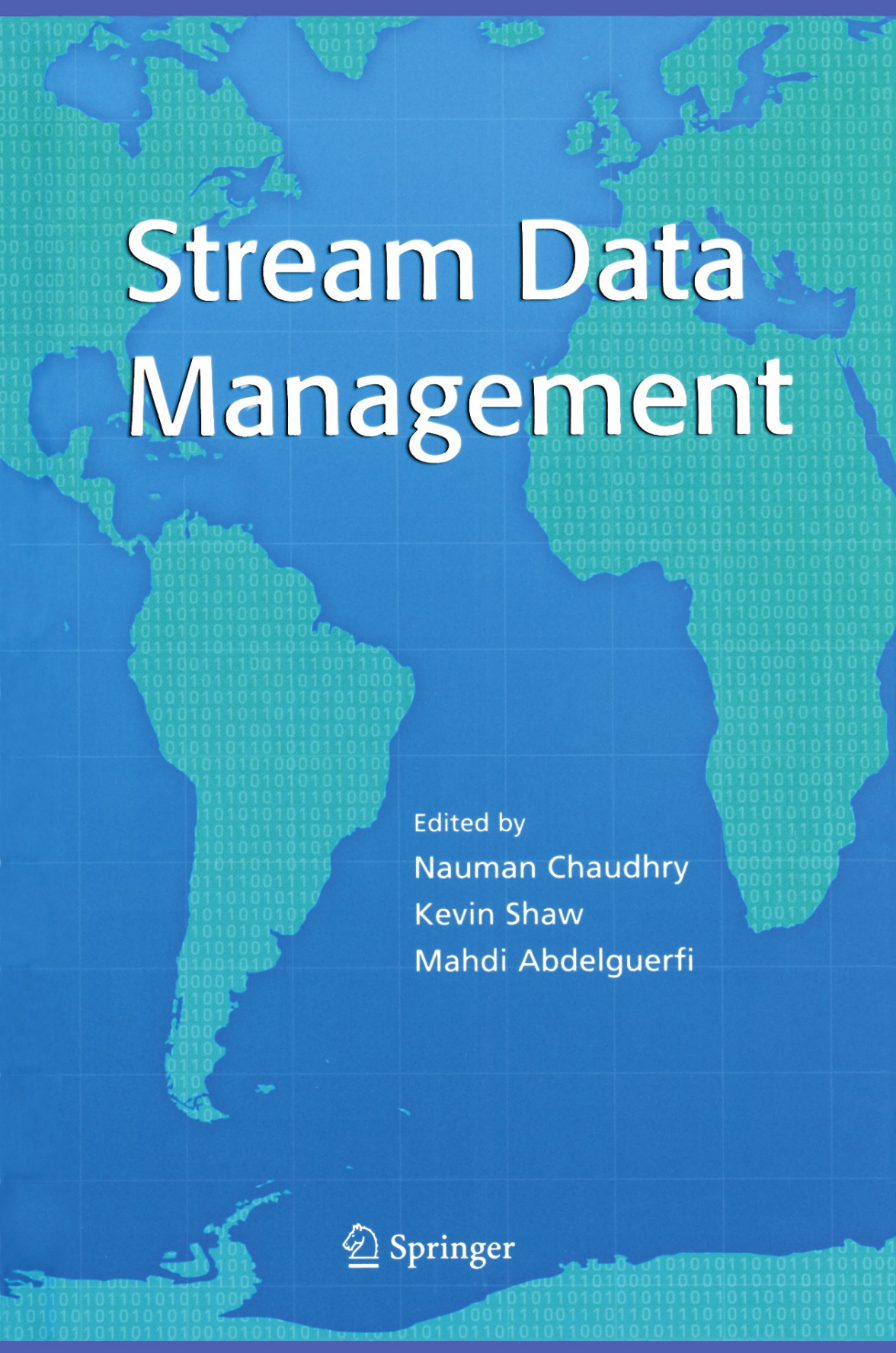

Stream Data Management
Edited by
Nauman Chaudhry
Kevin Shaw
Mahdi Abdelguerfi
Springer

STREAM DATA MANAGEMENT

ADVANCES IN DATABASE SYSTEMS

Series Editor

Ahmed K. Elmagarmid

Purdue University
West Lafayette, IN 47907

Other books in the Series:

FUZZY DATABASE MODELING WITH XML, *Zongmin Ma,* ISBN 0-387-24248-1; e-ISBN 0-387-24249-X

MINING SEQUENTIAL PATTERNS FROM LARGE DATA SETS, *Wei Wang and Jiong Yang;* ISBN 0-387-24246-5; e-ISBN 0-387-24247-3

ADVANCED SIGNATURE INDEXING FOR MULTIMEDIA AND WEB APPLICATIONS, *Yannis Manolopoulos, Alexandros Nanopoulos, Eleni Tousidou;* ISBN: 1-4020-7425-5

ADVANCES IN DIGITAL GOVERNMENT, *Technology, Human Factors, and Policy,* *edited by William J. McIver, Jr. and Ahmed K. Elmagarmid;* ISBN: 1-4020-7067-5

INFORMATION AND DATABASE QUALITY, *Mario Piattini, Coral Calero and Marcela Genero;* ISBN: 0-7923- 7599-8

DATA QUALITY, *Richard Y. Wang, Mostapha Ziad, Yang W. Lee:* ISBN: 0-7923-7215-8

THE FRACTAL STRUCTURE OF DATA REFERENCE: *Applications to the Memory Hierarchy,* *Bruce McNutt;* ISBN: 0-7923-7945-4

SEMANTIC MODELS FOR MULTIMEDIA DATABASE SEARCHING AND BROWSING, *Shu-Ching Chen, R.L. Kashyap, and Arif Ghafoor;* ISBN: 0-7923-7888-1

INFORMATION BROKERING ACROSS HETEROGENEOUS DIGITAL DATA: A Metadata-based Approach, *Vipul Kashyap, Amit Sheth;* ISBN: 0-7923-7883-0

DATA DISSEMINATION IN WIRELESS COMPUTING ENVIRONMENTS, *Kian-Lee Tan and Beng Chin Ooi;* ISBN: 0-7923-7866-0

MIDDLEWARE NETWORKS: Concept, Design and Deployment of Internet Infrastructure, *Michah Lerner, George Vanecek, Nino Vidovic, Dad Vrsalovic;* ISBN: 0-7923-7840-7

ADVANCED DATABASE INDEXING, *Yannis Manolopoulos, Yannis Theodoridis, Vassilis J. Tsotras;* ISBN: 0-7923-7716-8

MULTILEVEL SECURE TRANSACTION PROCESSING, *Vijay Atluri, Sushil Jajodia, Binto George* ISBN: 0-7923-7702-8

FUZZY LOGIC IN DATA MODELING, *Guoqing Chen* ISBN: 0-7923-8253-6

INTERCONNECTING HETEROGENEOUS INFORMATION SYSTEMS, *Athman Bouguettaya, Boualem Benatallah, Ahmed Elmagarmid* ISBN: 0-7923-8216-1

FOUNDATIONS OF KNOWLEDGE SYSTEMS: With Applications to Databases and Agents, *Gerd Wagner* ISBN: 0-7923-8212-9

DATABASE RECOVERY, *Vijay Kumar, Sang H. Son* ISBN: 0-7923-8192-0

For a complete listing of books in this series, go to http://www.springeronline.com

STREAM DATA MANAGEMENT

edited by

Nauman A. Chaudhry
University of New Orleans, USA

Kevin Shaw
Naval Research Lab, USA

Mahdi Abdelguerfi
University of New Orleans, USA

Springer

Nauman A. Chaudhry
University of New Orleans
USA

Kevin Shaw
Naval Research Lab
USA

Mahdi Abdelguerfi
University of New Orleans
USA

Library of Congress Cataloging-in-Publication Data

A C.I.P. Catalogue record for this book is available from the
Library of Congress.

STREAM DATA MANAGEMENT
edited by
Nauman A. Chaudhry
Kevin Shaw
Mahdi Abdelguerfi

Advances in Database Systems
Volume 30

ISBN 0-387-24393-3 e-ISBN 0-387-25229-0

Cover by Will Ladd, NRL Mapping, Charting and Geodesy Branch
utilizing NRL's GIDB® Portal System that can be utilized at
http://dmap.nrlssc.navy.mil

Printed on acid-free paper.

Printed in the United States of America.

9 8 7 6 5 4 3 2 1 SPIN 11054597, 11403999

springeronline.com

Contents

List of Figures

List of Tables

Preface

In recent years, a new class of applications has emerged that requires managing data streams, i.e., data composed of continuous, real-time sequence of items. However, database management systems were originally developed to support business applications. The data in such applications is changed as a result of human-initiated transactions. Similarly data is queried as a result of human-initiated queries. The database management system acts as a passive repository for the data, executing the queries and transactions when these are submitted. However, this model of a database management system as a repository of relatively static data that is queried as a result of human interaction, does not meet the challenges posed by streaming applications.

A data stream is a possibly unbounded sequence of data items. Streaming applications have gained prominence due to both technical and business reasons. Technologically data is now available from a wide variety of monitoring devices, including sensors that are extremely cheap. Data from such devices is potentially unbounded and needs to be processed in real-time. Additionally businesses and Federal agencies now increasingly want to perform analysis on data much sooner than is possible with the current model of storing data in a data warehouse and performing the analysis off-line. Application domains requiring data stream management include military, homeland security, sensor networks, financial applications, network management, web site performance tracking, real-time credit card fraud detection, etc.

Streaming applications pose new and interesting challenges for data management systems. Such application domains require queries to be evaluated continuously as opposed to the one time evaluation of a query for traditional applications. Streaming data sets grow continuously and queries must be evaluated on such unbounded data sets. The monitoring aspect of many streaming applications requires support for reactive capabilities in real-time from data management systems. These, as well as other challenges, require a major rethink of almost all aspects of traditional database management systems to support streaming applications. Consequently, stream data management has been a very active area of research over the past few years.

The goal of this edited manuscript is to gather a coherent body of work spanning various aspects of stream data management. The manuscript comprises eight invited chapters by researchers active in stream data management. The collected chapters provide exposition of algorithms, languages, as well as systems proposed and implemented for managing streaming data. We expect this book will appeal to researchers already involved in stream data management, as well as to those starting out in this exciting and growing area.

Chapter 1

INTRODUCTION TO STREAM DATA MANAGEMENT

Nauman A. Chaudhry
Department of Computer Science, University of New Orleans
2000 Lakeshore Drive, New Orleans, LA 70148
nauman@cs.uno.edu

Abstract In recent years, a new class of applications has emerged that requires managing data streams, i.e., data composed of continuous, real-time sequence of items. This chapter introduces issues and solutions in managing stream data. Some typical applications requiring support for streaming data are described and the challenges for data management systems in supporting these requirements are identified. This is followed by a description of solutions aimed at providing the required functionality. The chapter concludes with a tour of the rest of the chapters in the book.

Keywords: stream data management, stream query languages, streaming operators, stream data management systems.

1. Why Stream Data Management?

Database management systems (DBMSs) were originally developed to support business applications. The data in such applications is changed as a result of human-initiated transactions. Similarly the data is queried as a result of human-initiated queries. The DBMS acts as a passive repository for the data executing the queries and transactions when these are submitted. But such traditional DBMSs are inadequate to meet the requirements of a new class of applications that require evaluating persistent long-running queries against continuous, real-time stream of data items.

1.1 Streaming Applications

A data stream is a possibly unbounded sequence of tuples. At a high level, we can distinguish between two types of data streams, transactional and measurement [Koudas and Srivastava, 2003].

Transactional Data Streams: Transactional data streams are logs of interaction between entities. Example application domains with such data include:

- In many large web sites, the interaction of users with the web site is logged and these logs are monitored for applications such as performance monitoring, personalization. The information on this interaction continuously gets appended to the log in the form of new log entries.
- Vendors of credit card monitor purchases by their credit card holders to detect anomalies that indicate possible fraudulent use of a credit card issued by them. Log of credit card transactions forms a continuous data stream of transactional data.

Measurement Data Streams: These types of data streams are produced as a result of monitoring the state of entities of interest. Example application domains include:

- With sensors becoming cheaper, a whole area of applications is emerging where sensors distributed in the real world measure the state of some entities and generate streams of data from the measured values. E.g., such sensors may be monitoring soldiers in battlefield, or traffic on highways, or temperature at weather stations.
- Large-scale communication networks require continuous monitoring for many tasks, such as locating bottlenecks, or diagnosing attacks on the network. The data monitored consists of the packet header information across a set of network routers. Thus this data can be viewed as a data stream.

The typical queries that such application domains issue are long-running, standing and persistent. If the application has registered a query, the data management systems should continually evaluate the query over a long period of time. This evaluation is repeated at some defined interval.

EXAMPLE 1.1 Let us consider a concrete example domain, namely road traffic monitoring [Stream Query Repository; Arasu et al, 2004].

Consider a scenario where sensor embedded along expressways report on current traffic on the road. The data being reported by the sensors forms a continuous stream giving:

- Identifier of a car as well as its speed.

- The id of the expressway and the segment and lane of the expressway on which the car is traveling, as well the direction in which the car is traveling.

Various queries would be evaluated over this data.

a To manage the flow of traffic, a query would compute the average speed of cars over a segment. If the speed drops below a threshold, commuters about to travel on that segment can be notified to take alternative routes.

b Again, based on average speed of vehicles over a segment, the application might alert operators to possible accidents that are causing a slow down on that segment.

c On toll roads, the data stream is used to automatically deduct toll from the account of those vehicle owners that have tags recognized by the sensors.

d In advanced traffic management a goal is to manage congestion by varying the toll rate. To achieve this goal, the application might deduct toll as a function of average speed on the expressway.

1.2 Traditional Database Management Systems and Streaming Applications

Let us look at how the capabilities of traditional DBMSs match up to the functional requirements of streaming applications, and in particular, the road traffic management application described in the previous section.

- **One-time vs. continuous queries**: Applications of traditional DBMSs issue one-time queries, i.e., after issuance a query is evaluated once and the result at that point in time is returned to the user. As opposed to this, streaming applications require that once a query is registered then it should be continuously evaluated until it is deregistered. E.g., the road traffic management application requires that average speed of cars in the system should be continually evaluated based on new values reported by the sensors.

- **Notion of time**: Data in traditional applications does not necessarily have a notion of time. An update of an attribute overwrites the previous value of the attribute. In data streams on the other hand data items represent a sequence of values for the same entity. The queries over the streaming data also extend over a history of values. E.g., the traffic management application might require that the average speed of cars be computed over the last 5 minutes.

- **Unbounded Data Sets**: Queries in traditional DBMSs are evaluated against data that is finite and does not change while the queries are being executed. Perchance the data does change during query execution, the DBMSs implement mechanisms (e.g., cursor stability) so that the result of the query execution is correct with respect to a specific instance of time. In contrast to this, the datasets for streaming queries are continuously growing and queries must be evaluated over such unbounded data sets.

- **Unreliable Data**: Traditional DBMSs deal with data that are assumed to be accurate. However, for streaming applications, the DBMS must be capable of dealing with unreliable data. The causes of such unreliability may be sensors that have failed or delays in network that cause the data to be reported out-of-order. In certain domains, such as weather monitoring, even the data reported in correct sequence may have an associated degree of accuracy determined by the accuracy of the sensor reporting the value.

- **Reactive capability**: Traditional DBMS applications are mostly passive. The human initiates a transaction and the DBMS executes the transaction and returns the results to the human. A lot of streaming applications though have a monitoring nature, i.e., the application needs to react automatically when certain conditions hold over the data. E.g., if traffic on a segment slows down below a threshold, drivers of cars about to travel on that segment need to be automatically informed of alternative routes. This reactive capability shouldn't require human intervention. This requirement has led some researchers to describe the model for streaming applications as DBMS-Active, Human Passive in contrast with the traditional Human-Active, DBMS Passive model [Carney et al., 2002].

1.3 Towards Stream Data Management Systems

Database applications that exhibit subsets of requirements of streaming applications have existed for a long time. For example, the need for reactive capability has been recognized for almost two decades and research on active database systems was undertaken towards fulfilling that need [Widom and Ceri, 1996]. Similarly temporal database systems explicitly model the temporal aspect of data [Jensen and Snodgrass, 1999]. However, recent trends driven both by applications and technology have brought to prominence streaming applications that require a combination of the afore-mentioned requirements along with high levels of scalability.

Technologically availability of cheap sensors implies that pretty soon practically any object of interest can be tracked and thus a whole new class of monitoring applications emerges. Along the application dimension, performing very sophisticated analysis in near real-time is seen as an important business need,

as opposed to the traditional approach of carrying out such analysis off-line on data imported into data warehouses [Koudas and Srivastava, 2003].

The challenges posed by streaming applications impose certain architectural and functional requirements on stream data management systems (or stream systems for short). Some of the important high-level requirements are described below:

- The data model must include the notion of time, while the query mechanism should support definition of continuous queries over such data. Further discussion of issues related to data model and query languages appears in Section 2.
- Given the unbounded nature of streaming data and the fact that the data might be unreliable in certain case, a stream system must be capable of computing answers based on incomplete or unreliable data.
- An implication of the above requirement is that the stream system may not use blocking operators that process the entire stream before producing an answer. A trade-off has to be made between non-blocking operators capable of computing approximate answers and blocking operators computing exact answers.
- Since the queries are long-standing, the stream system should be capable of coping with variations in conditions. This implies that query execution plans need to be adaptive and dynamic instead of traditional static plans.
- Clearly stream systems need to have reactive capability. The challenge here is to build reactive capability that scales to possibly thousands of triggers as opposed to the current systems that don't scale beyond a limited number of triggers.
- In many streaming applications, allied to this need of reactive capability is the requirement of providing this capability in (near) real-time. This requires that a stream system intelligently manages resources and can provide Quality of Service (QoS) guarantees.
- A large number of streaming applications are inherently distributed in nature. E.g., sensors for traffic monitoring would be distributed over a large area. This distribution makes bandwidth considerations important in query processing. Furthermore, sensors may have limited battery life, so query processing must make efficient use of battery power.

1.4 Outline of the Rest of the Chapter

Having introduced important characteristics of streaming applications and the challenges posed on data management systems by these applications, in

the following sections we give an overview of solutions aimed at overcoming these challenges[1]. In Section 2, we discuss issues and solutions related to data model and query languages. Section 3 includes description of work in implementing query operators. A discussion of various projects related to stream data management appears in Section 4, while Section 5 gives a brief tour of the rest of the book.

2. Stream Data Models and Query Languages

A common, though not the only, approach in stream data models is to model each item in a data stream as a relational tuple with a timestamp or sequence number and developing a declarative SQL-like language for querying data streams [Arasu et al., 2003; Chandrasekaran et al., 2003; Lerner and Shasha, 2003].

2.1 Timestamps

An important issue in the data model is defining timestamps for items in the data stream. We can differentiate between different types of timestamps [Babcock et al., 2002].

- **Implicit timestamp** is added to an arriving tuple by the stream management system. This may simply be a sequence number that imposes a total order on the tuples in the stream.
- **Explicit timestamp** is a property of the stream data item itself. Typically streams in which tuples correspond to real-world events will have an explicit timestamp.

An additional issue is assigning an appropriate timestamp to a tuple output by a binary operator, e.g., a join. For streams with implicit timestamps one possibility is that the produced tuple is also assigned an implicit timestamp based on the time it was produced. Another approach applicable to either implicit or explicit timestamps is that the query includes the user's specification of the timestamp to assign to the new tuple.

2.2 Windows

For unbounded data streams many queries are interested in a subset or part of the complete stream. This requires support for specifying the range of tuples over which the query result should be computed via *windows*. Window specification in data stream is an extension of the SQL-99's notion of expressing physical or logical windows on relations.

EXAMPLE 1.2 Consider a traffic management application where car locations are being reported via a data stream `CarLocStream` [Stream Query Repository].

The tuples in this stream have attributes that specify the id of the car, its speed, the expressway, the direction, lane and segment on which the car is traveling, in addition to the (implicit or more likely explicit) timestamp attribute. A query to get the average speed of the cars over the last 5 minutes would be expressed as:

```
SELECT exp_way, dir, seg, AVG(speed)
FROM CarSegStr [RANGE 5 MINUTES]
GROUP BY exp_way, dir, seg
```

In Example1.2, `RANGE 5 MINUTES` defines a window of 5 minutes implying that the query results should be computed over all tuples received in the previous 5 minutes. This is an example of a *time-based (or physical) window*. An alternative specification is a *tuple-based (or logical) window* where the size of the window is expressed in terms of number of tuples.

Choices also exist in defining the *scope of the window* as new data items arrive [Gehrke et al., 2001]. In *sliding windows* the width of the window remains fixed but both the endpoints move to replace old items with new items. The query in Example1.2 uses a sliding window to evaluate the query over the tuples that have arrived in the last 5 minutes and replaces old tuples with new ones.

In *landmark windows* one endpoint remains fixed while the other endpoint moves. In the traffic monitoring application, a query computing the average speed since a particular accident was reported would use a landmark window.

2.3 Proposed Stream Query Languages

Most stream processing systems have proposed and developed declarative query languages by using SQL as a starting point and adding constructs for querying streams. These constructs provide specification of:

- Size and scope of windows,
- Referencing both streams and relations in the same query and possibly converting between these two representations,
- The frequency and count of query execution.

The SQL-like query in the Example1.2 in Section 2.2 is adapted from a repository of example stream queries maintained in the Stream Query Repository project. The query is expressed in CQL (Continuous Query Language) developed as part of Stanford's STREAM system [Arasu et al., 2003]. Other examples of SQL-like languages include StreaQuel [Chandrasekaran et al., 2003], GSQL [Johnson et al., 2003], AQuery [Lerner and Shasha, 2003]. Chapter 6 of this book contains a detailed description of ESL-TS, a powerful SQL-like language developed for stream querying.

A different approach for query specification is taken in the Aurora system [Carney et al., 2002]. Instead of extending SQL, a graphical interface is provided to the user to specify queries by defining the data flow through the system. Using the graphical interface an application administrator can specify queries by placing boxes, that represent different operators, at specific points on various streams as they flow through the Aurora system.

