paper introduces *Distributed Stream Filtering* (DSF) as a mechanism for implementing HVDS applications taking advantage of gigabit speed networks, while recognizing the limitations on bandwidth at the edges of the network.

The paper is organized as follows. Section 2 discusses related work both inside and outside of the database field. Section 3 explains and justifies the major assumptions made in this paper. Sections 4 and 5 introduce the DSF model and the underlying system. Section 6 discusses implementation of this system for relational databases. Section 7 introduces a new query optimization strategy tailored for DSF. Section 8 discusses other possible applications of DSF. Finally, Section 9 describes the status of our DSF implementation.

2. Related Work

Distributed stream filtering builds upon work in traditional database systems[22], and relational databases[3][4], but is most directly related to distributed databases[18], event filtering[15][16][19][25], and active databases[8][23][24]. In this section we cover the relevant work in these areas and outline the properties of DSF that differentiate it from conventional techniques.

A distributed database[18] is a collection of logically connected databases distributed throughout a network. There are several advantages of distributed databases over centralized databases. Distributed databases allow greater local access to and control over data because each locality can store the fragment of a database that it commonly uses. Distributed databases also provide greater fault tolerance as each site across which the database is fragmented is independent. Finally, a distributed database provides greater scalability as the distributed database management system can use resources distributed throughout the network, reducing the resource demands at any one location.

Much of the research relevant to distributed stream filtering is in the area of event filtering[15][16][19][25]. An event is a significant occurrence in a system. Examples of events include buffer overflows in a network, a change in the core temperature of a nuclear reactor and a change in a stock price. Event filtering is concerned with the automated handling of such events. An overview of event filtering is provided in [19]. Below we briefly discuss the major work in this area.

Packet filters[15][16][25], one form of event filtering, were developed to support event demultiplexing of network packets. A packet filter resides on the users system and filters incoming network packets based on protocol as they pass through the network interface of an operating system kernel[25]. Recent packet filters, such as the Mach Packet Filter[25], allow composition of multiple filters to reduce the demultiplexing overhead.

Active databases[8][23][24] grew out of database constraints which are used to specify that a database guarantees certain properties, such as specifying that all keys are unique. Active databases generalize and extend constraints through the use of triggers. A trigger is a mechanism that is invoked in response to specified events. Examples of events in active databases include the deletion of data, surpassing a database size limit and insertion of a value of a specified size. When an event occurs, an action is triggered which could be as simple as creating a log entry or as complicated as performing a predefined computation and multicasting the result to various users. Therefore, an active database is a conventional database with added functionality.

In traditional query processing, users submit queries to a specific database (which could be distributed across the network) and the database system processes the query and returns the result as depicted in Figure 1. To provide greater availability, the database can be mirrored at a number of sites that process queries. However, if the database is frequently updated, each mirror must be updated, and the updates must be carefully synchronized, or multiple copies must be stored to guarantee that only a consistent database is queried (as opposed to a partially updated database). Additionally, this approach does not take advantage of commonality in user queries. Rather, the answer to a query must be recomputed for each user that poses the same query. Finally, this approach does not scale well with increasing query frequency. In time-critical applications, users may require information from multiple databases every second or fraction of a second.

Event filtering has begun to provide the active behavior that is necessary for real-time applications. However, current event filtering mechanisms are intended to provide destinational filtering, in the case of packet filters, and local filtering at a particular database, in the case of active databases. Current event filtering mechanisms are not suited for filtering data that is distributed throughout a network.