3. Implementing Stream Query Operators

3.1 Query Operators and Optimization

To support querying over streams, suitable streaming versions need to be created for relational operators. Stateless operators, i.e., operators that do not need to maintain internal state, can simply be used as is in querying streams [Golab and Ozsu, 2003]. Examples of stateless operators are the selection operator and the projection operator that does not eliminate duplicates. Adapting stateful operators, i.e., operators that need to maintain internal state, to stream querying is though a much more involved tasks. Examples of stateful operators are join and aggregate functions.

Streaming environments also pose certain challenges on the techniques used to implement the operators and to execute the queries. The need for non-blocking operators and adaptive query plans has been mentioned before. To provide scalability, operator execution may be shared between different concurrent queries. This task can leverage work in traditional DBMS for optimizing multiple queries [Sellis, 1988]. Additionally, the need for real-time capability implies that making more than one pass over the data might be unfeasible, so algorithms may be restricted to making a single pass over the data.

For a detailed discussion of query optimization for streaming systems see Chapter 2 of the book, while for details of techniques to overcome the challenge of non-terminating data streams see Chapter 3.

3.2 Performance Measurement

With the myriad efforts in implementing stream data management systems, an important question is what metrics can be used to evaluate the systems. Possible metrics for comparing the functionality and performance of stream systems include:

- Response time: How long does it take for the system to produce output tuples?
- Accuracy: In case the system gets overloaded and resources are insufficient to cope with the stream arrival rate, the stream system may resort to approximate answers. How accurate the stream system is for a given load of data arrival and queries?

- Scalability: How much resources are needed by the stream system to process a given load with a defined response time and accuary?

Quality of Serivice (QoS) is another possible metric to measure performance. The Aurora system includes a QoS monitor to detect overload and poor system performance [Carney et al., 2002]. QoS is defined as a function of various attributes, such as response time and accuracy.

An important work in the area of performance measurement of stream systems is the Linear Road Benchmark [Arasu et al, 2004]. This benchmark simulates traffic on a highway system that uses "variable tolls" and includes:

- A set of continuous queries that monitor data streams and historical queries that query previously streamed data.
- Requirements on high-volume streaming data as well as historical data that must be maintained.
- Requirements on response and accuracy of real-time and historical queries.

4. Prototype Stream Data Management Systems

A number of stream systems have been and are being developed in academia as well as industry to meet the challenges of managing stream data. Below we give an overview of some of the prominent systems:

- Aurora [Carney et al., 2002]: Aurora is a data flow oriented system with a set of operators that can be used via a graphical user interface. A user specifies queries by arranging boxes, representing operators, and arrows, representing data flow among the operators. In essence thus the user actually specifies query plans. The Aurora system optimizes the flow of data in the system by real-time scheduling of the operators.
- CAPE: See Chapter 5 for details of this system.
- COUGAR [Demers et al., 2003]: COUGAR system has been developed to provide data management for sensor data. Instead of tuples, the data is modeled via ADTs (abstract data types). Query specification and execution utilize functions defined on the ADT.
- Gigascope [Johnson et al., 2003]: Gigascope is a specialized stream system built at AT&T for network applications. The system provides a declarative query language called GSQL is used.
- Hancock [Cortes et al., 2000]: The Hancock system is also built at AT&T. This system provides a C-based procedural language for querying transactional data streams. The target application is tracking calling patterns of around 100 million callers and raising real-time fraud alerts.

- NiagaraCQ [Chen et al., 2000]: NiagaraCQ is is a system for continuous query processing over dynamic data sets on the web. The data model and the query language are based on XML and XML-QL rather than relations and SQL.

- StatStream [Zhu and Shasha, 2002]: StatStream system is geared towards computing online statistics over data streams. The system is capable of computing statistcs over single streams, as well as correlations among streams.

- STREAM [Stream 2003]: STREAM system is a stream data manager with a SQL-like declarative query language called CQL. The system intelligently manages resources and is capable of dynamic approximation of results based on load and available resources.

- TelegraphCQ [Chandrasekaran et al., 2003]: TelegraphCQ is a system to continuously process queries over data streams. TelegraphCQ includes highly adaptive query optimization techniques for evaluating multiple queries over data streams.

- Tapestry [Terry et .al, 1992]: Though not a complete streaming system, this early work at Xerox PARC introduced the notion of continuous queries and implemented extensions to SQL to specify and execute such queries. The target application for these continuous queries was filtering email and news messages.

5. Tour of the Book

As described earlier in this chapter, query execution and optimization over streaming data requires fundamental changes to approaches used in traditional database systems. In Chapter 2, Viglas explores this issue in detail and presents static as well dynamic approaches to stream query optimization.

The non-terminating and high volume nature of many data steams presents major challenge for processing queries. In Chapter 3, Maier, et. al., describe techniques, including filters, punctuations, windowing and synopses, that are being developed to address the challenge of processing queries over unbounded data streams.

Discussion in the preceding sections has been on streaming data modeled as tuples. However, as Bruno, et. al., describe in Chapter 4, issues in streaming data need to be addressed for XML data as well. When XML is used in web services and for data dissemination in publish/subscribe systems, XML data collections are no longer static. Instead XML documents arrive continuously thus forming a stream of documents. The authors present algorithms for evaluating multiple XML queries against such a stream of XML documents.

In Chapter 5, Rundensteiner, et. al., describe the stream processing system CAPE (for Constraint-Aware Adaptive Stream Processing Engine). Given the dynamic nature of streaming data and environments, the ability to adapt is very important for streaming data management. CAPE adopts a novel architecture with highly adaptive services at all levels of query processing.

As discussed in Section 2, supporting continuous queries on streaming data requires appropriate query languages. Such query languages must have adequate expressive power and must be suitable for use by query optimizer. Bai, et. al., describe the Expressive Stream Language for Time Series (ESL-TS) in Chapter 6. To support querying over streaming data, ESL-TS includes SQL-like constructs to specify input data stream and patterns over such streams. The chapter also describes optimization techniques developed for ESL-TS.

A system that provides access to several types of geographical data, including stream data, is described in Chapter 7 by Sample, et. al.. This portal system, called Geospatial Information Database (GIDB) has been developed at Naval Research Labs to link together several hundred geographical information databases. The types of data servers linked by the portal range from web pages and geographic data files accessed via file transfer, to database management systems, geographical information systems and streaming data.

The book concludes with Chapter 8, in which Shah and Ramamritham describe architectures for managing streaming data in peer-to-peer systems.

Notes

1. Other surveys of stream managment systems can be found in [Koudas and Srivastava, 2003; Golab and Ozsu, 2003; Babcock et al., 2002]. Description of some stream management systems can also be found in a special issue of IEEE Data Engineering Bulletin devoted to stream data management[IEEE Bulletin, 2003]

Acknowledgments

The author was supported in part by Louisiana Board of Regents and NASA through Louisiana Space Consortium contract NASA/LEQSF(2004)-DART-11.

References

A. Arasu, S. Babu, and J. Widom (2003). The CQL continuous query language: semantic foundations and query execution. Stanford University TR No. 2003-67.

A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. Maskey, E. Ryvkina, M. Stonebraker and R. Tibbetts (2004). Linear Road: A Benchmark for Stream Data Management Systems. In *Proceedings of VLDB Conference.*

B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom (2002). Models and issues in data stream systems. In *Proceedings of PODS Conference.*

D. Carney, U. Cetintemel, M. Cherniack, C. Convey, L. Christian, S. Lee, G. Seidman, M. Stonebraker, Michael, N. Tatbul, and S. Zdonik (2002). Monitoring Streams - A New Class of Data Management Applications. In *Proceedings of VLDB Conference*, pages 215–226.

S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah (2003). TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Conference on Innovative Data Systems*.

J. Chen, D. DeWitt, F. Tian, Y. Wang (2000). NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of SIGMOD Conference*.

C. Cortes, K. Fisher, D. Pregibon, A. Rogers (2000). Hancock: a Language for Extracting Signatures from Data Streams. In *Proceedings of Conference on Knowledge Discovery and Data Mining*.

A. Demers, J. Gehrke, R. Rajaraman, N. Trigoni, and Y. Yao (2003). The Cougar Project: A Work-In-Progress Report. In *Sigmod Record*, Volume 34, Number 4, December 2003.

J. Gehrke, F. Korn, and D. Srivastava (2001). On Computing Correlated Aggregates Over Continual Data Streams. In *Proceedings of SIGMOD Conference*.

L. Golab and M. Ozsu (2003). Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

IEEE Data Engineering Bulletin, Special Issue on Data Stream Processing. Vol. 26 No. 1, March 2003.

C. Jensen and R. Snodgrass (1999). Temporal Data Management. In *IEEE Transactions on Knowledge and Data Engineering*. 11(1).

T. Johnson, C. Cranor, O. Spatscheck, and V. Shkapenyuk (2003). Gigascope: A stream database for network applications. In *Proceedings of ACM SIGMOD Conference*, pages 647–651.

N. Koudas and D. Srivastava (2003). Data stream query processing: a tutorial. In *Proceedings of VLDB Conference*.

A. Lerner and D. Shasha (2003). AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. In *Proceedings of VLDB Conference*.

T. Sellis (1988). Multiple-Query Optimization. In *ACM TODS*. 13(1): 23-52.

Stream Query Repository, Stanford University. http://www-db.stanford.edu/stream/sqr/

The STREAM Group, 2003. STREAM: The Stanford Stream Data Manager. In *IEEE Data Engineering Bulletin*, Vol. 26 No. 1, March 2003.

D. Terry, D. Goldberg, D. Nichols, and B. Oki (1992). Continuous Queries over Append-Only Databases. In *Proceedings of ACM SIGMOD Conference*, pages 321–330.

Jennifer Widom, Stefano Ceri (1996). *Active Database Systems: Triggers and Rules For Advanced Database Processing*. Morgan Kaufmann.

Y. Zhu and D. Shasha (2002) StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proceedings of VLDB Conference*.

Chapter 2

QUERY EXECUTION AND OPTIMIZATION

Stratis D. Viglas
School of Informatics
University of Edinburgh, UK
sviglas@inf.ed.ac.uk

Abstract Query execution and optimization for streaming data revisits almost all aspects of query execution and optimization over traditional, disk-bound database systems. The reason is that two fundamental assumptions of disk-bound systems are dropped: (i) the data resides on disk, and (ii) the data is finite. As such, new evaluation algorithms and new optimization metrics need to be devised. The approaches can be broadly classified into two categories. First, there are static approaches that follow the traditional optimize-then-execute paradigm by assuming that optimization-time assumptions will continue to hold during execution; the environment is expected to be relatively static in that respect. Alternatively, there are adaptive approaches that assume the environment is completely dynamic and highly unpredictable. In this chapter we explore both approaches and present novel query optimization and evaluation techniques for queries over streaming sources.

Keywords: Query execution, query optimization, arrival rate, sliding windows, join processing, operator scheduling, load shedding, query scrambling, adaptive evaluation.

1. Introduction

Stream data management has emerged in recent years as a research area of database systems. The need to address stream data management is mainly pragmatic since it is largely application-driven. A typical scenario in which the need for stream data management arises is a network of a large number of nodes (in the order of thousands or millions) where *real-time* processing over the information transmitted by these nodes is necessary. There are numerous real-world examples of such a scenario: sensors transmitting information about the environment in which they are deployed is one; real-time analysis of pairings between callers and dialed parties used by telecommunication companies

towards fraud detection is another; analysis of Internet Protocol (IP) network traffic, or concurrent accesses by external users to the shared resources of a cluster of Web servers are additional instances of such a scenario.

In this chapter we will focus on the practical need of processing streaming data by employing traditional database operators. As we will see, this opens an avenue of problems. Although the operations are well-known, in the sense that they are the operations of relational algebra, the evaluation of these operations is entirely different than in traditional database systems. The main reason is that by moving from disk-based data to network-bound data, two of the fundamental database assumptions are dropped: firstly, the data is no longer stored on a local disk; as if this were not enough, a stream may in fact be infinite in length. This means that the query processing paradigm employed by database systems needs to be revisited and, in consequence, largely modified to address the new assumptions.

The two dominant approaches to query execution and optimization for queries over streaming information sources can be classified into two broad categories:

- *The static approach*: The premise is that the system is relatively static, i.e., the optimize-time environment is not significantly different than the run-time environment. The latter is also expected to remain stable throughout query execution. For instance, a query over a continuous feed of data arriving at a constant rate is a good candidate for static optimization and execution.

- *The adaptive approach*: In this scenario, the environment is completely dynamic in the sense that the system cannot make any assumptions on the nature of the inputs, the rates at which they arrive, the selectivity factors of the predicates and so on. The only thing the system can do is adapt to the changing environment.

The remaining of this chapter is organized as follows: execution of conjunctive queries over streams is presented in Section 2, while static optimization approaches are presented in Section 3. Adaptive query evaluation is the subject of Section 4. Finally, a summary of query execution and optimization in streaming environments is given in Section 5.

2. Query Execution

We will concentrate on a powerful subset of relational algebra: *conjunctive queries*, *i.e.,* queries containing projections and conjunctions of selections and joins. We assume that the system employs a *push-based execution model*, *i.e.,* the tuples of the incoming streams, as well as the operator output tuples, are immediately pushed to subsequent operators for further processing. We will be deriving generic cost expressions for each evaluation algorithm we present

Table 2.1. Notation used in the extraction of cost expressions.

Term	*Explanation*
f_p	The selectivity factor of predicate p
λ_S	The average incoming rate of stream S ($\frac{\text{tuples}}{\text{unit time}}$)
W_S	The window size for stream S
$\mu_s(S)$	Processing cost of storing a tuple from S into a memory structure
$\mu_m(S)$	Processing cost of obtaining matches from S's memory structure
$\mu_i(S)$	Processing cost of invalidating a tuple in S's memory structure

in terms of *computational cost* and *output rate*. In subsequent sections we will see how these cost expressions can be used to optimize queries.

One subtlety that needs to be addressed is the semantics of queries over infinite sources. It is obvious that they have to be modified for certain operators. For instance, a join over infinite sources means that infinite memory is needed to buffer the inputs. We need to a mechanism to extract finite subsets of the inputs. This mechanism is sliding windows: a sliding window can be expressed either in terms of time (*e.g.*, "in the last then seconds") or in terms of tuples (*e.g.*, "in the last 1,000 tuples"). There is a clear way of moving from time-based to tuple-based windows: Little's Theorem [Bertsekas and Gallager, 1991] states that given a process with an average arrival rate of λ and a time period T, then the expected number of arrivals is equal to $\lambda \cdot T$. From the previous equation, a time-based window of length T becomes a tuple-based window of size N.

We will be using the notation of Table 2.1; there, we are using the general term *memory structure* to denote any structure that stores tuples. As we will see, the chosen memory structure has an impact on performance when it comes to join evaluation, but this discussion is deferred until a later time (see Section 2.2).

2.1 Projections and Selections

Selections and projections are unary operators, meaning they only affect the properties of a single stream. We will address projections as a special case of selections with a selectivity factor equal to one, *i.e.*, all tuples are propagated to the output. Assume that the number of tuples arriving from stream S in a time unit is equal to λ_S. Of those tuples, only f_p will qualify, so the output rate will be equal to $f_p\lambda_S$. On the other hand, if there are window constraints, the size of the window needs to be scaled by the same factor, so for a window size equal to W the output window will be equal to f_pW. These results are summarized in Equations 2.1 and 2.2, where λ_O and W_O denote an output rate and window size respectively.

$$\lambda_O = f_p \cdot \lambda_S \quad (2.1)$$
$$W_O = f_p \cdot W_S \quad (2.2)$$

2.2 Join Evaluation

Join evaluation even on traditional, relational, disk-bound database systems has received enormous attention. This is quite expected since a join is one of the most commonly used database operations. It is no surprise that join evaluation over network-bound streams is one of the most intensive query execution research areas. The key issue that needs to be resolved is that inputs arrive *asynchronously* at the stream processor. A secondary issue is that extra passes over the data (if and when possible) are better to be avoided. The need for disk-bound operation arises if the cardinality of the inputs, whether it is their true cardinality in the case of finite streams, or their window cardinality in the case of infinite streams, is too large for working memory. As a result, the stream has to be spooled to disk for later processing. In such cases, extra mechanisms need to be in place to facilitate disk-based operation.

2.2.1 Memory-fitting Sources and Sliding Windows. The first instance of an asynchronous join evaluation algorithm, and the ancestor of all algorithms that will be discussed, is the *symmetric hash join* (SHJ) [Wilschut and Apers, 1991]. Consider the equi-join operation $R \bowtie_{R.a=S.b} S$, and assume both inputs fit entirely in memory. Symmetric hash join proceeds symmetrically for both inputs. Two hash tables are built, one for R and one for S. As tuples from R (respectively, S) arrive, the sequence of operations, depicted in Figure 2.1, is as follows: (i) they are hashed on the join key $R.a$ (respectively, $S.b$) to obtain the hash value for the tuple; (ii) they are used to probe the hash table for S (respectively, R) for matches; and, finally (iii) they are stored in the hash table for R (respectively, S);

SHJ was first employed for finite, memory-fitting sources. The difference when it comes to infinite sources and sliding window predicates is that one additional step needs to be taken: *invalidating* tuples that have expired from the window [Kang et al., 2003; Golab and Ozsu, 2003]. Furthermore, it only works for equi-join predicates, *i.e.,* equality predicates between the streams. An interesting observation is that not only hash tables, but also trees or straightforward lists can be used as memory structures for storing the tuples of the streams. That way non-equi-join predicates can be evaluated using the same principles. Furthermore, use of a different memory structure, and under certain combinations of incoming rates, may considerably improve performance [Kang et al., 2003; Golab and Ozsu, 2003]. Choosing the best structure for a particular join query is an optimization problem, tightly coupled with the predicate

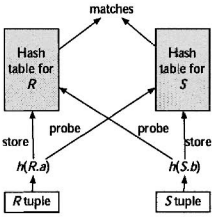


Figure 2.1. The symmetric hash join operator for memory-fitting finite streaming sources.

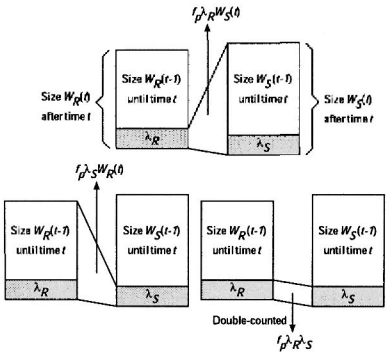


Figure 2.2. A breakdown of the effects taking place for the evaluation of $R \bowtie_p S$ during time-unit t.

being evaluated (for instance, a hash index cannot be used if the predicate is an inequality one) the rates of the incoming streams and the various parameters such as moving a tuple from one memory location to another or comparing two values.

It is interesting to identify the number of output tuples generated due to new arrivals from the input streams. Assume that $R \bowtie_p S$ is evaluated and $W_R(t)$ and $W_S(t)$ are the number of tuples read from R and S respectively after time t. (We are slightly abusing the window notation for uniformity of the expressions.) Given Little's theorem, the number of tuples in each stream up until time t will be $W_R(t) = \lambda_R t$ for R and $W_S(t) = \lambda_S t$ for S. During time-unit t there are λ_R new arrivals from R and λ_S arrivals from S. The λ_R arrivals will join with the $W_S(t)$ tuples, producing $f_p \lambda_R W_S(t)$ partial join results. Correspondingly, the λ_S arrivals will join with the $W_R(t)$ tuples, producing $f_p \lambda_S W_R(t)$ partial join results. The result is not simply the sum of these two quantities however, since, as shown in Figure 2.2, we have double-counted the contribution of the partial join between the new arrivals during time-unit t. That factor is equal to $f_p \lambda_R \lambda_S$ and by subtracting it from the computation we obtain that the number of outputs is $f_p \cdot (\lambda_R W_S(t) + \lambda_S W_R(t) - \lambda_R \lambda_S)$. The previous discussion assumes that we have ever increasing windows. The other possibility is having constant-size, tuple-based windows. If we assume that this is the case, there are two differences: (i) the window sizes are not time-dependent, *i.e.,* they are not a function of time so we can use the notation W_R and W_S, and (ii) for each new arrival from any stream, a tuple that is already in the window expires, *i.e.,* it compromises the window definition. As such, we need not discount any double-counted tuples, so the expected number of outputs becomes $f_p \cdot (\lambda_R W_S + \lambda_S W_R)$.

Finally, we also have to consider the effect of possible windows in the output window of the join. This is rather straightforward as we can view the two windows as two finite constant-sized sources so the expected output and, hence, the output window size is simply the product of the selectivity factor of the join predicate times the cross product of the two window sizes *i.e.,* $f_p W_R W_T$. The complete set of results is summarized in Equations 2.3 and 2.4.

$$\lambda_{R \bowtie_p S} = \begin{cases} f_p \cdot (\lambda_R W_S(t) + & \text{if there are no windows} \\ \quad \lambda_S W_R(t) - \lambda_R \lambda_S) & \\ f_p \cdot (\lambda_R W_S + \lambda_S W_R) & \text{if there are windows} \end{cases} \tag{2.3}$$

$$W_{R \bowtie_p S} = f_p \cdot W_R \cdot W_S \tag{2.4}$$

One important result of sliding window join evaluation comes from the following observation: the evaluation of $R \bowtie_p S$ can be *decomposed* into two independent components: $R \rtimes S$, signifying the operations that need to be carried out over stream R, and $R \ltimes S$, signifying S the operations that need to be carried out over stream S; we call each of those components a *join direction.* It is possible to use a different memory structure for each join direction. For instance, a hash table can be used to store the tuples of R, while a B-tree can be used to store the tuples of S. This lead to the introduction of *asynchronous* operators [Kang et al., 2003].

Let us now derive computational cost expressions for the evaluation process. Since the rates of the inputs are measured in tuples per unit time, we will present *per unit time* [Kang et al., 2003] computational cost expressions. First of all, the computational cost $\mu_{R \bowtie S}$ is equal to the sum of the computational costs in each join direction. Given the previous analysis of operations the per unit time computational cost for $R \rtimes S$ will be equal to the sum of the following factors: (i) the number of S arrivals per unit time multiplied by the cost of accessing R's memory structure for matches; and (ii) the number of R arrivals per unit time multiplied by the cost of storing them in R's memory structure and invalidating an equal number of tuples that have expired from the memory structure.

What seems counterintuitive, is that the computational cost for R tuples is dependent on the arrival rate of tuples from S. The explanation is the following: what essentially "triggers" the evaluation over R's memory structure are arrivals from S; this is the only measure we have of how many times the memory structure for R is searched for matches satisfying the join predicate. The number of storing and invalidation operations, however, is dependent on arrivals from stream R, as expected. The entire computation for both join directions is presented in Equations 2.5 to 2.7.

$$\mu_{R \rtimes S} = \lambda_S \cdot \mu_m(R) + \lambda_R \cdot (\mu_s(R) + \mu_i(R)) \quad (2.5)$$

$$\mu_{R \ltimes S} = \lambda_R \cdot \mu_m(S) + \lambda_S \cdot (\mu_s(S) + \mu_i(S)) \quad (2.6)$$

$$\mu_{R \bowtie S} = \mu_{R \rtimes S} + \mu_{R \ltimes S} \quad (2.7)$$

2.2.2 Finite Sources and Disk-based Operation. The previous discussion addresses the issue of sources that fit in memory. But the question of join evaluation over finite sources whose sizes exceed available memory still remains. The first extension to SHJ to address those issues came in the form of an operator called *XJoin* [Urhan and Franklin, 2000]. Consider again the equi-join $R \bowtie_{R.a=S.b} S$ and assume that the system allocates an amount of memory for R's hash table capable of storing M_R tuples, while the amount of memory allocated for S's hash table is sufficient to store M_S tuples. If $|\cdot|$ denotes the cardinality of a finite stream, $M_R < |R|$ and $M_S < |S|$ hold, *i.e.,* neither stream fits entirely in memory. In XJoin, each hash table is partitioned in m partitions, *i.e.,* partitions $p_1^R, \ldots p_m^R$ for R and partitions $p_1^S, \ldots p_m^S$ for S. A new incoming tuple is hashed twice; once to find the partition it belongs to, and once to find its position in the partition. Join evaluation using the XJoin operator, then, proceeds in three stages:

1 *The streaming memory-to-memory phase*: While the inputs are streaming into the system, XJoin behaves just as SHJ with one difference: as soon as a partition becomes full it is flushed to disk. This phase continues until either one of the inputs becomes blocked or both streams are fully read.

2 *The blocked memory-to-disk phase*: If one stream becomes blocked then an on-disk partition from the other stream is fully read and used to probe the corresponding in-memory partition for matches. Care needs to be taken so that no duplicates are generated at this point, *i.e.,* join results that have been already generated during the first phase are not generated again. This is achieved by checking additional conditions over the tuple timestamps, as well as on the intervals at which on-disk partitions were used for probing in-memory ones.

3 *The clean-up disk-to-disk phase*: After both streams are fully read, XJoin reverts to the final clean-up phase. The behavior during this phase is almost the same as in the second phase of the traditional hybrid hash join [Shapiro, 1986]. The only difference is that again care is taken so that no duplicate join results are generated.

A further extension to XJoin, came in the form of MJoin [Viglas et al., 2003], that addressed the issue of multi-way join query evaluation. Consider a scenario of a multi-way join query over m data streams $S_1, S_2, \ldots, S_m$. A

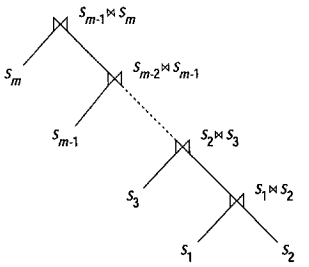


Figure 2.3. A traditional binary join execution tree.

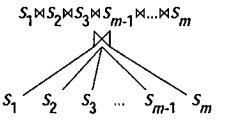


Figure 2.4. A multiple input join operator.

typical evaluation plan employing binary join operators is shown in Figure 2.3. There are two problems in this scenario: First, for each join in the execution tree two hash tables will be built; one of these hash tables will be for a temporary result (*e.g.*, the result of $S_2 \bowtie S_3$). That poses additional processing and storage burdens on the system. Furthermore, a more subtle problem is that the output rate of the query is not only a function of the chosen evaluation algorithm, but also a function of the shape (*e.g.*, deep or bushy) of the execution tree. The MJoin solves these problems by employing a single operator to evaluate the query, as shown in Figure 2.4. The general evaluation algorithm is the same as XJoin's, the only differences being that there are as many hash tables as there are inputs and that not all hash tables will be probed for every arrival, as the sequence of probes stops whenever a probe of a hash table finds no matches (since in this case it cannot produce answer tuples). Each operator has its own probing sequence and the sequence is organized in such a way so that the most selective predicates are evaluated first and it is different for each input. This ensures that the smallest number of temporary tuples is generated. The computational cost expressions for MJoin are quite straightforward extensions of those for binary joins. The main difference is that they need to be generalized so that the probing of not one but $m - 1$ hash tables is taken into account.

Finally, we need to point out that MJoin is not a "panacea" for all multi-way join queries; there are cases in which the processing required by the combined input rates of the streams is such that a single MJoin operator is better off split in a number of fewer input MJoins [Viglas et al., 2003]. Identifying those cases presents an interesting optimization problem.

3. Static Optimization

Cost-based optimization has been traditionally employed as the query optimization paradigm since the seminal work in System R [Selinger et al., 1979]. The shift when it comes to data stream systems as opposed to disk-bound

database systems stems from the optimization metric. The cost metric used by database systems has been *cardinality*, which lead to query optimization being described as *cardinality-based* query optimization. This makes perfect sense for disk-bound data: the dominant factor in determining the cost of a query evaluation strategy is disk I/O and cardinality is a good approximation of how much disk I/O is needed.

But cardinality is not the best cost metric for streams. If they are finite, but remotely accessed, their cardinality may be unknown until they have been fully read, which is of no use to query optimization. Even if cardinality is known at optimization time, the data is not readily available when query execution starts. Even worse, cardinality as a cost metric is not even applicable in the case of unbounded streams. The obvious argument here is that the cost of any alternative evaluation strategy, as estimated by a cardinality-based cost model, for a query over infinite streams is infinite. Clearly, something different is needed.

The three main optimization approaches proposed in the literature are rate-based query optimization, optimization for resource consumption and optimization for quality of service. We will present each approach in turn.

3.1 Rate-based Query Optimization

All the cost expressions we have so far presented make use of the rate of the incoming streams. As such, the rate forms the basis of a new cost model, called the *rate-based* cost model [Viglas and Naughton, 2002]. The idea behind the rate-based cost model and rate-based query optimization is that it captures all the important parameters of processing over streams: the number of tuples per unit time the system is expected to process; the per unit time processing cost; and the throughput of the query. It also provides the basis for the mechanism that allows us to move from he realm of time-based windows to that of tuple-based windows and vice versa.

There are more than one possibilities of what one can optimize for when using a rate-based cost model. The best way to present them is by concentrating on the throughput of any query $\mathcal{Q}$, defined as $\rho(\mathcal{Q}) = \frac{\lambda(\mathcal{Q})}{\mu(\mathcal{Q})}$, where $\lambda(\mathcal{Q})$ is the output rate and $\mu(\mathcal{Q})$ is the total processing cost. The quantity $\frac{1}{\mu(\mathcal{Q})}$ is also referred to as *utilization*. Given any query, there are a number of alternative evaluation plans. If the overall objective is to maximize the throughput of the system, we have two ways of doing so: either by choosing the evaluation plan that maximizes the output rate of the query, or by choosing the plan that maximizes utilization (*i.e.,* minimizes processing).

In some cases, only achieving one of those objectives is enough. For instance, in the case of sliding window queries, and in steady state, the expected number of outputs will be the same regardless of the choice of plan [Ayad and Naughton,

2004]. The argument is quite simple with the following analogy: once the window sizes are fixed, we have the same situation as in evaluating a query over disk-based data: the output cardinality (and, hence, the rate) will be the same regardless of the choice of plan. This is not the case, however, if the sources are finite and if part of the processing is disk-based. The rate can then drastically change over time as a function of which execution plan is chosen.

An interesting extension is what happens if the output rate of a query is time-dependent. Consider the case of query $\mathcal{Q}$ and a set of m possible evaluation plans for that query $\mathcal{P}_i, i = 1, \ldots, m$. Assume that the time-dependent output rate of each plan can be expressed as a function of time $\lambda_{\mathcal{P}_i}(t)$ (with obvious semantics). The number of outputs produced by plan $\mathcal{P}_i$ will be another function of time, which is shown in in Equation 2.8.

$$n_{\mathcal{P}_i}(t) = \int_0^t \lambda_{\mathcal{P}_i}(t)dt \tag{2.8}$$

Given the correlation between the number of outputs $n_{\mathcal{P}_i}(t)$ produced by time t by plan $\mathcal{P}_i$ and the estimated output rate of the plan $\lambda_{\mathcal{P}_i}(t)$, shown in Equation 2.8, it is now possible to optimize for two objectives. By fixing the time variable to some constant t_p, we can choose the plan that maximizes the number of outputs produced in that time interval. Alternatively, by treating time as a variable and fixing the number of output tuples to $n_{\mathcal{Q}}$, we can choose the plan $\mathcal{P}_i$ that minimizes the time needed to generate the specified number of outputs.

This is the theoretical formulation of the problem. In practice, the general optimization problem is reduced to a *local* optimization problem, using integral approximation techniques and heuristics. Two such reductions for each of the optimization objectives [Viglas and Naughton, 2002] are: (i) *local rate maximization*, in which the local output rates are maximized in hopes of maximizing the total output rate and, therefore, the number of tuples produced in a given time period; (ii) *local time minimization*, in which the time needed to reach partial result sizes is minimized in hopes of minimizing the time needed to reach the complete result size.

3.2 Resource Allocation and Operator Scheduling

The previous framework of rate-based query optimization assumes that the streams have a predictable arrival rate that can be characterized by an average number of arrivals per unit time. Research in data networks, however, postulates that network traffic may exhibit periods of high activity (bursts) and periods of lighter load. The effect this has on query evaluation is that during periods of high activity there may appear backlogs of unprocessed tuples between operators of a static plan. These backlogs are usually stored in queues

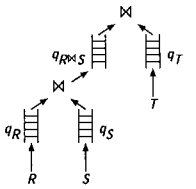


Figure 2.5. An execution plan in the presence of queues; q_S denotes a queue for stream S.

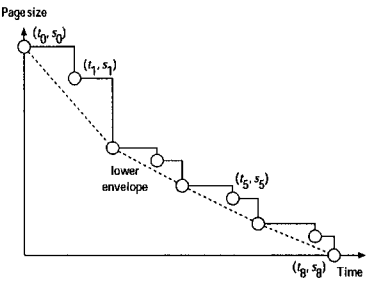


Figure 2.6. Progress chart used in Chain scheduling.

between operators, as shown in Figure 2.5. The problem then becomes one of scheduling: the execution system should schedule operators in a way that minimizes the length of those queues. In other words, what needs to be identified is the optimal scheduling strategy so that the total memory requirements of a query are minimized.

In doing so, we need a way to characterize memory requirements in terms of queue sizes of the various operators in a query. We assume that a paged memory approached is used, *i.e.,* a page is the unit of memory allocation and streams arrive in a paged fashion into the system. The memory requirements are captured by a *progress chart* [Babcock et al., 2003], an example of which is shown in Figure 2.6. The x-axis of the progress chart represents time, while the y-axis represents the number of pages used by a query to store its tuples in queues. The points (shown as circles) of the graph represent the operators of the query with the following semantics: if there are m operators in the query, there will be $m+1$ points in the graph. The i^{th} point represents an operator that takes $t_i - t_{i-1}$ time units to process s_{i-1} input pages, producing s_i pages in the output. The selectivity factor of the operator can therefore be defined as $\frac{s_i}{s_{i-1}}$. In the end, there will be zero pages to process, so S_m will always be equal to zero.

The progress of a page (*i.e.,* a memory unit) through the system is captured by a *progress line*, which is the "jagged" solid line between points in Figure 2.6. Pages are expected to move along this progress line. A new page p enters the system at time t_0 and has a size of s_0. After having been processed by operator O_i it has received t_i processor time and its size has been reduced to s_i. The last operator in the query plan always reduces the page size to zero, as the page no longer needs to be buffered in a queue.

For the memory requirements of the plan to be minimized, we need to be along the dashed line. The reason is that operators along this line reduce the

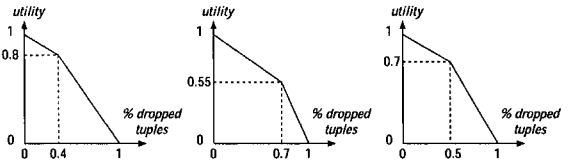


Figure 2.7. Example utility functions; the x-axis is the percentage of dropped tuples, while the y-acis is the achieved utility.

number of pages in the system the fastest. This dashed line is called the *lower envelope simulation* of the progress chart. This observation gives rise to the *chain scheduling* strategy [Babcock et al., 2003]: At any time instant, consider all pages that are currently in the system. Of these, schedule for a single time unit the page that lies on the segment with the steepest slope in its lower envelope simulation. If there are multiple such pages, select the page which contains the earliest arrival.

Chain scheduling is provably the optimal scheduling strategy in the presence of knowledge of the selectivity factors and per-tuple (and therefore per-page) processing costs. The cost expressions of the algorithms we have presented in the previous sections allow us to estimate those costs. As for the selectivity factors, they can be periodically re-estimated by a simple sampling methodology. After each re-estimation, the progress chart is recomputed and the operators are rescheduled in accordance to the new progress chart.

3.3 Quality of Service and Load Shedding

One of the potential problems of evaluating queries over streaming sources arises when, in order to cope with the combined input rates of the incoming sources, the processing required by the evaluation plan so that it "keeps up" with the incoming rates is more than what the CPU can devote. This is an instance of an *infeasible* plan, *i.e.,* a plan that saturates the CPU so that the queues of unprocessed tuples that appear between operators grow without limit.

Since no execution schedule can minimize the length of the queues, the only alternative is to "artificially" decrease the length of the queue by "dropping" some of the tuples in it. This is called *load shedding* [Kang et al., 2003; Tatbul et al., 2003; Ayad and Naughton, 2004]. However, choosing where tuples should be dropped from is not something to be performed blindly. This is performed on a Quality of Service (QoS) basis with respect to a *utility function.* The utility function is a *concave* function of the percentage of tuples dropped. More specifically, if no tuples are dropped, the utility of the evaluation plan is one; as the percentage of dropped tuples grows, the utility decreases until it reaches zero. Examples of utility functions are shown in Figure 2.7.

The entire problem is an optimization problem, which can be stated as follows: Consider a query $\mathcal{Q}$, an available total processing capacity $\mathcal{M}$, an evaluation plan $\mathcal{P}$ for that query and a concave utility function $u(\mathcal{P})$. The total processing required by the plan is $\mu(\mathcal{P})$ where $\mu(\mathcal{P}) > \mathcal{M}$, *i.e.,* the plan is infeasible. The objective is to identify an alternative plan $\mathcal{P}'$ that performs load shedding so that $\mu(\mathcal{P}') \leq \mathcal{M} < \mu(\mathcal{P})$ and $u(\mathcal{P}) - u(\mathcal{P}')$ is minimized.

The conceptual solution to the problem is the introduction of *drop-boxes* in the query plan. These can be thought of as operators inserted into the execution plan between successive operators, or between an incoming stream and the first operator to process it. The function of these operators is to selectively drop a percentage of incoming tuples before they reach the subsequent operator.

Introducing drop-boxes in the execution plan leads to another subtle decision that needs to be made. Not only do we need to determine how much load each drop-box should shed, we also need to determine the location of these drop-boxes in the query plan. The solution to both these problems makes use of Loss/Gain ratio modeling [Tatbul et al., 2003]. A Loss/Gain ratio is the ratio between the loss in utility incurred by the introduction of a drop-box, over the gain in QoS. Each alternative drop-box location is associated with a Loss/Gain ration. Once all drop locations are enumerated and their Loss/Gain ratios are computed, they are sorted in Loss/Gain ratio ascending order. The system then iterates over each location adding drop-boxes. After the addition of a drop box, new Loss/Gain ratios are computed, since the introduction of a drop box may affect them. The iteration stops once the optimization objective is met.

An important decision is how tuples are dropped once the drop-boxes are in place. The simplest method is, of course, *random dropping*. That is, once the percentage of tuples to be dropped is identified, these tuples are randomly dropped from the input, so long as the dropped tuples percentage quota is met. An alternative approach, and one that has been proven to work better in practice, is *semantic dropping*. When employing semantic dropping, the drop-box effectively becomes a selection operator evaluating a predicate on the values of the incoming stream. The utility of each tuple is computed, based on the tuple's values. The semantic predicate of the selection operator then becomes one that drops the least useful tuples, propagating only those having a higher utility [Tatbul et al., 2003].

One important aspect of load shedding is that it is independent of the scheduling algorithm. The assumption is that load shedding "saves" processing cycles by gracefully bringing the execution plan in the realm of feasible computation. Once the plan becomes feasible, the scheduler can go about doing its usual business of scheduling the operators so that intermediate queue sizes are minimized.

4. Adaptive Evaluation

The discussion on query execution and optimization so far revolves around the idea that the execution environment is relatively static. As such, the assumptions made at optimization-time will hold at execution-time. In network-bound data processing, however, it also makes sense to account for highly dynamic and unpredictable environments. In those cases, the execution environment is expected to drastically change over time due to reasons beyond the system's control, such as bursty arrivals.

All these approaches are static in the sense that once the execution plan (or operator) is set up, it does not change over time. An entirely different approach is to use an *adaptive* framework. The key idea of adaptive systems is that they are able to gradually re-organize the execution plan over time so that it is always close to the optimal plan for the current state of the execution environment. In the next sections we will refer to two adaptability ideas: *Query Scrambling* and *Eddies* and *Stems*.

4.1 Query Scrambling

Query scrambling [Urhan et al., 1998] is a reactive approach to address the unpredictability in the rates of the incoming streams. Optimization approaches like rate-based optimization may choose a plan that is truly optimal if the optimization time assumption of an average incoming rate with little fluctuation is true. Under bursty traffic, however, though the expected average may still be valid, the exhibited performance may be quite different than the expected one simply because of bigger fluctuations in the rate. In those cases, query scrambling "steps in" and dynamically alters the plan so that these issues are resolved. In the next section we will see how this is achieved.

Consider for instance the distributed execution plan of Figure 2.8, where a five-way join query over four remote sites is presented. The problem if there is no room for dynamic plan restructuring is that if one of the inputs exhibits delays, then the whole plan will experience the same delay. This situation is even worse for initial delays. Consider the scenario in which the network link between Site 1 and Site 4 is down; this means that if the operators at Site 1 are chosen to be executed first, then query execution cannot even be initialized as there will be no arrivals at all from Site 1 to the result producing Site 4.

Query scrambling addresses situations like these are addressed by employing a two-phase approach:

- *First phase: Query rescheduling.* The order in which operators are executed in the query plan is dynamically altered. In the example of Figure 2.8, if the network link between Site 1 and Site 4 is down, the operators at sites other than Site 1 can execute and transmit results to Site 4 regardless of the network link between Site 1 and Site 4 being down. The query

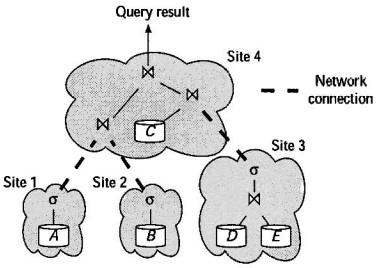


Figure 2.8. A distributed query execution tree over four participating sites.

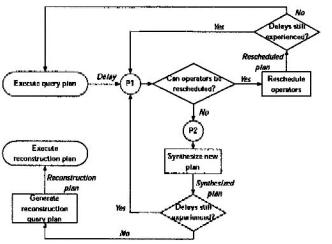


Figure 2.9. The decision process for query scrambling; the initiation of the scrambling phases is denoted by 'P1' for the first one and 'P2' for the second one.

plan does not change, but the system reschedules the evaluation order so these operators are executed first.

- *Second Phase: Operator synthesis.* As the name suggests, new operators are introduced in the query plan. For instance, a new join might be added to the query plan with another join removed. This means that the query plan is significantly different after operators are added or removed.

These two approaches are invoked in sequence, but iteratively by the system. That is, the system first tries to address potential problems by iteratively employing query rescheduling. After no progress can be made using this approach, it reverts to the more drastic approach of iteratively employing operator synthesis. The processing flow during query scrambling is depicted in Figure 2.9.

4.2 Eddies and Stems

The idea behind an *Eddy* [Avnur and Hellerstein, 2000] is to reduce query optimization to a routing problem. This is achieved by the very elegant idea of not explicitly connecting query operators to form an execution plan; instead, there is a central authority that undertakes the task of routing tuples to the appropriate operator as they enter the system. Once a tuple has been processed by all query operators, it is ready to exit the system as a result tuple. In this representation, the central authority is called an *Eddy*.

On the other hand, a *Stem* [Madden et al., 2002] is an on-the-fly index built on top of the incoming streams, depending on the query at hand (*e.g.,* a Stem might be a hash index or a B-tree). The query operators access the appropriate index. It is possible for the system to build two Stems over the same stream. For instance, given an input stream S, a hash index might be built on attribute $S.a$ and a B-tree index might be built on attribute $S.b$. The hash index can then

be used to evaluate an equi-join where $S.a$ is the participating attribute and the B-tree index can be used for a range selection over $S.b$. Moreover, indexes can be shared among queries. In the previous example, any other query needing to perform a range selection over $S.b$ can use the B-tree index.

Finally, the system creates an instance of every operator in the query. The operators communicate directly with the Eddy and may access one or more Stems depending on the operation they evaluate. The operators are not connected in any other sense, *i.e.*, if there are operators O_i and O_j, they will never exchange any tuples directly; all communication and information exchange will take place through the Eddy. Each tuple is augmented with a *query bitmap*, which contains as many bits as there are operators in the query. All bits are initialized to zero and are set to one after the corresponding operator has processed the tuple. After all bits have been set the tuple is output as a result tuple. The sequence of processing steps for a tuple, from its entry into the system until its exit, can therefore be summarized as follows:

1 On arrival to the Eddy, the tuple is augmented with the query bitmap (all bits set to zero) and propagated to the appropriate Stem; it is then returned to the Eddy.