3. Assumptions

In this paper we makes several assumptions about the feasibility of DSF. This section introduces the existence of mechanisms that justify these assumptions. Specifically, we assume the existence of:

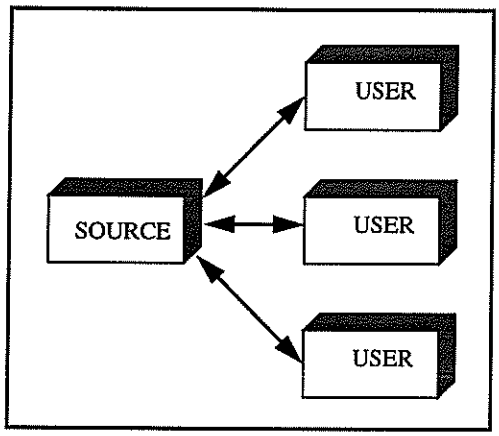

Figure 1: Conventional query processing model

- A module based programming environment in which components can be added and removed dynamically,
- high bandwidth low latency end-to-end communication within the "core" of the network and,
- scalable reliable multicast.

The first assumption is necessary to be able to dynamically add and remove filter modules in response to new user queries and the termination of existing user queries. One such programming model is *The Programmers' Playground* [9][10][11][12].

Playground is a software library and run-time system that supports a simple programming model for distributed applications. The model, called I/O abstraction, provides a separation of computation from communication that is well-suited for end-user construction of customized distributed applications from computational building blocks. Playground users do not need to write any source code to establish communication between the modules of a distributed application, nor do they need to understand the details of how communication occurs.

In the I/O abstraction model, each module in a distributed system has a *presentation* containing published variables that may be externally observed and/or manipulated. Modules are written in a standard programming language (e.g., C++) using the Playground library. The Playground library provides a set of publishable data types. These include base types (e.g., integer, real, string), aggregate types (e.g., arrays, mappings), and tuples. Programmers may arbitrarily nest these types to form new publishable data types and new publishable aggregates.

A distributed application consists of a collection of independent modules and a configuration of *logical connections* among the published variables in the module data boundaries. Whenever a module updates one of its own published data items, the new value is implicitly communicated to all connected variables in other modules. Input is observed passively, or handled by reactive control within a module. The details of how the communication is handled are hidden from the implementor and users of the module. This simplifies module construction and gives the run-time system flexibility in optimizing communication. The configuration of connections is determined dynamically at run-time, rather than statically at compile time. This gives users the flexibility to add new components or relationships to their applications dynamically.

The second and third assumptions regarding communications mechanisms are necessary to stream data over the network at a high rate in a DSF system. To provide these communications mechanisms, we are currently implementing a high performance version of the Programmers' Playground that takes advantage of a new Asynchronous Transfer Mode (ATM) Port Interconnect Controller (APIC)[5] and a new gigabit ATM switch that supports scalable reliable multicast[2][20][21]. The APIC chip will allow us to achieve application-to-application communication performance approaching raw network data rates by avoiding operating system and protocol stack overhead. Data written to the presentation by the sending module is picked up from memory by the APIC and sent out directly over an ATM virtual circuit. The APIC at the receiving end places the incoming data directly into the address space of the destination module. Protection is set up in advance by the operating system and enforced by the APIC hardware. This mechanism, in conjunction with support for reliable multicast, will enable efficient realization of the distributed stream filtering mechanism presented in this paper as support for HVDS applications.

4. The Distributed Stream Filtering Model

In the distributed stream filtering model, we view a database as a generated stream of data, as opposed to a passive data store. Databases are streamed out over the network and pushed through filters that process data to answer end-user queries. The end-user does not query a central facility for a "one-time" response, but instead connects to the stream of data that is the continually updated response to the query.

Figure 2 shows the basic components of a DSF system. A typical system includes multiple data sources, a filtering system composed of many filters distributed throughout the network, and multiple users of the fil-

tered data. Filters can be dynamically added and removed as new queries are initiated and existing queries are terminated. A DSF system provides much greater scalability because central locations are not overwhelmed with queries. Rather, the load can be distributed throughout the system based on available resources.