2 On re-acceptance of a tuple, the Eddy decides which operator the tuple should be sent to next. The operator accepts the tuple and processes it by accessing the appropriate Stem(s). If the operator decides the tuple should still be in the system, it sets the appropriate bit of the query bitmap, and returns it to the Eddy. One example of a tuple being dropped out of further processing is when it does not satisfy a selection predicate.

3 The previous step is repeated until all operators have processed the tuple (*i.e.*, all the bits in the query's bitmap are set).

Consider for example a three-way query with the following relational algebraic operators: $\sigma_{S_1.a_3>v}$, $S_1 \bowtie_{S_1.a_1=S_2.a_2} S_2$, $S_1 \bowtie_{S_1.a_3>S_3.a_3} S_3$. The use of an Eddy and four Stems for the evaluation of the three-way join query is presented in Figure 2.10.

Tuple Routing. The "heart" of an Eddy is the principle under which tuples are routed from one operator to another. This is called the *routing strategy* of the adaptive framework. The premise is that a good routing strategy will converge to the optimal evaluation plan at all times.

Eddies use a variant of the lottery scheme tailored for the query processing paradigm. In this scenario, "winning the lottery" means "accepting a tuple for processing." The concept is that the probability of the Eddy routing a tuple to an operator (and, therefore, the operator winning the lottery) is directly proportional to the number of tickets the operator holds. The operator that

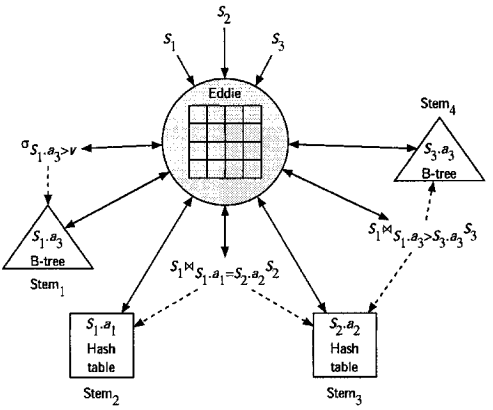


Figure 2.10. Combination of an Eddy and four Stems in a three-way join query; solid lines indicate tuple routes, while dashed lines indicate Stem accesses used for evaluation.

wins the lottery at each iteration is the one that holds the greatest number of tickets. Not all operators participate in every draw; only those that are eligible. Eligibility can be determined by the bitmap of a tuple: the eligible operators are those for which the corresponding bit in the tuple's query bitmap is not yet set. Once a tuple is routed to an operator, the operator receives a ticket. If it returns the tuple to the Eddy, it returns the ticket as well; if it does not, it holds on to the ticket. Operators that drop tuples out of further processing hold on to the greatest number of tickets. These are the most selective operators and most of the tuples should be routed to them. If the selectivity factor of a predicate increases over time the operator will slowly start returning the tickets it is holding on to. In that way, the system will adapt to a different evaluation plan.

5. Summary

Query execution and optimization over data streams revisits almost all aspects of query evaluation as these are known from traditional disk-bound database systems. Data stream systems can be thought of as database systems after two traditional assumptions have been dropped: (i) the data is disk-bound and stored on a local disk, and (ii) the data is finite.

When dealing with query evaluation over data streams there are two main approaches: static and adaptive. Systems adhering to the static approach employ the same paradigm as traditional systems in terms of query evaluation, in the sense that they statically optimize a query to identify an optimal execution plan, and then use that plan for the lifetime of the query. The challenge is identifying the new cost metrics that are better suited for query evaluation over

streaming sources, as well as devising evaluation algorithms that perform better over streams (with respect to the cost metrics).

Systems employing the adaptive approach treat the execution environment as entirely unpredictable, in the sense that any assumption made when the query is issued and statically optimized will fail during query evaluation. As such, the system only reacts to problems that may arise during query execution. One approach is to continuously restructure the execution plan This process is called query scrambling. An alternative is not to employ an explicit execution plan, but be completely adaptive by focussing only on the operators that need to be evaluated without connecting them in a static plan. Adaptation is achieved by having a central routing authority route tuples to operators. The best known example of this approach is the combination of Eddies and Stems.

References

Avnur, Ron and Hellerstein, Joseph M. (2000). Eddies: Continuously Adaptive Query Processing. In *SIGMOD Conference.*

Ayad, Ahmed and Naughton, Jeffrey F. (2004). Static Optimization of Conjunctive Queries with Sliding Windows over Unbounded Streaming Information Sources. In *SIGMOD Conference.*

Babcock, Brian, Babu, Shivnath, Datar, Mayur, and Motwani, Rajeev (2003). Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *SIGMOD Conference.*

Bertsekas, D. and Gallager, R. (1991). *Data Networks.* Prentice Hall.

Golab, Lukasz and Ozsu, M. Tamer (2003). Processing sliding window multi-joins in continuous queries over data streams. In *VLDB Conference.*

Kang, Jaewoo, Naughton, Jeffrey F., and Viglas, Stratis D. (2003). Evaluating Window Joins over Unbounded Streams. In *Proceedings of the International Conference on Data Engineering (ICDE).*

Madden, Sam, Shah, Mehul A., Hellerstein, Joseph M., and Raman, Vijayshankar (2002). Continuously Adaptive Continuous Queries over Streams. In *SIGMOD Conference.*

Selinger, Patricia G., Astrahan, Morton M., Chamberlin, Donald D., Lorie, Raymond A., and Price, Thomas G. (1979). Access Path Selection in a Relational Database Management System. In *SIGMOD Conference.*

Shapiro, Leonard D. (1986). Join Processing in Database Systems with Large Main Memories. *TODS*, 11(3):239–264.

Tatbul, Nesime, Cetintemel, Ugur, Zdonik, Stanley B., Cherniack, Mitch, and Stonebraker, Michael (2003). Load Shedding in a Data Stream Manager. In *VLDB Conference.*

Urhan, Tolga and Franklin, Michael J. (2000). XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33.

Urhan, Tolga, Franklin, Michael J., and Amsaleg, Laurent (1998). Cost-Based Query Scrambling for Initial Delays. In *SIGMOD Conference.*

Viglas, Stratis D. and Naughton, Jeffrey F. (2002). Rate-Based Query Optimization for Streaming Information Sources. In *SIGMOD Conference.*

Viglas, Stratis D., Naughton, Jeffrey F., and Burger, Josef (2003). Maximizing the Output Rate of Multi-Way Join Queries over Streaming Information Sources. In *VLDB Conference.*

Wilschut, Annita N. and Apers, Peter M. G. (1991). Pipelining in Query Execution. In *Conference on Databases, Parallel Architectures and their Applications.*

Chapter 3

FILTERING, PUNCTUATION, WINDOWS AND SYNOPSES

David Maier,[1] Peter A. Tucker,[2] and Minos Garofalakis[3]

[1] *OGI School of Science & Engineering at OHSU*
20000 NW Walker Road
Beaverton, OR 97006
maier@cse.ogi.edu

[2] *Whitworth College*
300 W Hawthorne Road
Spokane, WA 99251
ptucker@cse.ogi.edu

[3] *Bell Laboratories,*
Lucent Technologies
Murray Hill, NJ 07974
minos@research.bell-labs.com

Abstract This chapter addresses some of the problems raised by the high-volume, non-terminating nature of many data streams. We begin by outlining challenges for query processing over such streams, such as outstripping CPU or memory resources, operators that wait for the end of input and unbounded query state. We then consider various techniques for meeting those challenges. *Filtering* attempts to reduce stream volume in order to save on system resources. *Punctuations* incorporate semantics on the structure of a stream into the stream itself, and can help unblock query operators and reduce the state they must retain. *Windowing* modifies a query so that processing takes place on finite subsets of full streams. *Synopses* are compact, efficiently maintained summaries of data that can provide approximate answers to particular queries.

Keywords: data stream processing, disordered data, stream filtering, stream punctuation, stream synopses, window queries.

1. Introduction: Challenges for Processing Data Streams

The kinds of manipulations users would like to perform on data streams are reminiscent of operations from database query processing, OLAP and data mining: selections, aggregations, pattern-finding. Thus, one might hope that data structures and algorithms developed for those areas could be carried over for use in data stream processing systems. However, existing approaches may be inadequate when confronted with the high-volume and unbounded nature of some data streams, along with the desire for near-real time results for stream operations.

The data rate for a stream might outstrip processing resources on a steady or intermittent (bursty) basis. Thus extensive CPU processing or secondary storage access for stream elements may be infeasible, at least for periods of time. Nor can one rely on buffering extensive amounts of the stream input in memory. For some applications, such as network monitoring, a few seconds of input may exhaust main memory. Furthermore, while buffering might handle bursty input, it does so at a cost of delaying results to users.

The potentially unbounded nature of data streams also creates problems for existing database query operators, or for particular implementations of them. Blocking operators, such as group-by, difference and sort, and blocking implementations, such as most join algorithms, cannot in general emit any output until the end of one or more of the inputs is reached. Thus on a continuous data stream, they will never produce any output. In the case of join, there are alternative implementations, such as symmetric hash join [Wilschut and Apers, 1991] that are non-blocking, and hence more suitable for use with streams. But the other operators mentioned are inherently blocking for any implementation. Even if an operator has a non-blocking implementation, if it is stateful, such as join and duplicate elimination, it will accumulate state without limit, eventually becoming memory bound.

Thus, extensions or alternatives for current query processing and data analysis techniques are needed for streams. In this chapter, we survey several approaches to these challenges, based on data reduction, exploiting semantics of data streams, and approximation. We first cover exact and lossy filtering techniques, which attempt to reduce data stream volumes early in the processing chain, in order to reduce the computational demands on later operations. We then consider the use of stream "punctuation" to incorporate knowledge about the internal structure in a data stream that might be useful in unblocking operators or limiting the amount of state that must be retained. We then consider "windowed" versions of classical operators, which can be viewed as a continuous user query being approximated by a series of queries over finite subsequences of an unbounded stream. In this context we also briefly consider issues with disordered inputs. The final class of techniques we cover are syn-

opses, which in the stream case can be considered as representations of data streams that a) summarize the stream input, b) can be maintained online at stream input rates, c) occupy much less space than the full data, and d) can be used to provide exact or approximate answers to some class of user queries.

2. Stream Filtering: Volume Reduction

Faced with stream volumes beyond what available resources allow processing in their entirety, a stream processor can simply abort, or somehow reduce the volume to a manageable level. Such reduction can take several forms: precise filtering, data merging, or data dropping.

2.1 Precise Filtering

Precise filtering extracts some portion of a stream query for application nearer the stream source, with the expectation of reducing stream volume while not changing the final query answer. Filtering operations generally need to be simple, such as selection or projection, and applicable on an item-by-item basis, so as not to consume extensive processing cycles or memory. Filtering should also avoid introducing long delays into time-critical data streams. This filtering may happen at the stream source, near the stream-processing system, or in between.

A source may support subscription to a substream of the full data stream. For example, the Virtual Object Ring Buffer (VORB) facility of the RoadNet project [Rajasekar et al., 2004] supports access to real-time information from an environmental-sensing network. A VORB client can request a substream of this information restricted on (geographic) space, time and attribute. Financial feeds also support filtering, such as on specific stocks or currencies.

[Hillston and Kloul, 2001] describe an architecture for an online auction system where active network nodes serve as filters on the bid stream. Such a node can filter out any bid for which a higher bid has already been handled for the same item. (It is also possible that highest bid information is periodically disseminated from the central auction server to the active network nodes, as otherwise an active node is only aware of bid values that it handles.) Such processing is more complex than item-at-a-time filtering. It essentially requires an anti-semijoin of incoming bids with a cache of previous bids. However, the space required can be reduced from what is required for a general semijoin by two considerations. First, only one record needs to be retained for each auction item – the one with the maximum price so far. (Actually, just the item ID and bid price suffice.) Second, the cache of previous bids does not need to be complete – failure to store a previous bid for an item only means that later items with lower prices are not filtered at the node. Thus an active node can devote a

bounded cache to past information, and select bids to keep in the cache based on recency or frequency of activity on an item.

Gigascope [Johnson et al., 2003] is a stream-processing system targeted at network monitoring and analysis. It supports factoring of query conditions that can be applied to the raw data stream arriving at the processor. These conditions can be applied in the network interface subsystem. In some versions, these filter conditions are actually pushed down into a programmable network interface card (NIC).

2.2 Data Merging

Data merging seeks to condense several data items into one in such a way that the ultimate query can still be evaluated. Consider a query that is computing the top-5 most active network flows in terms of bytes sent. (Here a flow is defined by a source and destination IP address and port number.) Byte-count information for packets from the same flow can be combined and periodically transferred to the stream-processing system. This approach is essentially what routers do in generating Netflow records, [Cisco Systems, 2001] , reducing the volume of data that a network monitoring or profiling application needs to deal with. Of course, only certain queries on the underlying network traffic will be expressible over the aggregated Netflow records. A query looking for the most active connections is expressible, but not an intrusion-detection query seeking a particular packet signature. Merging can be viewed as a special case of synopsis. (See Section 6.)

2.3 Data Dropping

Data dropping (also called load shedding) copes with high data rates by discarding data items from the processing stream, or limiting the processing of selected items. Naïve dropping happens in an uncontrolled manner – for example, items are evicted without processing from an overflowed buffer. More sophisticated dropping schemes introduce some criterion that identifies which data items to remove, based for example, on the effect upon the accuracy of the answer or an attempt to get a fair sample of a data stream.

The simplest approaches can be termed *blind* dropping: the decision to discard a data item is made without reference to its contents. In the crudest form, blind dropping discards items when CPU or memory limits are exceeded: Data items are dropped until the stream-processing system catches up. Such a policy can be detrimental to answer quality, with long stretches of the input being unrepresented. Better approaches attempt to anticipate overload and spread out the dropped data items, either randomly or uniformly. For example a VORB client can throttle the data flow from a data source, requesting a particular rate for data items, such as 20 per minute.

Dropping can take place at the stream source, at the leaves of a stream query, or somewhere in the middle of a query plan. The Aurora data stream manager provides an explicit `drop` operator that may be inserted at one or more places in a network of query operators [Tatbul et al., 2003]. The `drop` can eliminate items randomly or based on a predicate (which is termed *semantic* dropping). Another approach to intra-query dropping is the modification of particular operators. [Das et al., 2003] and [Kang et al., 2003] present versions of window join (see Section 4) that shed load by either dropping items or avoiding the join of particular items.

Whatever mechanism is used for dropping data items, key issues are determining how much to drop and maximizing answer quality for a given drop rate. The Aurora system considers essentially all placements of `drop` operators in an operator network (guided by heuristics) and precomputes a sequence of alternative plans that save progressively more processing cycles, called a *load-shedding road map* (LSRM). For a particular level of cycle savings, Aurora selects the plan that maximizes the estimated quality of service (QoS) of the output. QoS specifications are provided by query clients and indicate, for example, how the utility of an answer drops off as the percentage of full output decreases, or which ranges of values are most important. The two window-join algorithms mentioned above attempt to maximize the percentage of join tuples produced for given resource limits. Das et al. point out that randomized dropping of tuples in a join can be ineffective by this measure. Consider a join between r tuples and s tuples on attribute A. Any resources expended on an r tuple with $r.A = 5$ is wasted if the only s tuple with $s.A = 5$ has been discarded. They instead collect statistics on the distribution of join values, and retain tuples that are likely to contribute to multiple output tuples in the join. Kang et. al. look at how to maximize output of a window join given limitations on computational or memory resources. They demonstrate, for example, with computational limits, the operator should favor joining in the direction of the smaller window to the larger window. For limited memory, however, it is better to allocate that resource to storing tuples from the slower input.

There are tensions in intelligent data dropping schemes, however. On one hand, one would like to select data items to discard carefully. However, a complicated selection process can mean more time is spent selecting a data item to remove than is saved by removing it. Similarly, the value of a data item in the answer may only be apparent after it passes through some initial operators. For example, it might be compared to frequent data values in a stream with which it is being joined. However, discarding a data item in the middle of a query plan means there are "sunk costs" already incurred that cannot be reclaimed.

2.4 Filtering with Multiple Queries

For any of the filtering approaches – precise filtering, data merging and data dropping – the situation is more complicated in the (likely) scenario that multiple queries are being evaluated over the data streams. Now, the combined needs of all the queries must be met. With precise filtering, for example, the filter condition will need to be the union of the filters for the individual queries, which means the processing of the raw stream may be more complex, and the net reduction in volume smaller. In a semantic data-dropping scheme, there may be conflicts in that the least important data items for one query are the most important for another. (In the multi-query case, Aurora tries to ensure different users receive answers of approximately equal utility according to their QoS specifications.)

3. Punctuations: Handling Unbounded Behavior by Exploiting Stream Semantics

Blocking and stateful query operators create problems for a query engine processing unbounded input. Let us first consider how a traditional DBMS executes a query plan over bounded data. Each query operator in the plan reads from one or more inputs that are directly beneath that operator. When all data has been read from an input, the operator receives an end of file (EOF) message. Occasionally a query operator will have to reread the input when it receives EOF (e.g., a nested-loops join algorithm). If not, the query operator has completed its work. A stateful query operator can purge its state at this point. A blocking operator can output its results. Finally, the operator can send the EOF message to the next operator along in the query plan.

The EOF message tells a query operator that the end of the entire input has arrived. What if a query operator knew instead that the end of a subset of the input data set had arrived? A stateful operator might purge a subset of the state it maintains. A blocking operator might output a subset of its results. An operator might also notify the next operator in the query plan that a subset of results had been output. We will explain how "punctuations" are included in a data stream to convey knowledge about ends of data subsets.

For example, suppose we want to process data from a collection of environmental sensors to determine the maximum temperature each hour using a DBMS. Since data items contain the time they were emitted from the sensor, we can assume that data from each sensor is sorted (non-decreasing) on time. In order to calculate the maximum temperature each hour from a single sensor, we would use the following query (in SQL):

```
SELECT MAX(temp)
FROM sensor
GROUP BY hour;
```

Unfortunately, since group-by is blocking and the input is unbounded, this query never outputs a result. One solution is to recognize that hour is non-decreasing. As data items arrive, the group-by operator can maintain state for the current hour. When a data item arrives for a new hour, the results for the current hour can be output, and the query no longer blocks.

This approach breaks down when the input is not sorted. Even in our simple scenario, data items can arrive out-of-order to the group-by operator for various reasons. We will discuss disorder in data streams in Section 5. By embedding punctuations into the data stream and enhancing query operators to exploit punctuations, the example query will output results before receiving an EOF, even if data arrive out-of-order.

3.1 Punctuated Data Streams

A *punctuation* is an item embedded into a data stream that denotes the end of some subset of data [Tucker et al., 2003]. At a high level, a punctuation can be seen as a predicate over the data domain, where data items that pass the predicate are said to *match* the punctuation. In a *punctuated stream*, any data item that matches a punctuation will arrive before that punctuation. Given a data item d and a punctuation p, we will use *match(d,p)* as the function that indicates whether a d matches p.

The behaviors exhibited by a query operator when the EOF message is received may also be partially performed when a punctuation is received. Clearly, EOF will not arrive from unbounded inputs, but punctuations break up the unbounded input into bounded substreams. We define three kinds of behaviors, called *punctuation behaviors*, to describe how operators can take advantage of punctuations that have arrived. First, *pass behavior* defines when a blocking operator can output results. Second, *keep behavior* defines when a stateful operator can release some of its state. Finally, *propagate behavior* defines when an operator can output punctuations.

In the environmental sensor example, data output from each sensor are sorted on time. We can embed punctuations into the stream at regular intervals specifying that all data items for a particular prefix of the sorted stream have arrived. For example, we can embed punctuations at the end of each hour. This approach has two advantages: First, we do not have to enhance query operators to expect sorted input (though we do have to enhance query operators to support punctuations). Second, query operators do not have to maintain sorted output.

3.2 Exploiting Punctuations

Punctuation behaviors exist for many query operators. Non-trivial behaviors are listed in Tables 3.1, 3.2 and 3.3. The pass behavior for group-by says that results for a group can be output when punctuations have arrived that match all

Table 3.1. Non-trivial pass behaviors for blocking operators, based on punctuations that have arrived from the input(s).

Group-by	Groups that match punctuations that describe the grouping attributes.
Sort	Data items that match punctuations that have arrived covering all possible data items in a prefix of the sorted output.
Difference (S_1-S_2)	Data items in S_1 that are not in S_2 and match punctuations from S_2.

Table 3.2. Non-trivial propagation behaviors for query operators, based on punctuations that have arrived from the input(s).

Select	All punctuations.
Dupelim	All punctuations.
Project$_A$	The projection of A on punctuations that describe the projection attributes.
Group-by	Punctuations that describe the group-by attributes.
Sort	Punctuations that match all data in a prefix of the sorted output.
Join	The result of joining punctuations that describe the join attributes.
Union	Punctuations that equal some punctuation from each other inputs.
Intersect	Punctuations that equal some punctuation from each other inputs.
Difference	Punctuations that equal some punctuation from each other inputs.

possible data items that could participate in that group. The keep behavior for group-by says that state for a group can be released in similar circumstances. Finally, the propagate behavior for group-by says that punctuations that match all possible data items for a group can be emitted (after all results for that group have been output). For example, when group-by receives the punctuation marking the end of a particular hour, the results for that hour may be output, state required for that hour can be released, and a punctuation for all data items with that hour can be emitted. Notice that ordering of data items on the hour attribute does not matter. Even if data arrives out of order, as long as the punctuation correctly denotes the end of each hour, the results will still be accurate.

Many query operators require specific kinds of punctuations. We saw above that the pass behavior for group-by was to output a group when punctuations had arrived that matched all possible data items that can participate in that group. A set of punctuations P *describes* a set of attributes A if, given specific values for A, every possible data item with those attribute values for A matches some punctuation in P. For example, punctuations from the environment sensors that denote the end of a particular hour describe the hour attribute, since they match all possible data items for a particular hour.

Table 3.3. Non-trivial keep behaviors for stateful query operators, based on punctuations that have arrived from the input(s).

Dupelim	Data items that do not match any punctuations received so far.
Group-by	Data items that do not match punctuations describing the grouping attributes.
Sort	Data items that do not match any punctuations covering all data items in the prefix of the sorted output defined in the pass behavior.
Join	Data items that do not match any punctuations from the other input that describe the join attributes.
Intersect	Data items that do not match any punctuations from the other input.
Difference	Data items that do not match any punctuations from the other input.

Table 3.4. Punctuation patterns.

Pattern	Representation	Match Rule
wildcard	*	All values.
constant	c	The value c.
list	$\{c_1, c_2, \ldots\}$	Any value c_i in the list.
range	(c_1, c_2)	Values greater than c_1 and less than c_2.

3.3 Using Punctuations in the Example Query

Suppose in the environmental sensor example each sensor unit outputs data items that contain: sensor id, temperature, hour, and minute. Thus an example stream from sensor 3 might contain: $[< 3, 75, 1, 15 >, < 3, 78, 1, 30 >, < 3, 75, 1, 45 >, < 3, 76, 2, 0 >, < 3, 75, 2, 15 >, \ldots]$. We would like to have the sensors emit punctuations that denoted the end of each hour, to unblock the group-by operator. We treat punctuations as stream items, where punctuations have the same schema as the data items they are matching and each attribute contains a pattern. Table 3.4 lists the patterns an attribute in a punctuation can take.

We want punctuations embedded into the data stream denoting the end of data items for a specific hour. One possible instantiation of such a stream might be (where the punctuation is prefixed with P): $[< 3, 75, 1, 15 >, < 3, 78, 1, 30 >, < 3, 75, 1, 45 >, < 3, 76, 2, 0 >, P < *, *, 1, * >, < 3, 75, 2, 15 >]$. All data items containing the value 1 for hour match the punctuation.

How will punctuations that mark the end of each hour help our example query, where we take input from many sensors? We examine each operator in turn. Suppose our query plan is as in Figure 3.1, and each sensor emits punctuations at the end of an hour. As data items arrive at the union operator, they are immediately output to the group-by operator. Note that union does not attempt to enforce order. Due to the propagation invariant for union, however,

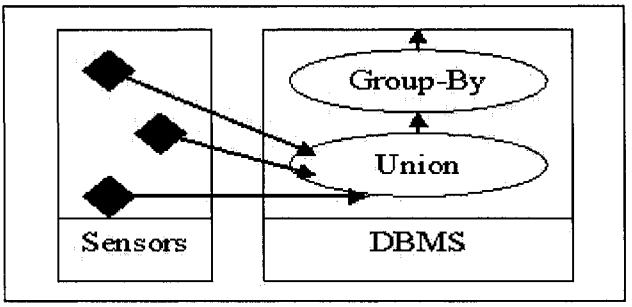


Figure 3.1. Possible query tree for the environment sensor query.

punctuations are not immediately output as they arrive. Instead, union stores punctuations in its state until all inputs have produced equal punctuations. At that point, a punctuation is output denoting the end of data items for that hour.

When a data item arrives at group-by, the appropriate group is updated, in this case, the maximum temperature for a specific hour. When a punctuation denoting the end of an hour arrives, group-by can output results for that hour, clear out its state for that hour, and emit a new punctuation denoting the end of data items for that hour. Thus, the query is unblocked, and the amount of state required has been reduced, making it more appropriate for unbounded data streams.

3.4 Sources of Punctuations

We have seen how punctuated streams help query operators. However, we have not explained how punctuations get into a data stream. We posit a logical operator that embeds punctuations and can occur in various places: at the stream source, at the edge of the query processor, or after query operators within the query. We call this operator the *insert punctuation* operator. There are many different schemes for implementing the insert punctuation operator. Which scheme to choose depends on where the information resides for generating punctuation. We list some alternatives below:

- **Source or sensor intelligence**: The stream source may know enough to emit a punctuation. For example, the individual environmental sensors produced data sorted on time. When an hour ended, the sensor emitted punctuation that all reports for that hour had been output.

- **Knowledge of access order**: Scan or fetch operations may know something about the source, and generate punctuations based on that knowledge. For example, if scan is able to use an index to read a source, it may

use information from that index to tell when all values for an attribute have been read.

- **Knowledge of stream or application semantics**: An insert punctuation operator may know something about the semantics of its source. In the environmental example, temperature sensors might have temperature limits, say -20F and 125F. An insert punctuation operator can output two punctuations immediately: One that says there will not be any temperature reports below -20F and another that says there will not be any reports above 125F.

- **Auxiliary information**: Punctuation may be generated from sources other than the input stream, such as relational tables or other files. In the environmental example, we might have a list of all the sensor units. An insert punctuation operator could use that to determine when all sensors output results for a particular hour, and embed the punctuation itself. This approach can remove punctuation logic from the sensors.

- **Operator semantics**: Some query operators impose semantics on output data items. For example, the sort operator can embed punctuations based on its sort order. When it emits a data item, it can follow that data item with a punctuation stating that no more data items will appear that precede that item in order.

3.5 Open Issues

We have seen that punctuations can improve the behavior of individual query operators for processing unbounded data streams. One issue not addressed yet is how to determine if punctuations can improve the behavior of entire queries. There are two questions here: First, what kinds of queries can be helped by punctuations? Not all queries can be improved by punctuations; we would like to be able to characterize those that can. The second question is, given a query (that we believe can be improved by punctuations), what kinds of punctuations will help that query? We refer to the set of punctuations that will be emitted from a stream source as the *punctuation scheme* of that source. In the sensor query, a punctuations scheme that describes the hour attribute helps the query, but so do schemes that punctuate every 20 minutes, or at the end of every second hour.

A related question is, of the kinds of punctuation schemes that will improve the behavior of a query, which are most efficient? Again referring to the environmental query, if punctuations are emitted at the end of each hour, memory usage is minimized since state is purged as soon as possible. However, this choice maximizes the number of punctuations in the stream. If instead punctuations are embedded every six hours, then memory usage is increased but the

number of punctuations in the stream is reduced and the processing time for them is reduced.

One final issue relates to query optimization. Given a logical query, do two (or more) equivalent query plans exist that exhibit different behaviors based on the same input punctuation scheme? For example, if one query plan is unblocked by the scheme and another is not, then choosing the unblocked query plan is most logical. Optimizing for state size is more difficult, since punctuation schemes do not give guarantees on when a particular punctuation will arrive. However, it would be useful for a query optimizer to choose the query plan with the smallest predicted requirement for memory.

3.6 Summary

Punctuations are useful for improving the behavior of queries over unbounded data streams, even when the input arrives out-of-order. Query operators act on punctuations based on three kinds of behaviors: Pass behavior defines when a blocking operator can output results. Keep behavior defines what state must be kept by a stateful operator. Propagation behavior defines when an operator can emit punctuation.

4. Windows: Handling Unbounded Behavior by Modifying Queries

Windowing operates on the level of either a whole query or an individual operator, by changing the semantics from computing one answer over an entire (potentially unbounded) input streams to repeated computations on finite subsets (windows) of one or more streams. Two examples:

1 Consider computing the maximum over a stream of temperature readings. Clearly, this query cannot emit output while data items are still arriving. A windowed version of this query might, for example, compute the maximum over successive 3-minute intervals, emitting an output for each 3-minute window.

2 Consider a query that matches packet information from two different network routers. Retaining all items from both sources in order to perform a join between them will quickly exhaust the storage of most computing systems. A windowed version of this query might restrict the matching to packets that have arrived in the last 15 seconds. Thus, any packet over 15 seconds old can be discarded, once it has been compared to the appropriate packets from the other input.

There are several benefits from modifying a query with windows.

- An operation, such as aggregation, that would normally be blocking can emit output even while input continues to arrive.

- A query can reduce the state it must retain to process the input streams.
- Windowing can also reduce computational demands, by limiting the amount of data an operation such as join must examine at each iteration.

There have been many different ways of defining windows proposed. The size of a window can be defined in terms of the number of items or by an interval based on an attribute in the items, such as a timestamp. The relationship between successive window instances can vary. In a *tumbling* window [Carney et al., 2002], successive window instances are disjoint, while in a *sliding* window the instances overlap. Window instances may have the same or different sizes. For example, in a *landmark* window [Gehrke et al., 2001], successive instances share the same beginning point (the landmark), but have successively later endpoints.

5. Dealing with Disorder

Stream query approaches such as windowing often require that data arrive in some order. For example, consider the example from Section 3, where we want the maximum temperature value from a group of sensors each hour. This query can be modified to a window query that reports the maximum temperature data items in each hour interval is output, as follows (using syntax similar to CQL [Arasu et al., 2003]):

```
SELECT MAX(temp)
FROM sensor [RANGE 60 MINUTES];
```

In a simple implementation, when a data item arrives that belongs to a new window, the results for the current window is "closed" its maximum is output, and state for a new window is initialized. However, such an implementation assumes that data arrive in sorted order. Suppose the data items do not quite arrive in order. How can we accurately determine if a window is closed?

5.1 Sources of Disorder

A data stream is in *disorder* when it has some expected arrival order, but its actual arrival order does not follow the expected arrival order exactly. It may be nearly ordered, but with a few exceptions. For example, the following list of integers is in disorder: [1,2,3,5,4,6,7,9,10,8]. Clearly the list is close to being in order, and can be put back in order with buffering.

Disorder can arise in a data stream for several reasons: Data items may take different routes, with different delays, from their source; the stream might be a combination of many sources with different delays; the ordering attribute of interest (e.g., event start time) may differ from the order in which items are produced (e.g., event end time). Further, an operator in a stream processing

system may not maintain sort order in its output, even if the data items arrive in order. For a simple example, consider the union operator. Unless it is implemented to maintain sorted order, its output will not necessarily be ordered.

5.2 Handling Disorder

A query operator requiring ordered data can be modified to handle data streams in disorder. First, it must know the *degree of disorder* in the stream: how far away from sorted order each data item in the stream can be. There are two approaches we discuss: *global disorder properties* and *local disorder properties*. Once the operator can determine the degree of disorder, it has a least two choices on how to proceed. It can put its input into sorted order, or it can process the input out of order.

5.2.1 Expressing the Degree of Disorder in a Data Stream. The degree of disorder can be expressed using global or local stream constraints. A global disorder property is one that holds for the entire stream. Several systems use this approach. In Gigascope [Johnson et al., 2003], the degree of disorder can be expressed in terms of the position of a data item in the stream, or in terms of the value of the sorting attribute in a data item. A stream is *increasing within* δ if, for a data item t in stream S, no data item arrived δ items before t on S that precede t in the sort order. Thus, disorder is expressed in terms of a data item's position in the stream. Similarly, a stream is *banded-increasing* (ε) for an attribute A if, for a data item t in stream S, no data item precedes t in S with a value for A greater than $t.A + \varepsilon$.

Related to these notions from Gigascope are *slack* in Aurora [Carney et al., 2002] and *k-constraints* in STREAM [Babu et al., 2004]. In Aurora, an operator that requires sorted input is given an ordering specification, which contains the attribute on which the order is defined and a slack parameter. The slack parameter specifies how out of order a data item might arrive, in terms of position. In STREAM, a k-constraint specifies how strictly an input adheres to some constraint. One kind of k-constraint is *k-ordering*, where k specifies that out-of-order items are at most k positions away from being in order. Note that $k = 0$ implies sorted input.

There are two advantages to using a global disorder property approach. First, it is relatively simple to understand in that it is generally expressed with a single integer. Second, it generally gives a bound on the amount of state required during execution and the amount of latency to expect in the output. However, global disorder properties also have disadvantages. First, it is not always clear what the constraint should be for non-leaf query operators in a query plan. For example, suppose a query has a windowed aggregate operator above the union of five inputs. We may know the degree of disorder of each input to the union, but what is the degree of disorder for the output of union? A second

disadvantage is that it is generally not flexible. A bursty stream will likely have a higher degree of disorder during bursts and a lower degree during lulls. If we want accurate results, we must set global disorder constraint to the worst-case scenario, increasing the latency at other times.

A second way to express the degree of disorder is through local disorder properties [Tucker and Maier, 2003]. In this method, we are able to determine through properties of the stream the degree of disorder during execution. One method to determining local disorder is to use punctuations. Appropriate punctuation on an ordering attribute can be used, for example, to close a window for a windowed operator. Punctuations are propagated to other operators higher up in the query plan. Thus, there is not the problem of how disorder in lower query operators translates to disorder in operators further along in a query tree. In STREAM, the k value for a k-constraint can dynamically change based on data input, similar to a local disorder property. A monitoring process checks the input data items as they arrive, and tries to detect when the k value for useful constraints changes during execution.

The main advantage of using a local disorder property approach is its flexibility. The local disorder property approach can adapt to changes in the stream, such as bursts and lulls. However, since the degree of disorder may not remain static throughout execution, we cannot determine a bound for the state requirement as we can with global disorder properties.

5.2.2 Processing Disordered Data Streams. Once an operator knows the degree of disorder in its input, it can begin processing data from the input stream. One approach in handling disorder is to reorder the data as they arrive in the leaf operators of the query, and use order-preserving operators throughout the query. In Aurora, disordered data streams are ordered using the *BSort* operator. BSort performs a buffer-based sort given an ordering specification. Suppose n is the slack in the ordering specification. Then the BSort operator sets up a buffer of size $n+1$, and as data items arrive they are inserted into the buffer. When the buffer fills, the minimum data item in the buffer according to the sort order is evicted. Note that if data items arrive outside the slack parameter value, they are still placed in the buffer and output as usual. Thus, the BSort operator is only an approximate sort, and its output may still be in disorder.

As data are sorted (at least approximately), later operators should preserve order. Some operators, such as select and project, already maintain the input order. It is a more difficult task for other operators. Consider an order-preserving version of union, and suppose it is reading from two inputs already in order. Union outputs the minimum data item, according to the sort order, from the two inputs. This implementation is simple for reliable inputs, but data streams are not always reliable. Suppose one of the inputs to union stalls. The union oper-

ator cannot output data items that arrive on the other input until the stalled input resumes. Maintaining order in other operators, such as join, is also non-trivial.

Instead of forcing operators to maintain order, an alternative is for data to remain disordered, and process each data item as it arrives. Many operators (again select and project are good examples) do not require data to arrive in order. However, operators that require some sort of ordered input must still determine the degree if disorder in the input. If we use one of the global disorder property approaches, then we must estimate the degree of disorder of the output based on the global disorder properties of the input. However, if we use punctuations, then disorder information is carried through the stream automatically using each operator's propagation behaviors.

5.3 Summary

Many operators, such as window operators, are sensitive to window order. However, as streams are not always reliable data sources, disorder may arise. To handle disorder, an operator must first determine the degree of disorder in its inputs. Once the degree of disorder is determined, then the operator can either resort the data process the data out-of-order. We have presented different ways to express disorder in a stream, and the advantages and disadvantages of sorting data compared to processing data out-of-order.

6. Synopses: Processing with Bounded Memory

Two key parameters for processing user queries over continuous, potentially unbounded data-streams are (1) the amount of *memory* made available to the on-line algorithm, and (2) the *per-item processing time* required by the query processor. Memory, in particular, constitutes an important design constraint since, in a typical streaming environment, only limited memory resources are available to the data-stream processing algorithms. In such scenarios, we need algorithms that can summarize the underlying streams in concise, but reasonably accurate, *synopses* that can be stored in the allotted amount of memory and can be used to provide *approximate answers* to user queries along with some reasonable guarantees on the quality of the approximation. Such approximate, on-line query answers are particularly well suited to the exploratory nature of most data- stream processing applications such as, e.g., trend analysis and fraud or anomaly detection in telecom-network data, where the goal is to identify generic, interesting or "out-of-the-ordinary" patterns rather than provide results that are exact to the last decimal.

In this section, we briefly discuss two broad classes of data-stream synopses and their applications. The first class of synopses, termed *AMS sketches*, was originally introduced in an influential paper by [Alon et al., 1996] and relies on taking random linear projections of a streaming frequency vector. The second

class of synopses, termed FM sketches, was pioneered by [Flajolet and Martin, 1995] and employs hashing to randomize incoming stream values over a small (i.e., logarithmic-size) array of hash buckets. Both AMS and FM sketches are small-footprint, randomized data structures that can be easily maintained on-line over rapid- rate data streams; furthermore, they offer tunable, probabilistic accuracy guarantees for estimating several useful classes of aggregate user queries. In a nutshell, AMS sketches can effectively handle important aggregate queries that rely on *bag semantics* for the underlying streams (such as frequency- moment or join-size estimation), whereas FM sketches are useful for aggregate stream queries with *set semantics* (such as estimating the number of distinct values in a stream). Before describing the two classes of sketches in more detail, we first discuss the key elements of a stream-processing architecture based on data synopses.

6.1 Data-Stream Processing Model

Our generic data-stream processing architecture is depicted in Figure 3.2. In contrast to conventional DBMS query processors, our query-processing engine is allowed to see the data tuples in relations $R_1, ..., R_r$ *only once* and in the fixed order of their arrival as they stream in from their respective source(s). Backtracking over a stream and explicit access to past tuples is impossible; furthermore, the order of tuples arrival for each streaming relation R_i is arbitrary and duplicate tuples can occur anywhere over the duration of the stream. (In general, the stream rendering each relation can comprise tuple *deletions* as well as insertions, and the sketching techniques described here can readily handle such *update streams*.)

Consider an aggregate query Q over relations $R_1, ..., R_r$ and let N denote an upper bound on the total number of streaming tuples. Our data-stream processing engine is allowed a certain amount of memory, typically significantly smaller than the total size of its inputs. This memory is used to continuously maintain a concise *sketch synopsis* of each stream R_i (Figure 3.2). The key constraints imposed on such synopses are that: (1) they are much smaller than the size of the underlying streams (e.g., their size is logarithmic or poly-logarithmic in N); and, (2) they can be easily maintained, during a single pass over the streaming tuples in the (arbitrary) order of their arrival. At any point in time, the approximate query-processing engine can combine the maintained collection of synopses to produce an approximate answer to query Q.

6.2 Sketching Streams by Random Linear Projections: AMS Sketches

Consider a simple stream-processing scenario where the goal is to estimate the size of a binary equi-join of two streams R_1 and R_2 on join attribute A,

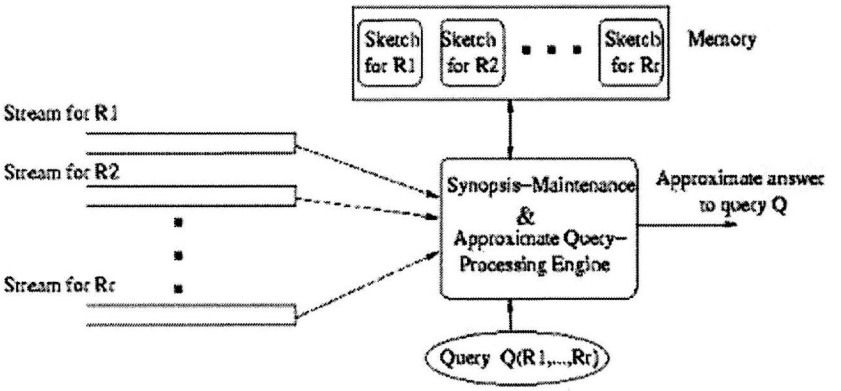


Figure 3.2. Synopsis-based stream query processing architecture.

as the tuples of R_1 and R_2 are streaming in. Without loss of generality, let $[M] = \{0, ..., M - 1\}$ denote the domain of the join attribute A, and let $f_k(i)$ be the frequency of attribute value i in R_k . Thus, we want to produce an estimate for the expression $Q = \sum_i f_1(i)f_2(i)$. Clearly, estimating this join size exactly requires space that is at least linear in M, making such an exact solution impractical for a data-stream setting.

In their influential work, [Alon et al., 1996], [Alon et al., 1999] propose a randomized join-size estimator for streams that can offer strong probabilistic accuracy guarantees while using space that can be significantly sublinear in M. The basic idea is to define a random variable X that can be easily computed over the streaming values of $R_1.A$ and $R_2.A$ such that: (1) X is an *unbiased* (i.e., correct on expectation) estimator for the target join size, so that $E[X] = Q$; and, (2) X's variance can be appropriately upper-bounded to allow for probabilistic guarantees on the quality of the Q estimate. This random variable X is constructed on-line from the two data streams as follows:

- Select a family of *four-wise independent binary random variables* $\{\xi_i : i = 0, ..., M - 1\}$, where each ξ_i assumes a value of either +1 or −1, each with probability 1/2. Informally, the four-wise independence condition means that for any 4-tuple of ξ_i variables and for any 4-tuple of $\{+1, -1\}$ values, the probability that the values of the variables coincide with those in the $\{+1, -1\}$ 4-tuple is exactly 1/16 (the product of the equality probabilities for each individual ξ_i). The crucial point here is that, by employing known tools for the explicit construction of small sample spaces supporting four-wise independence, such families can be efficiently constructed on-line using only $O(\log M)$ space.

- Define $X = X_1.X_2$, where $X_k = \sum_i f_k(i).\xi_i$, for k=1,2. The scalar quantities X_1 and X_2 are called the atomic AMS sketches of streams R_1 and R_2, respectively. Each X_k is simply a random linear projection (i.e., an inner product) of the frequency vector of attribute $R_k.A$ with the random vector of ξ_i's that can be efficiently generated from the streaming values of $R_k.A$: Initialize a counter with $X_k = 0$ and simply add ξ_i to X_k whenever value i is observed in the R_k stream.

Using the four-wise independence property for the ξ's, it is easy to verify that the atomic estimate X constructed using the process above is an unbiased estimate for Q and its variance can be appropriately upper bounded (Alon et al., 1996, 1999). Furthermore, note that, by virtue of linearity, handling deletions in the stream(s) becomes straightforward: To delete an occurrence of value i, simply subtract ξ_i from the running counter.