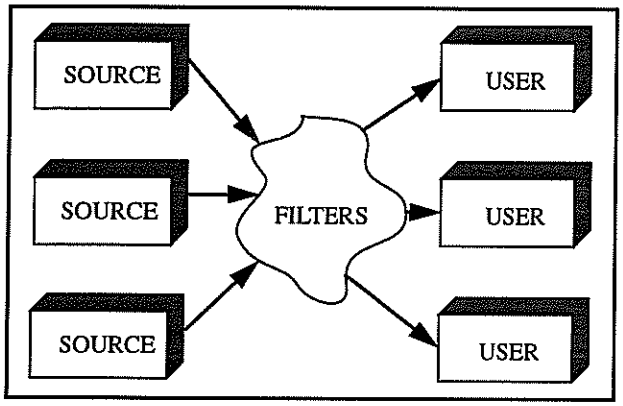

Figure 2: Distributed stream filtering model

Another major advantage of the DSF model is that it supports the reuse of query and subquery results. For example, in a stock market analysis system, it is likely that many users will request information about the most active stocks on a given day. This information can be computed once and multicast to all users who request it instead of being computed separately for each user. Similarly, common subqueries can be computed once and multicast to other filters that perform additional computations.

5. Overview of the DSF System

In a distributed stream filtering system, data is streamed from the data source(s) and filtered at one or more levels and locations to answer queries. At the heart of the DSF system are *control modules* that manage and control access to a subset of filters. Each control module maintains a list of all resources to which it has access. Examples of resources include a dynamically changing collection of filters and data sources.

The user submits a query to an assigned control module, which may or may not be local. The user expects a stream representing the answer to the query in return. It is then the job of the control module to find the resources specified in the query and configure a set of filters to answer the query if it is not already being answered.

To answer queries, control modules must locate non-local resources in the system. To facilitate this, we build on top of the distributed stream filter mechanism itself. Each control module streams out a list of the resources it "knows about." Control modules can be configured in a hierarchy so that all resource information filters up to the top of the hierarchy. In this way, the list of resources streamed out from the module at the top of the hierarchy will contain all available resources in the system. Having each control module stream out a list of available resources provides much greater scalability than the alternative of using remote procedure calls (RPCs) to locate resources as each control module can multicast this information to an arbitrary number of locations. The solid grey arrows in Figure 3 denote this stream of data.

Unlike filter modules, control modules are statically placed throughout the network. They are located at each data source and wherever databases could be joined or merged in order to provide access to both databases and combined databases. Placement of control modules is not addressed in this paper, but we assume that they are strategically placed for availability and performance.

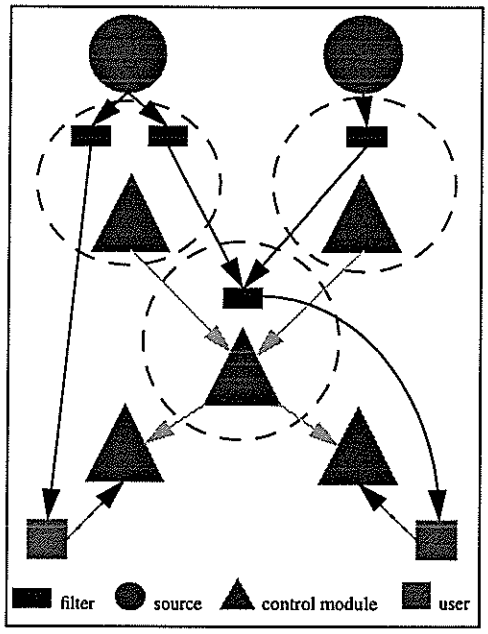

Figure 3: A distributed stream filtering system

The main function of the control module is to arrange filters to process user queries. When a user submits a query (dashed line in Figure 3), a control module determines which parts of the query are already being answered from the data stream containing the list of available resources. It then optimizes the query (see below) taking into account which subquery results already exist. Finally, the user's control module makes

requests to the appropriate control modules to create and connect new filters.

6. Implementing Filtering Operations for Relational Databases

Although filters for a DSF system could be written to perform virtually any filtering operation, we focus here on the filtering operations that implement the basic operators on relational databases, since these filters can be constructed automatically in response to a traditional database query.