As an example, suppose the $R_1.A$ and $R_2.A$ streams comprise, in order, the data values [1, 1, 2, 3, 1, 3] and [3, 1, 3, 1, 1], respectively. Projecting on the family of random variables ξ_i, the atomic sketches of the two streams are $X_1 = \xi_1 + \xi_1 + \xi_2 + \xi_3 + \xi_1 + \xi_3 = 3\xi_1 + \xi_2 + 2\xi_3$ and $X_2 = 3\xi_1 + 2\xi_3$, respectively. Using a specific family of binary random variates, say $\xi = \{\xi_1 = -1, \xi_2 = +1, \xi_3 = -1\}$, we get the atomic AMS sketches $X_1 = -3 + 1 - 2 = -4$ and X_2 = -3-2 = -5, and the atomic estimate X = (-4)(-5) = 20, which approximates the true size of the binary join, i.e., 13.

The approximation guarantees of the randomized AMS join-size estimate can be improved using standard boosting techniques that maintain several independent instantiations of the above-described process, and use averaging and median-selection operators over the atomic X estimates to boost accuracy and probabilistic confidence (Alon et al. 1996, 1999). Thus, the AMS sketch for each stream (Figure 3.2) essentially comprises several independent atomic AMS sketch instances (constructed by simply selecting independent random seeds for generating the families of four-wise independent ξ 's for each instance).

Extensions of the Basic Method and Applications. The basic ideas of AMS (more generally, random-linear-projection) sketches have found applications in a number of important data-stream processing problems. [Dobra et al., 2002] and [Dobra et al., 2004] extend the techniques and results of Alon et al. to handle the estimation of complex, multi-join aggregate queries over streams; they also develop algorithms for effectively processing multiple such queries concurrently over a collection of streams by intelligently sharing sketching space and processing. [Feigenbaum et al., 1999] and [Indyk, 2000] use random linear projections to accurately estimate L_p norms over vectors rendered as streams of item arrivals. AMS sketches are also employed by [Charikar et al., 2002] to efficiently process top-k queries over a stream of items, and [Gilbert et al., 2001], [Gilbert et al., 2002] to build approximate histograms and wavelet

decompositions over streams. Recent work has also demonstrated the utility of AMS sketching in dealing with more complex stream-processing scenarios, such as approximating queries with spatial predicates (e.g., overlap joins) over streams of multi-dimensional spatial data [Das et al., 2004], or estimating tree-edit-distance similarity joins over streaming XML documents [Garofalakis and Kumar, 2003].

6.3 Sketching Streams by Hashing: FM Sketches

Consider the problem of estimating the number of distinct values in a stream of arriving attribute values *R.A*, where the domain of the attribute is again assumed, without loss of generality, to be $[M] = \{0, ..., M\text{-}1\}$. (Here, *R* can denote the union of any subset of the R_i streams in Figure 3.2.) As a simple example, for the stream [1, 3, 1, 3, 5, 3, 7] the exact number of distinct values is 3; note that, unlike joins, this query has set semantics (i.e., the multiplicity of values appearing in the stream is unimportant). Once again, this estimation problem can be solved exactly in space that is linear in *M*, which could be impractical in a data-stream setting.

To build a small-space estimate for the number of distinct values in a stream, [Flajolet and Martin, 1995] employ a combination of: (1) a hash function *h()* that maps incoming data values uniformly and independently over the collection of binary strings in the input data domain [*M*]; and, (2) the *lsb()* operator that returns the position of the least-significant 1-bit in its input binary string. The basic idea in their scheme is to map each incoming data value *i* to *lsb*(*h*(*i*)). Obviously, $lsb(h(i)) \in \{0, ..., logM - 1\}$ and, furthermore, it is easy to verify that *lsb(h(i))*= *k* with probability $2^{-(k+1)}$ for each $k = 0, ..., \log M\text{-}1$.

An atomic FM sketch maintained by the basic Flajolet-Martin scheme is simply a bit-vector of size $O(\log M)$. This bit-vector is initialized to all zeros and, for each incoming stream value *i*, the bit located at position *lsb(h(i))* is switched on. The key observation here is that, by virtue of the exponentially-decaying probabilities for the *lsb(h())* values, we expect a fraction of $2^{-(k+1)}$ of the distinct values in the stream to map to location *k* in the bit-vector; in other words, if *D* denotes the number of distinct values in the stream, we expect *D/2* values to map to bit 0, *D/4* values to map to bit 1, and so on. Thus, intuitively, at any point in the stream, the location *l* of the leftmost zero in the FM bit-vector sketch provides a good basic estimate of log *D*, or $2^l \approx D$.

Again, the accuracy and probabilistic confidence of FM-sketching estimates can be boosted using several independent instantiations of the process above (i.e., several atomic FM sketches with independently-chosen hash functions). Detailed analyses and formal results for FM-sketching techniques can be found in [Alon et al., 1996], [Flajolet and Martin, 1995]; [Ganguly et al., 2003]. FM sketches can also handle deletions in the stream: The basic idea is to

maintain a counter (instead of a bit) for each location of the synopsis vector, and simply increment (decrement) the counter at location *lsb(h(i))* for each insertion (respectively, deletion) of value *i*.

Extensions of the Basic Method and Applications. Recent work has extended the ideas of FM (i.e., hashing-based) sketches and explored their use in different data-stream processing domains. [Gibbons, 2001] employs the idea of hashing into buckets with exponentially decaying probabilities to obtain a *distinct sample* summary for estimating SQL aggregates with a `DISTINCT` clause. [Ganguly et al., 2003] extend the basic FM sketch synopsis structure and propose novel estimation algorithms for estimating *general set-expression* cardinalities over streams of updates. Finally, [Considine et al., 2004] propose FM-sketching techniques for approximate, communication-efficient aggregation over wireless sensor networks.

6.4 Summary

AMS and FM sketches represent two important classes of randomized synopsis data structures for streaming data with several applications in stream-processing problems. Besides having a small memory footprint and being easily computable in the streaming model, these sketch synopses can also easily handle deletions in the streams. An additional benefit of both AMS and FM sketches is that they are *composable*; that is, they can be individually computed over a distributed collection of sites (each observing only a portion of the stream) and then combined (e.g., through simple addition or bit-wise OR) to obtain a sketch summary of the overall stream. Several other types of (deterministic and randomized) stream synopses have been proposed for different streaming problems. Vitter's reservoir-sampling scheme for constructing a uniform random sample over an insert-only stream [Vitter, 1985] is probably one of the first known stream-summarization techniques. [Greenwald and Khanna, 2001] and [Manku and Motwani, 2002] propose deterministic, small-footprint stream synopses for computing approximate quantiles and frequent itemsets, respectively. [Datar et al., 2002] consider the problem of maintaining approximate counts over a sliding window of an input stream; their proposed (deterministic) *exponential histogram* synopses employ histogram buckets of exponentially-growing sizes and require space that is only poly-logarithmic in the size of the sliding window. Other stream-synopsis structures for sliding-window computation have been recently proposed by [Gibbons and Tirthapura, 2002], and [Arasu and Manku, 2004].

7. Discussion

We wish to raise two points in closing. The first is that there are areas of overlap among the various techniques described in this chapter. For example,

a windowed aggregate query is not that different from a group-by query on the window attribute with appropriate punctuation. Both serve to unblock a normally blocking operation, and both limit the amount of state the operations in a query must maintain. The second is that these techniques can sometimes be used in combination. For example, the Data Triage architecture of the TelegraphCQ system switches to computing a synopsis of an incoming data stream when it must drop tuples because it cannot keep up with the current data rate [Reiss and Hellerstein, 2004].

Acknowledgments

We would like to thank Leonidas Fegaras, Jin Li, Vassilis Papadimos, Tim Sheard and Kristin Tufte for discussions on punctuations, window queries and disorder, as well as Rajeev Rastogi for numerous discussions on stream synopses. The first two authors were supported in part by DARPA through NAVY/SPAWAR contract N66001-99-108908 and by NSF ITR award IIS 0086002.

References

Alon, N., Gibbons, P., Matias, Y., and Szegedy, M. (1999). Tracking join and self-join sizes in limited storage. In *Proceedings of ACM PODS Conference*, pages 10–20.

Alon, N., Matias, Y., and Szegedy, M. (1996). The space complexity of approximating the frequency moments. In *Proceeding of ACM STOC Conference*, pages 20–29.

Arasu, A., Babu, S., and Widom, J. (2003). The CQL continuous query language: semantic foundations and query execution. *Stanford University TR No. 2003-67 (unpublished)*.

Arasu, A. and Manku, G. S. (2004). Approximate counts and quantiles over sliding windows. In *Proceedings of ACM PODS Conference*, pages 286–296.

Babu, S., Srivastava, U., and Widom, J. (2004). Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *TODS*, 29(3):545–580.

Carney, D., Cetintemel, Ugur, Cherniack, Mitch, Convey, Christian, Lee, Sangdon, Seidman, Greg, Stonebraker, Michael, Tatbul, Nesime, and Zdonik, Stanley B. (2002). Monitoring Streams - A New Class of Data Management Applications. In *VLDB Conference*, pages 215–226.

Charikar, M., Chen, K., and Farach-Colton, M. (2002). Finding frequent items in data streams. In *Proceedings of ICALP Conference*, pages 3–15.

Cisco Systems. (2001). *Netflow Services Solutions Guide*.

Considine, J., Li, F., Kollios, G., and Byers, J. (2004). Approximate aggregation techniques for sensor databases. In *Proceedings of IEEE ICDE Conference*, pages 449–460.

Das, A., Gehrke, J., and Riedewald, M. (2003). Approximate join processing over data streams. In *Proceedings of ACM SIGMOD Conference*, pages 40–51.

Das, A., Riedewald, M., and Gehrke, J. (2004). Approximation techniques for spatial data. In *Proceedings of ACM SIGMOD Conference*, pages 695–706.

Datar, M., Gionis, A., Indyk, P., and Motwani, R. (2002). Maintaining Stream Statistics over Sliding Windows. In *Proceedings of SODA Conference*, pages 635–644.

Dobra, Alin, Garofalakis, Minos, Gehrke, Johannes, and Rastogi, Rajeev (2002). Processing Complex Aggregate Queries over Data Streams. In *Proceedings of ACM SIGMOD Conference*, pages 61–72.

Dobra, Alin, Garofalakis, Minos, Gehrke, Johannes, and Rastogi, Rajeev (2004). Sketch-Based Multi-Query Processing over Data Streams. In *Proceedings of EDBT Conference*, pages 551–568.

Feigenbaum, J., Kannan, S., Strauss, M., and Viswanathan, M. (1999). An approximate L^1- difference algorithm for massive data streams. In *Proc. IEEE FOCS Conference*, page 501.

Flajolet, P. and Martin, N. (1995). Probabilistic counting algorithms for data base applications. *JCSS Journal*, 31(2):182–209.

Ganguly, S., Garofalakis, M., and Rastogi, R. (2003). Processing set expressions over continuous update streams. In *Proceedings of ACM SIGMOD Conference*, pages 265–276.

Garofalakis, M. and Kumar, A. (2003). Correlating XML data streams using tree-edit distance embeddings. In *Proceedings of ACM PODS Conference*, pages 143–154.

Gehrke, J., Korn, F., and Srivastava, D. (2001). On computing correlated aggregates over continual data streams. In *Proceedings of ACM SIGMOD Conference*, pages 13–24.

Gibbons, P. (2001). Distinct sampling for highly-accurate answers to distinct values queries and event reports. In *Proceedings of VLDB Conference*, pages 541–550.

Gibbons, P. and Tirthapura, S. (2002). Distributed streams algorithms for sliding windows. In *Proceedings of ACM SPAA Conference*, pages 63–72.

Gilbert, A. C., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2001). Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proceedings of VLDB Conference*, pages 79–88.

Gilbert, A. C., Guha, S., Indyk, P., Kotidis, Y., Muthukrishnan, S., and Strauss, M. (2002). Fast, small-space algorithms for approximate histogram maintenance. In *Proceedings of ACM STOC Conference*, pages 389–398.

Greenwald, M. B. and Khanna, S. (2001). Space-efficient online computation of quantile summaries. In *Proceedings of ACM SIGMOD Conference*, pages 58–66.

Hillston, J. and Kloul, L. (2001). Performance investigation of an on-line auction system. *Concurrency and Computation: Practice and Experience*, 13:23–41.

Indyk, P. (2000). Stable Distributions, Pseudorandom generators, embeddings, and data stream computation. In *Proceedings of IEEE FOCS Conference*, page 189.

Johnson, T., Cranor, C., Spatscheck, O., and Shkapenyuk, V. (2003). Gigascope: A stream database for network applications. In *Proceedings of ACM SIGMOD Conference*, pages 647–651.

Kang, J., Naughton, J. F., and Viglas, S. D. (2003). Evaluating window joins over unbounded streams. In *Proceedings of the International Conference on Data Engineering (ICDE)*.

Manku, G. S. and Motwani, R. (2002). Approximate frequency counts over data streams. In *Proceedings of VLDB Conference*, pages 346–357.

Rajasekar, A., Vernon, F., Hansen, T., Linquist, K., and Orcutt, J. (2004). Virtual object ring buffer: A framework for real-time data grid. In *Proceedings of HDPC Conference*.

Reiss, F. and Hellerstein, J. M. (2004). Data triage: An adaptive architecture for load shedding in TelegraphCQ. *Intel Research Berkeley Report IRB-TR-04-004*.

Tatbul, N., Cetintemel, U., Zdonik, S., Cherniack, M., and Stonebraker, M. (2003). Load shedding in a data stream manager. In *Proceedings of VLDB Conference*, pages 309–320.

Tucker, P. A. and Maier, D. (2003). Dealing with disorder. In *MPDS Workshop*.

Tucker, P. A., Maier, D., Fegaras, L., and Sheard, T. (2003). Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3):555–568.

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

Wilschut, Annita N. and Apers, Peter M. G. (1991). Dataflow query execution in a parallel main-memory environment. In *Proceedings of PDIS Conference*, pages 68–77.

Chapter 4

XML & DATA STREAMS

Nicolas Bruno,[1] Luis Gravano,[2] Nick Koudas,[3] and Divesh Srivastava[3]

[1] *Microsoft Research*
nicolasb@microsoft.com

[2] *Columbia University*
gravano@cs.columbia.edu

[3] *AT&T Labs–Research*
koudas,divesh@research.att.com

Abstract XQuery path queries form the basis of complex matching and processing of XML data. Most current XML query processing techniques can be divided in two groups. *Navigation-based* algorithms compute results by analyzing an input stream of documents one tag at a time. In contrast, *index-based* algorithms take advantage of (precomputed or computed-on-demand) numbering schemes over each input XML document in the stream. In this chapter, we present an index-based technique, *Index-Filter*, to answer multiple path queries. *Index-Filter* uses indexes built over the document tags to avoid processing large portions of an input document that are guaranteed not to be part of any match. We analyze *Index-Filter*, compare it against *Y-Filter*, a state-of-the-art navigation-based technique, and present the advantages of each technique.

Keywords: Data streams, XML, XPath

1. Introduction

The eXtensible Markup Language (XML) is a standard for data exchange, which in recent years, has attracted significant interest from industrial and research forums. Applications increasingly rely on XML to not only exchange but also query, filter and transform data. Thanks to standard specifications for web services (such as SOAP, WSDL, etc.) applications can receive requests for data (specified in XML) and return their answers tagged in XML. In addition, via the use of specific query languages such as XPath and XQuery, users and applications can compose declarative specifications of their interests as well as filter and transform specific data items represented in XML.

The reasons behind XML's popularity are not surprising. The ability to exchange data between entities (applications, systems, individuals, etc.) has always been of vast importance. Before XML became a reality, the need for data exchange was met with proprietary techniques defined and supported by specific vendors. The emergence of XML as a standard and the development of a core set of specifications around XML enabled the steady transition from proprietary data representations and languages to XML-enabled suites of applications. XML is a relatively simple markup language. Every XML document consists of a set of element nodes, which can be nested and assembled in a hierarchical (tree-structured) fashion. Element nodes have start and end tags for data associated with each element. XML, as its name suggests, is extensible, in the sense that one can easily include new element names (tailored to application requirements or semantics). Queries in XML query languages, such as XQuery [Boag et al., 2004b], typically specify patterns of selection predicates on multiple elements that have some specified tree structured (e.g., parent-child, ancestor-descendant) relationships, as the basis for matching XML documents. For example, the path query `//book[.//title = 'XML']` matches `book` elements that have a descendant `title` element with a value 'XML'. Finding all matches to such path queries in an XML document is a core operation in various XML query processing scenarios that have been considered in the literature. Since XML representation can be very flexible, Document Type Definitions (DTDs) and XML schemas are used to ensure that XML documents conform to specific constraints imposed on the structural relationships the elements can assume. Various efforts are underway to standardize DTDs and schemas for specific application domains and kinds of data.

There are two broad classes of problems in which XML has been at the center of attention, from both an industrial and a research point of view.

1.1 XML Databases

The first class of problems concerns the efficient management of stored data represented as XML. In this case, traditional data management issues are ex-

plored, such as storage of XML in relational and non-relational stores, specification of query languages for XML, as well as efficient query processing strategies over large collections of XML data. This XML query processing scenario typically involves asking a single query against (possibly preprocessed and indexed) XML documents. The goal here is to identify the matches to the input query in each XML document. Approaches to solving this problem typically take advantage of the availability of indexes on XML elements and values, and use specialized join algorithms for composing the results of the index lookups to compute answers to path queries. These are referred to as *index-based* algorithms.

This area has been very active in terms of research and development. All major DBMS vendors support storage and querying of XML documents in their engines. Moreover, native XML data managers (i.e., deploying XML specific storage managers) are now available. We briefly review related work in these areas later in this chapter.

1.2 Streaming XML

The second class of problems emphasizes the online aspects of XML data dissemination and querying. A common characteristic of applications in this class is that XML data collections are not static any more. Instead, XML documents arrive continuously forming (potentially unbounded) *streams* of documents. This latter class of problems is the focus of this chapter.

There are two main application domains in which XML streams are currently commonly encountered, namely web services and data dissemination in publish/subscribe systems.

Web services provide a novel way to link systems and applications together and expose logic and functionality over networks, eliminating the complexity and expense of more traditional approaches to connectivity. Such services are enabled by standards-based technologies, such as SOAP, WSDL and UDDI, dispensing the need for knowledge of system specifics, such as languages, operating systems, software and version numbers, etc. A prevalent type of web services is message-based or conversational services in which loosely coupled systems interact by exchanging entire documents (e.g., purchase orders) tagged in the lingua franca of web services, namely XML, rather than exchanging discrete sets of parameters. In these types of services, XML tagged documents are constantly exchanged between applications forming continuous document streams. A central aspect of such services is the ability to efficiently operate on XML document streams executing queries in order to extract parts of documents and drive legacy back-end applications. Commonly, multiple queries could require execution against the incoming documents. For example, one query could seek document fragments that relate to a purchase order in order to

populate back-end relational databases. This could mean that the query would not only extract but also transform the information into relational tuples. A different query could seek to extract billing-related fragments of the document to populate other databases or trigger an event. Thus, commonly in the web services application domain, multiple queries have to be executed against incoming XML documents. As a result, continuously arriving streams of XML data pose significant problems regarding simultaneous processing of multiple queries. Application constraints and/or performance reasons make the online evaluation of queries on such document streams imperative.

Publish/subscribe systems (see, e.g., [Pereira et al., 2001]) aim to facilitate data dissemination in various domains. In such systems, users (or applications) register their interest for specific pieces of information as queries. Incoming documents are matched against such queries and matching documents are disseminated to interested users. In an XML publish/subscribe system, queries are expressed in some XML query language (such as XPath) and are subsequently evaluated over each incoming XML document. As in the web services scenario, a number of challenges exist related to the efficient simultaneous evaluation of multiple XML queries against incoming documents. It is evident that in such applications the number of registered queries to be matched against incoming documents can be large. Depending on the specific application domain (e.g., news articles, measurement data, etc.) the size of incoming documents can vary vastly.

Approaches to solving this class of problems (see, e.g., [Altinel and Franklin, 2000; Chan et al., 2002; Ives et al., 2002; Lakshmanan and Parthasarathy, 2002; Peng and Chawathe, 2003; Green et al., 2003; Barton et al., 2003; Gupta and Suciu, 2003]) typically navigate through the input XML document one tag at a time, and use the preprocessed structure of path queries to identify the relevant queries and their matches. These are referred to as *navigation-based* algorithms. In contrast, *index-based* algorithms take advantage of numbering schemes over the input XML document to identify the relevant document fragments and their matching queries.

1.3 Contributions

In general, we consider scenarios where *multiple XML queries need to be matched against a stream of XML documents*, and either (or neither) of the queries and the documents may have been preprocessed. In principle, each of the query processing strategies (index-based and navigation-based) could be applied in our general scenario. How this is best achieved, and identifying the characteristics of our general scenario where one strategy dominates the other, are the subjects of this chapter, the core contents of which appeared in [Bruno et al., 2003]. We consider the following methods:

- A straightforward way of applying the *index-based* approaches in the literature to our general scenario would answer each path query separately, which may not be the most efficient approach, as research on multi-query processing in relational databases has demonstrated.

  We present an index-based technique, *Index-Filter*, to answer multiple XML path queries against an XML document. *Index-Filter* generalizes the *PathStack* algorithm of [Bruno et al., 2002], and takes advantage of a prefix tree representation of the set of XML path queries to share computation during multiple query evaluation.

- *Navigation-based* approaches in the literature could be applied to the general scenario as well. We enhance *Y-Filter* [Diao et al., 2003], a state-of-the-art navigation-based technique, to identify multiple query matches.

The rest of the chapter is structured as follows. In Section 2 we discuss our data, streaming, and query models. Section 3 discusses two query processing techniques. In Section 3.2, we review *Y-Filter*, a state-of-the-art navigation-based algorithm, suitably enhancing it for our application scenarios. Then, in Section 3.3, we present the index-based algorithm, *Index-Filter*. Finally, in Section 4, we review related work and draw some connections with broader research in stream query processing.

2. Models and Problem Statement

In this section, we introduce the XML data, query, and streaming models that we use in the rest of the chapter, and define our problem statement.

2.1 XML Documents

An *XML document* can be seen as a rooted, ordered, labeled tree, where each node corresponds to an element or a value, and the edges represent (direct) element-subelement or element-value relationships. The ordering of sibling nodes (children of the same parent node) implicitly defines a total order on the nodes in a tree, obtained by traversing the tree nodes in preorder. An XML stream consists of multiple XML documents.