It is important to remember that the data to be filtered is a continuous stream of tuples as opposed to a snapshot of a database at a particular moment. For instance, in many applications some tuples in a stream may have been updated more recently than others, so there is no notion of the database at any one moment in time. Rather, the stream represents the most current information available. Additionally, to preserve the streamed nature of the database as it flows through filters, all operations are performed as each tuple passes through a filter and little or no state information is stored.

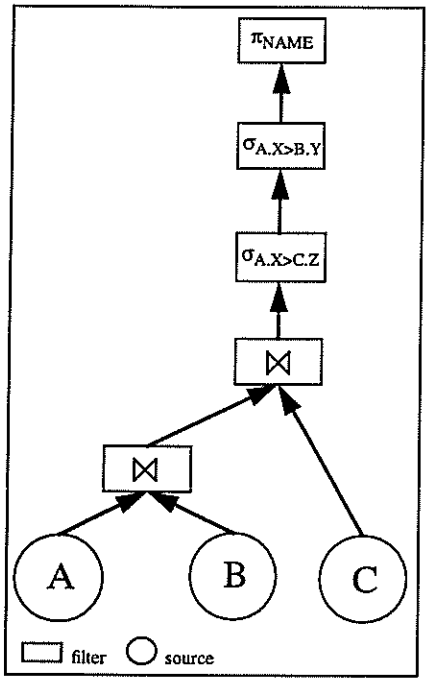

Figure 4: Example filter configuration

Consider the following Structured Query Language (SQL) query:

```
SELECT A.W
FROM A, B, C
WHERE A.X > B.Y
AND A.X > C.Z
```

A (possibly naive) filter configuration to filter the above query is presented in Figure 4.

6.1 Select (σ)

As each tuple passes through a select filter, the select operation is performed on that tuple and the tuple is output if it meets the select condition. Furthermore, since we are calculating whether each tuple meets or does not meet a given condition we can output those tuples that don't meet the given condition as a separate stream. In this way, we need not duplicate the work already performed to answer a query that requires the opposite select operation.

Performing a select on a tuple can be done in constant time as it does not depend on the size of the database. In fact, for the select operation there is no concept of a database of size N as the select is performed on a stream of data that is considered infinite.

6.2 Project(π)

Performing a project without duplicate elimination on a tuple could also be done in constant time. However, performing a project with duplicate elimination not only has at minimum linear time complexity, but also complicates our simple model. Until now we have assumed that no data is stored in a filter. A filter simply processes each tuple as it is received and sends the result on its way. But to perform duplicate elimination, a filter must store state information, the database in this case. We introduce two possible solutions to this problem below.

6.2.1 Time Stamping

One possible solution to the storage problem is to time-stamp each tuple, in effect making each one unique. Time-stamping allows the end-user to choose the appropriate tuple when all other fields are identical. However, time-stamping also increases the amount of data that must be transmitted and may not provide the desired semantics for a particular application.

6.2.2 Pipelined Filtering

Duplicate elimination can also be performed by using a method we call *pipelined filtering*. Pipelined filtering involves sending the same database stream through a sequence of one or more filters (a pipeline) from both directions as shown in Figure 5. We are guaranteed that each tuple coming from the left side will

pass each tuple coming from the right side. One stream is a reference stream, used only to detect duplicates. Duplicates are eliminated from the other stream, which is the result of projection with duplicate elimination upon completion. As tuples pass each other we can compare each pair and eliminate any duplicates.

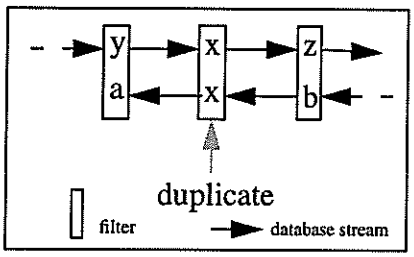

Figure 5: Using pipelined filtering to perform duplicate elimination

There are several minor issues that must be dealt with in order to perform pipelined filtering. Assume we are filtering a stream that contains the sequence of tuples: 1 2 3 4 5. We are guaranteed that each tuple passing through the pipeline from the left side will pass an identical tuple coming from the right side because the two streams are identical. The 5 coming from the left side will pass by the 5 coming from the right side, and so on. We do not want to eliminate these "duplicates" because they are not duplicates at all. To solve this problem, we can add an extra bit to each tuple in the reference stream as it enters the pipeline that stores a flag denoting whether it has passed a duplicate yet. The first time a tuple passes a duplicate, the flag in the reference stream tuple is set, the second time the tuple in the other stream is eliminated.

The second issue that must be resolved is that a database stream is continuous, and therefore has no beginning or end. However, we only want to eliminate duplicates in each instance of a database that passes by. The simple solution is to use a flag to logically separate different instances of a database.

The potential drawback to using pipelined filtering is that its storage requirements could be quite large. The pipeline must store up to two copies of a database and they must be stored in main memory as the I/O costs associated with external storage in a filter are prohibitive. For small databases, this would not be a problem, but, for very large databases, the amount of memory required may be unreasonable. The issue is the standard space-time tradeoff, where a gain in database stream throughput is achieved through additional memory units in the pipeline.

6.3 Join(⋈)

The join operation, like the project operation with duplicate elimination, requires state information about both databases involved in the join. However, storing and retrieving databases runs contrary to the motivation behind DSF. We introduce a new join method, the *progressive join*, as a stream-oriented join of two databases. The progressive join provides the following logical view of the joining of two data streams. Each tuple of one database, as it enters the join filter, is joined with the other database, and vice versa. In this way, we preserve the stream property of the database. Although not necessarily the most efficient implementation, at a logical level we can think of a join filter as containing instances of both databases that are being updated by the two database streams. As a tuple enters the filter, it is joined with each tuple from the other database that meets the join condition regardless of how recently it has been updated. For example, the first half of one database may have been updated more recently than the second half.

The major question that still remains concerns efficiency. It is necessary to access at least part of one database in order to join it with individual tuples from the other database. This seems to imply that one or both databases must be stored. We discuss two possible solutions to this problem.

6.3.1 Rapid Refresh Rates

One solution is to have one or both of the database streams to be joined refreshed at a very high rate. The join filter can then grab pieces of each stream and join those pieces. If the refresh rate is sufficiently fast, grabbing tuples from a stream can be considered analogous to performing a page fault to retrieve the information, but without external storage and the associated I/O costs.

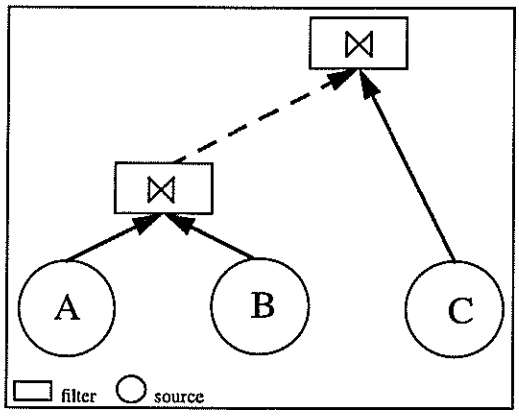

Figure 6: Multi-level join

There are several drawbacks to using rapid refresh rates. Possibly the greatest potential drawback is that

rapid refresh rates limit scalability. Consider a database that is joined with another database that is itself the result of a join operation as in Figure 7. The refresh rate of the second database is bounded by the speed at which its join is computed. It is easy to see that after a few levels of joins, the refresh rate would become intolerably slow. One way to increase the refresh rates of databases that are processor bound is through the use of strategically placed storage modules. A storage module would be placed after a join filter and would store a single copy of the database. The storage module could then repeatedly stream out the database at a faster rate than it was coming in from the join filter.