EXAMPLE 4.1 *Figure 4.1 shows a fragment of an XML document that specifies information about a book. The figure shows four children of the* `book` *node, namely:* `title`, `allauthors`, `year`, *and* `chapter`, *in that order. Intuitively, we can interpret the XML fragment in the figure as a book published in the year* 2000 *by* `Jane Poe`, `John Doe`, *and* `Jane Doe`, *entitled* `XML`. *Its first chapter, also entitled* `XML`, *starts with the section* `Origins`.

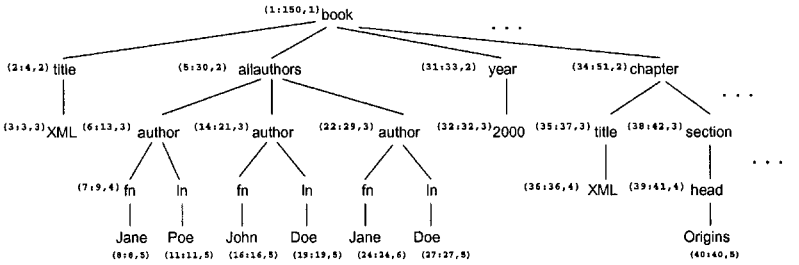


Figure 4.1. A fragment XML document.

book	title	XML
(1:150,1)	(2:4,2)	(3:3,3)
(1:150,1)	(35:37,3)	(36:36,4)

(a) Path query. (b) Query answer.

Figure 4.2. Query model used in this chapter.

The numbers associated with the tree nodes can be thought of as unique node identifiers in the XML tree, which will be used for efficient query processing. This will be explained later in Section 3.3.1.

2.2 Query Language

XQuery [Boag et al., 2004b] path queries can be viewed as sequences of location steps, where each node in the sequence is an element tag or a string value, and query nodes are related by either *parent-child* steps (depicted using a single line) or *ancestor-descendant* steps (depicted using a double line).[1]

Given a query q and an XML document D, a *match of q in D* is a mapping from the nodes in q to nodes in D such that: (i) node tags and values are preserved under the mapping, and (ii) structural (parent-child and ancestor-descendant) relationships between query nodes are satisfied by the mapped document nodes. The *answer* to a query q with n nodes is an n-ary relation where each tuple $(d_1, \ldots, d_n)$ consists of the document nodes that identify a match of q in D.[2]

EXAMPLE 4.2 *Consider again Figure 4.1. The path query* `//book[.//title = 'XML']` *identifies* `book` *elements that have a descendant* `title` *element that*

in turn have a child 'XML' (value) node. This query can be represented as in Figure 4.2(a). A match for this query in the XML fragment of Figure 4.1 is the following mapping, where nodes in the document are represented by their associated numbers: book $\rightarrow (1:150,1)$, title $\rightarrow (2:4,2)$, *and* XML $\rightarrow (3:3,3)$. *It is easy to check that both the name tags and the structural relationships between nodes are preserved by the mapping above. Figure 4.2(b) shows the answer to the query in Figure 4.2(a) over the XML fragment of Figure 4.1.*

2.3 Streaming Model

In streaming XML applications, XML documents arrive continuously forming a potentially unbounded stream. At a fine granularity, this stream can be viewed as a stream of begin and end tags for document nodes, interspersed with data values. At a coarser granularity, this stream can be viewed as a stream of individual XML documents.

The granularity of streaming influences the query processing strategies that can be used on the stream. Navigation-based strategies are more versatile, and can be used both at the fine granularity and at the coarser granularity. Index-based techniques can be used only at the coarser granularity, since they need to pre-process an entire incoming XML document before query processing can begin. In both web services and publish/subscribe applications, the coarser granularity of document streams is appropriate. Hence, in this chapter, we use the model where the stream consists of individual XML documents.

2.4 Problem Statement

Finding all matches of a path query in an XML document is a core operation in various XML query processing scenarios that have been considered in the literature. In this chapter, we consider the general scenario of matching multiple XML path queries against an XML document in the stream, and focus on the following problem:

> **XML Multiple Query Processing**: Given a set of path queries $Q = \{q_1, \ldots, q_n\}$ and an XML document D, return the set $R = \{R_1, \ldots, R_n\}$, where R_i is the answer (all matches) to q_i on D.

In the XML database query processing scenario, Q includes a single query q_1, and the document D is large and indexed. When the XML database query processing scenario is augmented to deal with multi-query processing, Q includes multiple queries, and the document D is large and indexed. Finally, in the XML information dissemination scenario, Q includes many queries, and the document D is not indexed. In the next section, we study algorithms for our problem of processing multiple XML path queries efficiently.

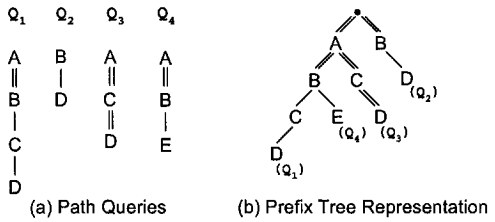


Figure 4.3. Using prefix sharing to represent path queries.

3. XML Multiple Query Processing

We now address the central topic of this chapter: processing strategies for multiple path queries over an XML document. First, in Section 3.1, we describe a mechanism to compress the representation of multiple input path queries that is used by the different algorithms in this chapter. Then, in Section 3.2, we review *Y-Filter* [Diao et al., 2003], a state-of-the-art algorithm that consumes the input document one tag at a time and incrementally identifies all input path queries that can lead to a match. Finally, in Section 3.3, we present *Index-Filter*, an algorithm that exploits indexes built over the document tags and avoids processing tags that will not participate in a match. Section 3.3.3 addresses the problem on how to efficiently materialize the indexes needed by *Index-Filter*.

To draw an analogy with relational query processing, *Y-Filter* can be regarded as performing a sequential scan of the input "relation," while *Index-Filter* accesses the relation via indexes. Extending the above analogy, not surprisingly, neither algorithm is best under all possible scenarios. In Section 3.4, we summarize the scenarios for which each algorithm is best suited.

3.1 Prefix Sharing

When several queries are processed simultaneously, it is likely that significant commonalities between queries exist. To eliminate redundant processing while answering multiple queries, both the navigation- and index-based techniques identify query commonalities and combine multiple queries into a single structure, which we call *prefix tree*. Prefix trees can significantly reduce both the space needed to represent the input queries and the bookkeeping required to answer them, thus reducing the execution times of the different algorithms. Consider the four path queries in Figure 4.3(a). We can obtain a prefix tree that represents such queries by sharing their common prefixes, as shown in Figure 4.3(b). It should be noted that although other sharing strategies can be applied, we do not explore them in this work.

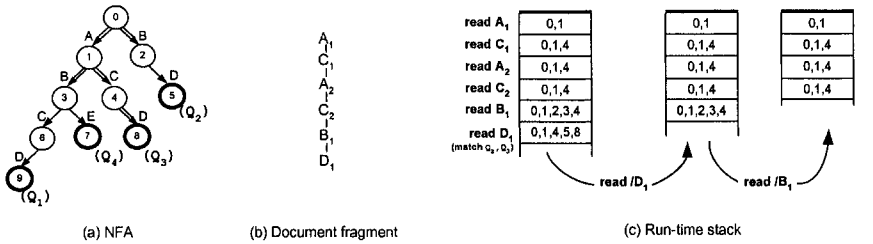


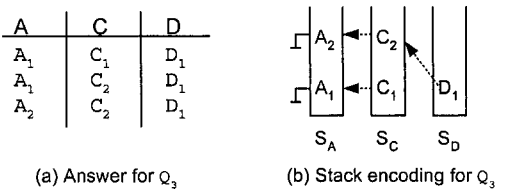
Figure 4.4. Y-Filter algorithm.

3.2 Y-Filter: A Navigation-Based Approach

Y-Filter is a state-of-the-art navigation-based technique for processing multiple path queries. The main idea is to augment the prefix tree representation[3] of the input queries into a *non-deterministic finite automaton* (NFA) that behaves as follows: (i) The NFA identifies the exact "language" defined by the union of all input path queries; (ii) when an output state is reached, the NFA outputs all matches for the queries accepted at the state. Unlike an NFA used to identify a regular language, the filtering of XML documents requires that processing continues until all possible accepting states have been reached.

The incoming XML document is parsed one tag at a time. While parsing, *start-tag* tokens trigger transitions in the NFA (the automaton is non-deterministic, so many states can be active simultaneously). When an *end-tag* token is parsed, the execution backtracks to the state immediately preceding the corresponding start-tag. To achieve this goal, a run-time stack structure is used to maintain the active and previously processed states.

EXAMPLE 4.3 *Consider the NFA shown in Figure 4.4(a), which corresponds to the prefix tree of Figure 4.3(b). Note that each node in the prefix tree is converted to a state in the NFA, and the structural relationships in the prefix tree are converted to transitions in the NFA, triggered by the corresponding tags. As each start-tag from the document fragment in Figure 4.4(b) is parsed, the NFA and the run-time stack are updated. Figure 4.4(c) shows the run-time stack after each step, where each node in the stack contains the set of active states in the NFA. Initially, only the starting state, 0, is active. When reading the start-tag for node A_1, state 0 fires the transition to state 1, and both states 0 and 1 become active (state 0 remains active due to the descendant edge from state 0 to state 1; otherwise we would not be able to capture the match that uses A_2, which is a descendant of A_1). As another example, after reading the start-tag D_1, both states 5 and 8 become active, and therefore a match for queries Q_2 and Q_3 is detected (note that after reading D_1, node 2 is not active anymore, since the firing transition from node 2 to node 5 used a child –not descendant–*



(a) Answer for Q_3 (b) Stack encoding for Q_3

Figure 4.5. Compact solution representation.

structural relationship).[4] *As a final example, after reading the close-tag for* D_1, *the run-time stack is backtracked to the state immediately before reading the start-tag for* D_1.

Implementation-wise, *Y-Filter* augments each query node in the NFA with a hash table. The hash table is indexed by the children node tags, and is used to identify a particular transition of a given state. Therefore, the NFA can be seen as a tree of hash tables. This implementation is a variant of a hash table-based NFA, which has been shown to provide near constant time complexity to perform state transitions (see [Diao et al., 2003] for more details).

Compact Solution Representation. The original formulation of *Y-Filter* [Diao et al., 2003] just returns the set of queries with a non-empty answer in a given document. However, we are interested in returning all matches as the answer for each query (see Section 2). Consider again Figure 4.4 when the algorithm processes D_1. The partial matches for query Q_3 are shown in Figure 4.5(a). When parsing node D_1, the original *Y-Filter* algorithm would get to the final state for Q_3, only guaranteeing that there is *at least one match* for Q_3 in the document. In other words, there is no way to *keep track* of repeating tags that might result in multiple solutions.

To overcome this limitation, in this chapter we augment *each node* q in the prefix tree (NFA) with a stack S_q. These stacks S_q efficiently keep track of all matches in the input document from the root to a given document node. Each element in the stack is a pair: $\langle$ node from the XML document, pointer to a position in the parent stack $\rangle$. At *every* point during the computation of the algorithm, the following properties hold:

1 The nodes in S_q (from bottom to top) are guaranteed to lie on a root-to-leaf path in the XML document.

2 The set of stacks contains a *compact encoding* of partial and total matches to the path query, which can represent in linear space a potentially exponential (in the number of query nodes) number of answers to the path query.

Given a chain of stacks in a leaf-to-root path in the prefix tree, corresponding to some input query, the following recursive property allows us to extract the set of matches that are encoded in the stacks: given a pair $\langle t_q, p_q \rangle$ in stack S_q, the set of partial matches using tag t_q that can be extracted from the stacks is found by extending t_q with either the partial matches that use the tag pointed by p_q in $S_{\texttt{parent}(q)}$ or any tag that is below p_q in stack $S_{\texttt{parent}(q)}$. The following example clarifies this property.

EXAMPLE 4.4 *Consider again Figure 4.4. The set of all matches for the NFA in Figure 4.4(a) is shown in Figure 4.5(a). Figure 4.5(b) shows the chain of stacks for query Q_3 and the stack encoding for the document fragment at the point that D_1 is processed. The match $[A_2, C_2, D_1]$ is encoded since D_1 points to C_2, and C_2 points to A_2. Since A_1 is below A_2 on S_A, $[A_1, C_2, D_1]$ is also encoded. Finally, since C_1 is below C_2 on S_C and C_1 points to A_1, $[A_1, C_1, D_1]$ is also encoded. Note that $[A_2, C_1, D_1]$ is not encoded, since A_2 is* above *the node (A_1) on S_A to which C_1 points.*

It can be shown that in order to maintain the stacks in the NFA we need to proceed as follows: (i) every time an open tag t_o is read and consequently some state n becomes active due to a transition from state n_p, we push into n's stack t_o along with a pointer to the top of n_p's stack; and (ii) every time a close tag t_c is read and the top of the (global) run-time stack structure contains states $\{n_1, \ldots, n_k\}$, we pop the top element from the stacks, associated with states n_i, that were modified when the corresponding open-tag t_o was read. It is important to note that the stacks are shared among queries in the prefix-tree. In fact, we only need one stack per state in the NFA to represent all partial matches of the queries that share such state. We refer to [Bruno et al., 2002] for more details about maintaining compact solutions using stacks.

In conclusion, our modification to the original *Y-Filter* algorithm allows us to return not only the queries that have at least one match in the document, but all its matches. Moreover, this can be done by using a limited amount of memory (proportional to the height of the input XML document and the number of active states). We present now an alternative technique to return the set of all matches for the given input queries that is based on a different design principle: the use of index information over the input XML document.

3.3 Index-Filter: An Index-Based Approach

In this section we present *Index-Filter*, a technique to answer multiple path queries by exploiting indexes that provide structural information about the tags in the XML document. By taking advantage of this additional information, *Index-Filter* is able to avoid processing certain tags in the document that are guaranteed not to be part of any match. We first discuss the index structure that we use in *Index-Filter*, and then we present the main algorithm.

3.3.1 Indexing XML Documents. We now describe how to extend the classic inverted index data structure used in information retrieval [Salton and McGill, 1983] to provide a positional representation of elements and values in the XML document. This representation was introduced in [Consens and Milo, 1994] and has been used in [Zhang et al., 2001; Li and Moon, 2001; Bruno et al., 2002; Chien et al., 2002; Jiang et al., 2003b] for matching XML path queries. As we will see, this representation allows us to efficiently check whether two tags in the XML documents are related by a parent/child or ancestor/descendant structural relationship. The position of an element occurrence in the XML document is represented as the pair `(L:R,D)` where `L` and `R` are generated by counting word numbers from the beginning of the document to the start and the end of the element being indexed, respectively, and `D` is the nesting depth of the element in the document (see Figure 4.1 for examples of pairs associated with some tree nodes based on this representation).

We can easily determine structural relationships between tree nodes using this indexing scheme. Consider document nodes n_1 and n_2, encoded as ($\mathtt{L}_1$: $\mathtt{R}_1, \mathtt{D}_1$) and ($\mathtt{L}_2$: $\mathtt{R}_2, \mathtt{D}_2$), respectively. Then, n_1 is an *ancestor* of n_2 (and n_2 is a *descendant* of n_1) if and only if $\mathtt{L}_1 < \mathtt{L}_2$ and $\mathtt{R}_2 < \mathtt{R}_1$. To check whether n_1 is the *parent* of n_2 (n_2 is a *child* of n_1) we also need to verify whether $\mathtt{D}_1+1=\mathtt{D}_2$.

EXAMPLE 4.5 *Consider the XML fragment in Figure 4.1. The* `author` *node with position* (6:13, 3) *is a descendant of the* `book` *node with position* (1:150, 1), *since* $L_{\mathtt{book}} = 1 < 6 = L_{\mathtt{author}}$, *and* $R_{\mathtt{author}} = 13 < 150 = R_{\mathtt{book}}$. *Also, the* `author` *node just mentioned is the parent of the* `fn` *node with position* (7:9, 4), *since* $L_{\mathtt{author}} = 6 < 7 = L_{\mathtt{fn}}$, $R_{\mathtt{fn}} = 9 < 13 = R_{\mathtt{author}}$, *and* $D_{\mathtt{fn}} = 4 = 3 + 1 = D_{\mathtt{author}} + 1$.

An important property of this positional representation is that checking an ancestor-descendant relationship is computationally as simple as checking a parent-child relationship, i.e., we can check for an ancestor-descendant structural relationship without knowledge of intermediate nodes on the path.

We now introduce the *Index-Filter* algorithm. Later, in Section 3.3.3 we address the issue of how to efficiently materialize a set of indexes for a given XML document.

3.3.2 Algorithm Index-Filter. Based on the representation of positions in the XML document described above, we now present the *Index-Filter* algorithm. Analogously to the case of *Y-Filter*, we augment the input prefix tree structure for *Index-Filter*. Specifically, before executing *Index-Filter*, we associate with each node q in the input prefix tree the following information: (i) an *index stream* T_q, which contains the indexed positions of document nodes that match q sorted by their `L` values, (ii) an empty *stack* S_q as discussed in Section 3.2, and (iii) a priority queue P_q that allows dynamic and efficient access to

```
Algorithm Index-Filter(q)
01 while (true) // find candidate node
02   repeat
03     min = getMin(P_q)
04     if (¬min ∨ (isAccept(q) ∧ nextL(T_min) > nextR(T_q)))
05       q.knowSolution=true; return
06     while (nextR(T_q) < nextL(T_min))
07       advance(T_q) // advance q's stream
08     if (eof(T_min)) q.knowSolution=false; return
09     while (¬empty(S_q) ∧ topR(S_q) < nextL(T_min))
10       pop(S_q) // clean q's stack
11     while (nextL(T_min) < skipToL(q))
12       advance(T_min)
13       min.knowSolution=false
14     knewSolution= min.knowSolution
15     if (¬min.knowSolution) Index-Filter(min)
16   until (knewSolution)
17   // process candidate node
18   if (nextL(T_min) > nextL(T_q))
19     q.knowSolution= true
20     return
21   else
22     push(S_min, (next(T_min), ptr_top(S_q)))
23     if (isAccept(min))
24       outputSolutions(min)
25     advance(T_min); if (isLeaf(min)) pop(S_min)
26     Index-Filter(min)
27 end while

Function skipToL(q)
01 if (empty(S_q)) return nextL(T_q)
02 else return topL(S_q)
```

Figure 4.6. Algorithm Index-Filter.

the child of q having the smallest L value in its stream. To ensure correctness, initially we add the index entry $(-\infty : +\infty, 0)$ to the stack S_{root}.

In the rest of the section, the concepts of a prefix tree and its root node are used interchangeably. We denote the current element in stream T_q as the *head of* T_q, and we access the head's L and R components by the functions `nextL` and `nextR`, respectively (if we consume T_q entirely, `nextL`(T_q)=`nextR`(T_q)=$+\infty$). Similarly, we access the L and R components of the top of S_q by the functions `topL` and `topR`, respectively. We now describe *Index-Filter*, which is shown in pseudocode in Figure 4.6.

We execute *Index-Filter*(q) to get all the answers for the prefix tree rooted at q. The algorithm's invariant ensures that after executing *Index-Filter*(q), we are guaranteed that either (1) T_q's head participates in a new match when all structural relationships are regarded as ancestor/descendant (`outputSolutions` in line 24 will later enforce the appropriate relationships); or otherwise (2) the stream T_q is consumed entirely. Additionally, we can guarantee that for all descendants q' of q in the prefix tree, every index entry in $T_{q'}$ with L component smaller than `nextL`(T_q) was already processed. To avoid redundant computations, we memorize this property by carefully manipulating the boolean variable q.`knowSolution`: if q.`knowSolution`=**true**, we know that T_q's head participates in at least one new match; otherwise all we can say is that T_q's head *might* participate in a new match, but we do not know for sure (initially, q.`knowSolution` is set to **false** for every node q). The algorithm iterates through two phases until all matches are returned. In the first phase (lines 2-16), we identify min, the child of q with the minimal L value in its stream's head that participates in some match. In the second phase (lines 17-26), we process min depending on the actual relationship with T_q's head. We now give some details on each phase.

To identify min, we first use the priority queue P_q to select the child of q with the smallest stream head (line 3). Lines 4-5 cover the special case that node q is a leaf node in the prefix tree (so q has no children and there is no min child), or q is an internal node in the prefix tree but some query has q as its accept state and q's position ends before the position of any of q's children. In such cases, we simply update q.`knowSolution` = **true** and return. Otherwise, in the general case, if T_{min}'s head starts after T_q's head ends, we can guarantee that no new match can exist for T_q's head, so we advance T_q (see Figure 4.7(a)). At this point, if T_{min} is consumed entirely, we know that there are no new solutions for q so we return (line 8). Otherwise, we clean from q's stack all elements that cannot participate in any new match, i.e., those elements in S_q whose R component is smaller than the L component of T_{min}'s head (see Figure 4.7(b)). After that, we compute the value `skipToL`, which is the smallest L value for a node from q for which a new match can exist. If T_{min}'s head starts before `skipToL`, we know that T_{min}'s head cannot participate in any new match, so

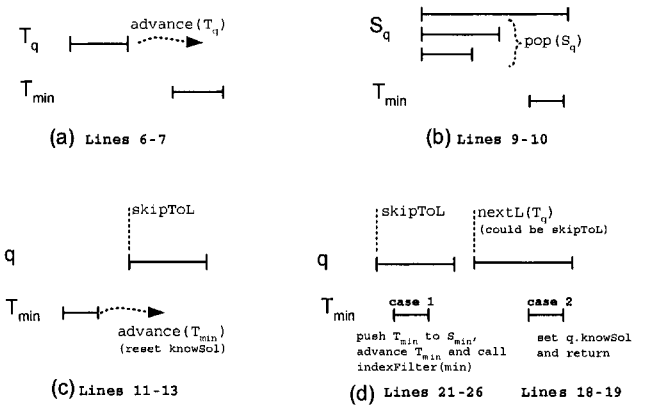


Figure 4.7. Possible scenarios in the execution of Index-Filter.

we advance T_{min} (see Figure 4.7(c)). In such a case, we need to *reset* the value min.`knowSolution` (line 13), since we can no longer guarantee that T_{min}'s head participates in a new match after advancing T_{min} in line 12. At this point, in line 15, if we cannot guarantee that T_{min}'s head participates in a match (i.e., min.`knowSolution` = **false**), we recursively call *Index-Filter*(min). After we return from this recursive call, we can guarantee (from the algorithm's invariant) than T_{min}'s head participates in a new match. However, T_{min} could have been advanced in the recursive call, so we cannot guarantee that min is the children of q with the *minimal* value of L participating in a match. Therefore, we only continue with the second phase if we could guarantee that min participated in a match *before* the recursive call (see lines 14 and 16). Otherwise, we simply repeat the procedure of finding the minimal child of q with a match (of course, in the next iteration node min could be the minimal one, although it is not always the case).