Another drawback to using rapid refresh rates is that it would not result in the join of two "consistent" databases. Rather, the join is performed on fragments of each database, some of which would be more recent than others. Therefore, tuples could be deleted or added in the middle of a join such that only parts of the joined database would contain these tuples. This "inconsistency" would not be a problem for some applications, but may be considered incorrect in others.

6.3.2 Pipelined Filtering

Pipelined filtering, which was introduced in the context of the project operation, can also be used to compute the join of two database streams. In the case of the join, two different data streams are sent into the pipeline from opposite directions and each tuple in one stream is compared to each tuple in the other stream as they pass each other. If the two tuples being compared satisfy the join condition, they are joined.

Pipelined filtering can also be used to eliminate duplicates resulting from joining two stream as was explained in the discussion of project with duplicate elimination.

7. DSF Query Optimization

For distributed databases on wide area networks (WANs), the communication cost is traditionally assumed to be dominant when choosing a query optimization strategy. Therefore, most query processing algorithms for distributed database systems attempt to minimize the amount of data transmitted over the network[6][18]. This is accomplished, in part, by performing select and project operations first and possibly using semi-joins. However, this traditional approach to query optimizations turns out to be inappropriate for distributed stream filtering. This section explains why this is the case and outlines the cost factors involved in DSF.

While communication costs remain high over voice grade modems, new technological developments promise scalable 2.4 Gb/s ATM networks within the next few years[2][20][21]. This disparity in communication speeds creates an environment in which network bandwidth, for many applications, can be considered a low-cost resource at the ATM network backbone, but is still very costly at the edges of the network[13]. Figure 7 illustrates how a DSF system takes advantage of the disparity in network resources. Large amounts of data are streamed out over the ATM network and filtered at various levels. However, only much smaller amounts of data, approximating the answers to user queries are then sent to the user across lower speed connections.

Another important factor to consider when optimizing queries for use with DSF is that large efficiency gains can often be achieved by reusing subqueries that are already being filtered. To take advantage of query reuse, we propose a strategy in which filtering operations are performed in order of generality. Joins and unions should be performed first, followed by selects and projects.

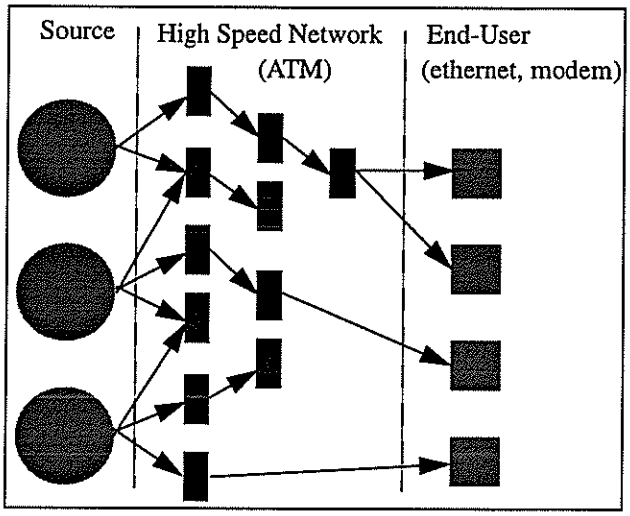

Figure 7: Placement of DSF modules in a disparate network environment

Unfortunately, the filtering operations that are the most general are also the most expensive in terms of data transmitted and computation. So, for an individual query, this strategy may seem expensive. However, if we expect that many users will require information from a join, for example, then it is best to do this expensive work early, as close to the source possible, rather than repeating it for multiple, narrower views of the database. Performing general operations first also makes optimal use of network resources. High data rates near the source(s) facilitate streaming of large amounts of data. The data is then reduced as it nears lower speed networks. Therefore, it is necessary to take both factors into account when optimizing a query. The point to be

made here is that in a DSF system it is not always optimal to reduce data by performing selects and projects before performing joins.

8. Other Applications of DSF

We have discussed the applicability of DSF for databases that are represented as streams. In this section, we examine other kinds of database applications that may benefit from distributed stream filtering.

8.1 Heavily Queried Databases

One possible alternative to querying stored databases is to stream these databases out. Databases that receive many similar queries would not have to recompute each query. Instead the query would be computed once and multicast to any number of users. Using a DSF system also distributes the processing burden throughout the network, alleviating the bottleneck at the database system.

8.2 Computationally Intensive Applications

Applications that have distributed computational needs will also benefit from using DSF. Multiple applications may require the use of a machine that is well suited to the particular problem (e.g., massively parallel machines) or a single application may require multiple distinct processing resources that are distributed throughout the network.

One such application is the TeleMed System under development at the Los Alamos National Laboratory[7]. TeleMed consists of distributed repositories of radiographic images and patient records and will allow physicians throughout the country to view radiographic data from their location. Of particular interest is the "query by example" feature that allows users to search image databases for images that match certain criteria. Images are matched based on signatures that are generated for each image and contain representative features of each image. The matching algorithm is inherently parallel as well as computationally intensive and, therefore, achieves significantly greater performance on a massively parallel computer. Many other operations in the TeleMed system do not require a massively parallel computer and can be performed on less expensive workstations. The choice of processing sites for each operation must be transparent to the user so the system must be able to distribute the processing of user queries throughout the network dynamically.

In Applications like TeleMed that require a variety of computing resources, filters in a DSF system can be located on machines that provide the necessary resources and data can then be streamed through the appropriate filters throughout the network to answer user queries. DSF also facilitates the dynamic placement and configuration of filters throughout the network, which greatly simplifies the development of applications with distributed processing needs.

8.3 Security Policy Enforcement

In database applications, different users may be authorized to access different portions of the database. In the case of patient records, certain doctors would have access to the personal data portion of a record (e.g. the patients name), and others may only have access to less personal portions of records.

DSF could be used to enforce security policies by producing different streams of data representing different levels of security. A stream that represented full access to database records could be routed through another filter which removed information that was not allowed by more stringent security policies and both streams could be sent to users based on their security level.

DSF would be especially useful when security policies are defined for data stored or produced at multiple locations (e.g. multiple corporate databases). DSF would relieve each individual database from enforcing security on its data and would serve as a distributed firewall for a database system.

9. Implementation Status

We are currently in the process of implementing a DSF system for relational databases that supports a subset of SQL. At this point, the underlying filtering modules have been completed and we are currently developing the control module infrastructure. When completed, the DSF system will run over a high-speed ATM network on machines using APIC chips. This will allow us to evaluate the performance of a DSF system in a high-speed communications environment.

This paper makes several limiting assumptions that should be addressed in future work. We have assumed that control modules in DSF system are statically placed at appropriate locations throughout the network. A more flexible and potentially more efficient system would permit the dynamic placement of control modules based on changing system resources and requirements. Finally, although we discuss a query optimization strategy in this paper, it would be helpful to develop an explicit cost model.

Acknowledgments

We thank Doug Schmidt for preliminary discussions on the topic. This research was supported in part by the National Science Foundation under grants CCR-94-12711 and CCR-94-12711S, and by the Advanced Research Projects Agency (ARPA) under contract DABT-63-95-C-0083.

References

1. Blaine, G.J., J.R. Cox, and R.G. Jost, Networks for Electronic Radiology," Radiologic Clinics of North America, May 1996, pp. 505-524.
2. Chaney, T., A. Fingerhut, M. Flucke and J.S. Turner, "Design of a Gigabit ATM Switch", Washington University Department of Computer Science, WUCS-96-07.
3. Codd, E. F., Relational Completeness of Data Base Sublanguages. In Data Base Systems, R. Rustin (ed.), Engleweed Cliffs, N.J.: Prentice-Hall, 1972, pp. 65-98.
4. Codd, E. F., "A Relational Model for Large Shared Data Banks," Communications of the ACM, October 1970, pp. 377-387.
5. Dittia, Z.D., J.R. Cox, and G.M. Parulkar, "Design of the APIC: A High Performance ATM Host-Network Interface Chip," Proc. IEEE INFOCOM 95, Boston, 1995, pp. 179-187.
6. Egyhazy C. and Triantis K., "A Query Processing Algorithm for Distributed Relational Database Systems," The Computer Journal 31:34-40, 1988.
7. Forslund, D., et al, "TeleMed: A Graphical, CORBA-based, Virtual Patient Record System for Clinical Use," Object Management Group, FIrst Class Magazine, March/April 1996. LA-UR-96-647.
8. Gehani, N., H. V. Jagadish and O. Shmueli, "COMPOSE: A System for Composite Specification and Detection," In Advanced Database Systems, N. R. Adam and B. Bhargava (eds.), Lecture Notes in Computer Science, Berlin, Germany: Springer-Verlag, 1993, pp. 3-15.
9. Goldman, K.J., B. Swaminathan, T.P. McCartney, M.D. Anderson and R. Sethuraman, "The Programmers' Playground: I/O Abstraction for User-Configurable Distributed Applications," IEEE Transactions on Software Engineering, 21(9):735-746, September 1995.
10. Goldman, K.J., T.P. McCartney, R. Sethuraman and B. Swaminathan, "The Programmers' Playground: A Demonstration," In Proceedings of the Third ACM International Multimedia Conference (MM'95), San Francisco, CA, November 1995, pp. 317-318.
11. Goldman, K.J., T.P. McCartney, R. Sethuraman, B. Swaminathan and T. Rodgers, "Building Interactive Distributed Applications in C++ With The Programmers Playground," Washington University Department of Computer Science WUCS-95-20, July 1995.
12. Goldman, K.J. et al. "Welcome to the Programmers Playground!" http://www.cs.wustl.edu/cs/playground/
13. Green, P. E. Jr., "Optical Networking Update," IEEE Journal on Selected Areas in Communications, June 1996, pp. 764-779.
14. Isis Distributed Systems, Inc., Marlboro, MA, Isis Users Guide: Reliable Distributed Objects for C++, April 1994.
15. McCanne, S. and V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture," in Proceedings of the Winter USENIX Conference, January 1993, pp. 259-270.
16. Mogul, J.C., R. F. Rashid and M.J. Accetta, "The Packet Filter: an Efficient Mechanism for User-level Network Code," in Proceedings of the 11th Symposium on Operating System Principles, November 1987.
17. National Weather Service. "Advanced Interactive Weather Processing System (AWIPS)." http://www.nws.noaa.gov/modernize/AWIPtech.htm
18. Özsu, M. T. and P. Valduriez, Principles of Distributed Database Systems. Englewood Cliffs, NJ: Prentice Hall, 1991.
19. Schmidt, D. C., "Scalable High-Performance Event Filtering for Dynamic Multi-point Applications," Proceedings of 1st Workshop on High Performance Protocol Architectures (HIPPARCH), INRIA, Sophia Antipolis, France, December, 1994, p 1--8.
20. Turner, J. S., "An Optimal Nonblocking Multicast Virtual Switch," In Proceedings of Infocom, June 1994, pp. 298-305.
21. Turner, J. S., "Multicast Virtual Circuit Switching Using Cell Recycling." US Patent #5,402,415, March, 1995.
22. Ullman, J.D., Principles of Database Systems (2nd edition). Rockville, Md: Computer Science Press, 1982.
23. Widom, J. and Ceri, S. (eds.), Active Database System. San Francisco, CA: Morgan Kaufmann, 1996.
24. Wolfson, O., S. Sengupta, and Y. Yemini, "Managing Communication Networks by Monitoring Databases," IEEE Transactions on Software Engineering, September 1991, pp. 944-952.
25. Yuhara, M., B. Bershad, C. Maeda, and E. Moss, "Efficient Packet Demultiplexing for Multiple Endpoints and Large Messages," in Proceedings of the Winter Usenix Conference, January 1994.