When we enter the second phase, we can guarantee that T_{min}'s head participates in some match and its position relative to q can be just one of the two cases of Figure 4.7(d). In the case that T_{min}'s head starts after T_q's head (case 2 in Figure 4.7(d)), we know that T_q's head participates in a match as well, so we set q.`knowSolution` and return (lines 19-20). Otherwise (case 1 in Figure 4.7(d)), T_{min}'s head ends before T_q's head starts but participates in a match with nodes in S_q. Therefore, we need to process node min before returning with any match for T_q's head. We first push T_{min}'s head to S_{min}, and if some query has min as its accept node, we expand the new matches from the chain of stacks. Finally, in preparation for the next iteration, we advance T_{min} and recursively call *Index-Filter*(min) to process any remaining entries from the subtree rooted at min.

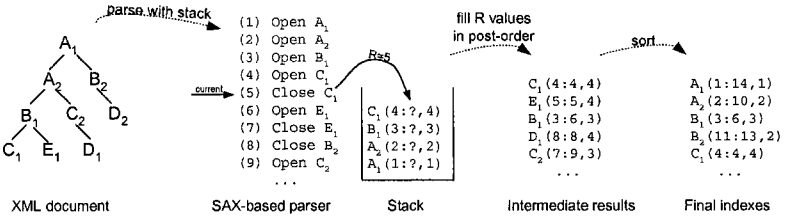


Figure 4.8. Materializing the positional representation of XML nodes.

3.3.3 Building Indexes. When analyzing the *Index-Filter* algorithm in the previous section, we assumed that the required indexes were already precomputed and available to *Index-Filter*. We now give more details on how to efficiently materialize a set of indexes given an input XML document. Conceptually, we can assign the positional representation of the nodes in the XML document by traversing the XML tree in preorder as explained next. We maintain a global counter and increment it every time we move to a new node (either by moving to a new child or when returning to the parent node after traversing all of its subtrees). Whenever we reach a node for the first time, we assign the current value of the counter to the L component of the positional representation, and when we leave a node after traversing all its subtrees we assign the current value of the counter to the R component of the positional representation (the D component of the positional representation is easily derived from the number of ancestors of each node). We now present a concrete implementation of this procedure that uses little main memory and scales gracefully with the size of the input XML document. The general procedure to obtain the indexes for the nodes of the XML document consists of two steps that can be summarized as follows (see Figure 4.8):

1 Use a SAX-based parser on the input XML tree. The i-th tag found (irrespective of whether it is a start- or an end-tag) receives integer i as its identifier.[5] Every time we parse a start-tag, we push into a global stack the value of the tag along with its identifier, which becomes the L value for the node, and the level value, which is simply the current number of elements in the stack. (Figure 4.8 shows a snapshot of the execution right after parsing the start-tag for node C_1.) On the other hand, every time we parse an end-tag, we know (assuming the XML document is well-formed) that the top element in the stack has the information of the corresponding start-tag, so we pop the top of the stack, assign the current identifier to the R component, and output the index entry to a temporary file.

2 The order of the tags in the intermediate file produced in the previous step would match that of a post-order traversal of the XML tree (in particular, the different tags are not even grouped together). As seen in the description of *Index-Filter* (Section 3.3) a crucial property of the index entries for a given tag is that they are sorted by L value. For that reason, in the second step we sort the intermediate file by $\langle tag, L \rangle$. In that way, all tags are grouped together and sorted by their L value, as desired (see Figure 4.8).

To provide efficient access to the indexes of individual tags, we build a B-tree over the tags. Throughout the index-building process, and with the exception of the sorting phase, memory requirements are proportional to the height of the XML tree to maintain the stack. Interestingly, the memory requirements are independent of the size of the XML tree.

Main Memory Optimization. It turns out that if the whole document and the indexes fit in main memory, we can build the indexes without the sorting step. The intuition is to use growing arrays in memory to hold the index for each tag separately. Every time we parse a start-tag, we append a new index entry to the corresponding tag array with the L and D entries as before (the R entry remains unknown). We still use a global stack, but this time we just store in it pointers to index entries in the arrays, which still contain unknown R values. When we parse an end-tag, we pop the top pointer from the stack and update the corresponding index entry in the array with the R value as explained before. This way, each index is created independently and in the right order, so there is no need to sort any intermediate result.

3.4 Summary of Experimental Results

Both *Index-Filter* and the enhanced *Y-Filter* have their advantages. In particular, in [Bruno et al., 2003], we experimentally show that:

- When the number of queries is small, or the XML document is large, *Index-Filter* is much more efficient than *Y-Filter* if the required indexes are already materialized, due to the focused processing achieved by the use of indexes.

- When we also consider the time spent for building indexes on the XML document on the fly, the trends remain the same, but the gap between the algorithms is reduced.

- For a very large number of queries, and small documents, *Y-Filter* is more efficient due to the scalability properties of *Y-Filter*'s hash tables.

4. Related Work

XPath [Boag et al., 2004a] and XQuery [Boag et al., 2004b] are the main XML query languages. Their efficient implementation and evaluation has been the subject of considerable recent research, both for XML databases and streaming XML applications.

4.1 XML Databases

XML data and issues in their storage, query evaluation, query optimization, etc., has attracted a lot of attention in the context of semistructured and XML databases. In particular, substantial early work on query processing on such data was done for the Lore system [McHugh et al., 1997] and the Niagara system [Naughton et al., 2001]. Various issues in their storage and query processing using relational DBMSs have been considered in, among others, [Florescu and Kossmann, 1999; Fernandez et al., 2000; Fiebig and Moerkotte, 2000; Shanmugasundaram et al., 2000; Yoshikawa et al., 2001; DeHaan et al., 2003]. In these papers, the authors considered different ways of mapping XML data to a number of relational tables, along with a translation of XML queries to SQL queries.

Recognizing the inadequacy of traditional relational join algorithms for efficiently processing XML queries, [Zhang et al:, 2001; Li and Moon, 2001; Al-Khalifa et al., 2002; Bruno et al., 2002; Chien et al., 2002; Jiang et al., 2003b], among others, introduced various novel join algorithms as primitives for matching structural (edge, path, twig) queries against an XML document. In particular, [Zhang et al., 2001; Li and Moon, 2001; Al-Khalifa et al., 2002; Chien et al., 2002] proposed binary structural join algorithms as primitives for matching twig queries. The algorithms in [Bruno et al., 2002; Jiang et al., 2003b] are generalizations of the binary structural join algorithms to holistically match path and twig queries. The main contribution of these holistic algorithms is that no large intermediate results are generated for complex path or twig queries, eliminating the need for an optimization step that was needed when stitching together partial results from the binary structural join algorithms. Our *Index-Filter* algorithm of Section 3.3 is loosely based on the *PathStack* technique of [Bruno et al., 2002].

These join algorithms typically rely on numbering schemes that represent the positions of XML elements, as discussed in Section 3.3.1. Such numbering schemes were used in [Consens and Milo, 1994], who considered a fragment of the PAT text searching operators for indexing text databases, and computing containment relationships between "text regions" in the text databases.

Complementing the work on structural join algorithms has been the development of a variety of index structures to find matches to individual XPath axes (see, e.g., [Grust 2002; Jiang et al., 2003a]), to paths in an XML document (see,

e.g., [Cooper et al., 2001; Chung et al., 2002]), and to twigs (see, e.g., [Wang et al., 2003; Rao and Moon, 2004]).

4.2 Streaming XML

[Altinel and Franklin, 2000; Chan et al., 2002; Diao et al., 2003; Ives et al., 2002; Lakshmanan and Parthasarathy, 2002; Peng and Chawathe, 2003; Green et al., 2003; Barton et al., 2003; Gupta and Suciu, 2003] proposed various navigation-based techniques to match single and multiple, path and twig queries. These works assume the fine granularity model of streaming, as discussed in Section 2.3. In particular, [Ives et al., 2002] introduced the X-Scan operator, which matches path expression patterns over a streaming (non-materialized) XML document and [Peng and Chawathe, 2003] explores single query evaluation against streaming XML documents using transducers. References [Altinel and Franklin, 2000; Diao et al., 2003] consider the problem of answering multiple path queries over incoming documents. The algorithms and data structures in both [Altinel and Franklin, 2000] and [Diao et al., 2003] are tailored for the case of very large numbers of queries and small input documents. While [Altinel and Franklin, 2000] uses separate finite state machines to represent each query, [Diao et al., 2003] compresses the set of input queries by sharing prefixes, as explained in Section 3.1. Reference [Chan et al., 2002] proposes a trie-based data structure, called XTrie, to support filtering of complex twig queries. The XTrie, along with a sophisticated matching algorithm, is able to reduce the number of redundant matchings.

We note that the query model in [Altinel and Franklin, 2000; Diao et al., 2003; Chan et al., 2002] is slightly different from ours. They are mainly concerned with queries for which at least one match exists (therefore several optimizations are available to avoid processing queries beyond their first match). In contrast, in *Index-Filter*, we are interested in returning the set of all matches for each input query. Finally, [Lakshmanan and Parthasarathy, 2002] addresses the problem of obtaining all matches for a set of path and tree queries. The algorithms use an index structure, denoted the "requirements index", which helps to quickly determine the set of queries for which a certain structural relationship (e.g., parent-child, ancestor-descendant) is relevant. The main difference with our query model is that in [Lakshmanan and Parthasarathy, 2002] each input query identifies a unique "distinguished" query node, so the result matches are 1-ary relations. The algorithms in [Lakshmanan and Parthasarathy, 2002] make at most two passes on the input document, and provide performance guarantees on the number of I/O invocations required to find the resulting matches.

4.3 Relational Stream Query Processing

There is a vast body of recent work in the area of relational stream query processing (see, e.g., [Babcock et al., 2002; Koudas and Srivastava, 2003], and the references therein). While much of the research in relational and XML stream query processing has proceeded independently, the tutorial slides of [Koudas and Srivastava, 2003] explored preliminary connections between these fields. In particular, under the fine granularity model of XML streams, (1) numbers can be associated with start and end tags, and with values, such that the stream of XML tokens can be viewed as multiple homogeneous relational streams (for start tags, text values, and end tags); (2) path and twig matching on the XML streams can be modeled as multi-way joins (with arithmetic conditions on the numbers mentioned above) over the relational streams; and (3) the XML end tags have a natural correspondence with explicit punctuations used in relational streams (see, e.g., [Tucker et al., 2003]), to identify "end of processing".

There have been many join algorithms proposed for relational stream query processing (see, e.g., [Kang et al., 2003; Viglas et al., 2003; Golab and Ozsu, 2003]). While these algorithms could, in principle, be used for matching XML paths based on the above correspondences, they are not as efficient as the specialized streaming algorithms proposed for matching XML paths and twigs. How this gap can be bridged is an interesting direction of future work.

5. Conclusions

We surveyed algorithms to answer multiple path queries over XML documents efficiently. In particular, we reviewed *Y-Filter*, a state-of-the-art *navigation-based* algorithm. *Y-Filter* computes results by analyzing an input document stream one tag at a time, typically by using SAX-based parsers. We extended *Y-Filter*'s original formulation so that it returns all matches for the set of input queries. We also presented an *index-based* algorithm, *Index-Filter*, which avoids processing portions of the XML document that are guaranteed not to be part of any match. *Index-Filter* takes advantage of precomputed indexes over the input document, but can also build the indexes on the fly. Both techniques have their advantages. In particular, while most XML stream query processing techniques work off SAX events, in some cases it pays off to parse the input document in advance and augment it with auxiliary information that can be used to evaluate the queries faster.

Notes

1. XQuery path queries permit other axes, such as *following* and *preceding*, which we do not consider in this chapter.

2. Actually, an XQuery path query is a 1-ary projection of this n-ary relation. Compositions of path queries need to be used (as in the XQuery FOR clause) to obtain an n-ary relation. To allow for this generality, while keeping the exposition simple, we identify the path query answer with the n-ary relation.

3. Actually, *Y-Filter* uses a slightly different representation of the prefix tree, but we omit details to simplify the presentation.

4. The actual implementation of *Y-Filter*'s NFA is slightly more complex than described above, to address a special situation. In particular, when a given state has two children with the same tag but different structural relationships (child and descendant), a new intermediate state is added to the NFA to differentiate between the two transitions.

5. To keep the presentation simple, we treat values, such as 'Jane' or 'XML', as if they were composed of adjacent pairs of open- and close-tags, e.g., <Jane></Jane>, but we assign the same integer to both the open- and close-tags.

References

Mehmet Altinel and Michael J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of VLDB*, 2000.

Shurug Al-Khalifa, H. V. Jagadish, Nick Koudas, Jignesh M. Patel, Divesh Srivastava, and Yuqing Wu. Structural joins: A primitive for efficient XML query pattern matching. In *Proceedings of ICDE*, 2002.

Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proceedings of ACM PODS*, 2002.

Scott Boag, Don Chamberlin, Mary F. Fernandez, Michael Kay, Jonathan Robie, and Jerome Simeon. XML path language (XPath) 2.0. *W3C Working Draft. Available from*
http://www.w3.org/TR/xpath20, July 2004.

Scott Boag, Don Chamberlin, Mary F. Fernandez, Daniela Florescu, Jonathan Robie, and Jerome Simeon. XQuery 1.0: An XML query language. *W3C Working Draft. Available from*
http://www.w3.org/TR/xquery, July 2004.

Charles Barton, Philippe Charles, Deepak Goyal, Mukund Raghavachari, Marcus Fontoura, and Vanja Josifovski. Streaming XPath processing with forward and backward axes. In *Proceedings of ICDE*, 2003.

Nicolas Bruno, Nick Koudas, and Divesh Srivastava. Holistic twig joins: Optimal XML pattern matching. In *Proceedings of ACM SIGMOD*, 2002.

Nicolas Bruno, Luis Gravano, Nick Koudas, and Divesh Srivastava. Navigation-vs. index-based XML multi-query processing. In *Proceedings of ICDE*, 2003.

Chee Yong Chan, Pascal Felber, Minos Garofalakis, and Rajeev Rastogi. Efficient filtering of XML documents with XPath expressions. In *Proceedings of ICDE*, 2002.

Mariano P. Consens and Tova Milo. Optimizing queries on files. In *Proceedings of ACM SIGMOD*, 1994.

Chin-Wan Chung, Jun-Ki Min, and Kyuseok Shim. APEX: An adaptive path index for XML data. In *Proceedings of ACM SIGMOD*, 2002.

Brian Cooper, Neal Sample, Michael J. Franklin, Gisli R. Hjaltason, and Moshe Shadmon. A fast index for semistructured data. In *Proceedings of VLDB*, 2001.

Shu-Yao Chien, Zografoula Vagena, Donghui Zhang, Vassilis J. Tsotras, and Carlo Zaniolo. Efficient structural joins on indexed XML documents. In *Proceedings of VLDB*, 2002.

David DeHaan, David Toman, Mariano P. Consens, and M. Tamer Ozsu. A comprehensive XQuery to SQL translation using dynamic interval encoding. In *Proceedings of ACM SIGMOD*, 2003.

Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter M. Fischer. Path sharing and predicate evaluation for high-performance XML filtering. *ACM TODS*, 28(4), 2003.

Daniela Florescu and Donald Kossmann. Storing and querying XML data using an RDBMS. *IEEE Data Engineering Bulletin*, 22(3), 1999.

Thorsten Fiebig and Guido Moerkotte. Evaluating queries on structure with extended access support relations. In *Proceedings of WebDB*, 2000.

Mary F. Fernandez, Wang Chiew Tan, and Dan Suciu. SilkRoute: Trading between relations and XML. *Computer Networks*, 33(1-6), 2000.

Torsten Grust. Accelerating XPath location steps. In *Proceedings of ACM SIGMOD*, 2002.

Todd J. Green, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML streams with deterministic automata. In *Proceedings of ICDT*, 2003.

Lukasz Golab and M. Tamer Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of VLDB*, 2003.

Ashish Kumar Gupta and Dan Suciu. Stream processing of XPath queries with predicates. In *Proceedings of ACM SIGMOD*, 2003.

Zachary G. Ives, Alon Y. Halevy, and Daniel S. Weld. An XML query engine for network-bound data. *VLDB Journal*, 11(4), 2002.

Haifeng Jiang, Hongjun Lu, Wei Wang, Beng Chin Ooi. XR-Tree: Indexing XML data for efficient structural joins. In *Proceedings of ICDE*, 2003.

Haifeng Jiang, Wei Wang, Hongjun Lu, and Jeffrey Xu Yu. Holistic twig joins on indexed XML documents. In *Proceedings of VLDB*, 2003.

Jaewoo Kang, Jeffrey F. Naughton, and Stratis Viglas. Evaluating window joins over unbounded streams. In *Proceedings of ICDE*, 2003.

Nick Koudas and Divesh Srivastava. Data stream query processing: a tutorial. In *Proceedings of VLDB*, 2003.

Laks V. S. Lakshmanan and Sailaja Parthasarathy. On efficient matching of streaming XML documents and queries. In *Proceedings of EDBT*, 2002.

Quanzhong Li and Bongki Moon. Indexing and querying XML data for regular path expressions. In *Proceedings of VLDB*, 2001.

Jason McHugh, Serge Abiteboul, Roy Goldman, Dallan Quass, and Jennifer Widom. Lore: A database management system for semistructured data. *SIGMOD Record*, 26(3), 1997.

Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The Niagara Internet query system. *IEEE Data Engineering Bulletin*, 24(2), 2001.

Feng Peng and Sudarshan S. Chawathe. XPath queries on streaming data. In *Proceedings of ACM SIGMOD*, 2003.

Joao Pereira, Francoise Fabret, Hans-Arno Jacobsen, Francois Llirbat, and Dennis Shasha. WebFilter: A high throughput XML-based publish and subscribe system. *Proceedings of VLDB*, 2001.

Praveen Rao and Bongki Moon. PRIX: Indexing and querying XML using Prufer sequences. In *Proceedings of ICDE*, 2004.

Gerard Salton and Michael J. McGill. *Introduction to modern information retrieval.* McGraw-Hill, 1983.

Jayavel Shanmugasundaram, Eugene J. Shekita, Rimon Barr, Michael J. Carey, Bruce G. Lindsay, Hamid Pirahesh, and Berthold Reinwald. Efficiently publishing relational data as XML documents. In *Proceedings of VLDB*, 2000.

Peter A. Tucker, David Maier, Tim Sheard, and Leonidas Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE TKDE*, 15(3), 2003.

Stratis Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of VLDB*, 2003.

Haixun Wang, Sanghyun Park, Wei Fan, and Philip S. Yu. ViST: a dynamic index method for querying XML data by tree structures. In *Proceedings of ACM SIGMOD*, 2003.

Masatoshi Yoshikawa, Toshiyuki Amagasa, Takeyuki Shimura, and Shunsuke Uemura. XRel: a path-based approach to storage and retrieval of XML documents using relational databases. *ACM TOIT*, 1(1), 2001.

Chun Zhang, Jeffrey F. Naughton, David J. DeWitt, Qiong Luo, and Guy M. Lohman. On supporting containment queries in relational database management systems. In *Proceedings of SIGMOD*, 2001.

Chapter 5

CAPE: A CONSTRAINT-AWARE ADAPTIVE STREAM PROCESSING ENGINE

Elke A. Rundensteiner, Luping Ding, Yali Zhu, Timothy Sutherland and Bradford Pielech
Computer Science Department
Worcester Polytechnic Institute (WPI)
{rundenst, lisading, yaliz, tims, winners}@cs.wpi.edu

1. Introduction

1.1 Challenges in Streaming Data Processing

The growth of electronic commerce and the widespread use of sensor networks has created the demand for online processing and monitoring applications [Madden and Franklin, 2002; Stream Repository; Tucker et al., 2003]. In these applications, data is no longer statically stored. Instead, it becomes available in the form of continuous streams. Furthermore, users often ask long-running queries and expect the results to be delivered incrementally in real time. Traditional query execution techniques, which assume finite persistent datasets and aim for producing a one-time query result, become largely inapplicable in this new stream paradigm due to the following reasons:

- The data streams are potentially infinite. Thus the existence of blocking operators in the query plan, such as group-by, may block query execution indefinitely because they need to see all input data before producing a result. Moreover, stateful operators such as join may require infinite storage resources to maintain all historical data for producing exact results.

- Data streams are continuously generated at query execution time. Meta knowledge about streaming data, such as data arrival patterns or data statistics, is largely unavailable at the initial query optimization phase. Therefore the initial processing decisions taken before query execution commences, including the query plan structure, operator execution algorithm and operator scheduling strategy, may not be optimal.

- Stream environments are usually highly dynamic. For example, the data arrival rates may fluctuate dramatically. Moreover, as other queries are registered into or removed from the system, the computing resources available for processing may vary greatly. Hence an optimal query plan may become sub-optimal as it proceeds, requiring run-time query plan restructuring and in some cases even across-machine plan redistribution.

It is apparent that novel strategies must be found to tackle the evaluation of continuous queries in such highly dynamic stream environments. In particular, this raises the need to offer adaptive services at all levels of query processing. The challenge is to cope with the variations in both stream environment and system resources, while still guaranteeing the precision and the timeliness of the query result. This is exactly the challenge that the stream processing system introduced in this chapter, named CAPE (for **C**onstraint-Aware **A**daptive Stream **P**rocessing **E**ngine) [Rundensteiner et al., 2004], tackles.

1.2 State-of-the-Art Stream Processing Systems

Many existing stream processing systems have begun to investigate various aspects of adaptive query execution. STREAM [Motwani et al., 2003] for instance applies runtime modification of memory allocation and supports memory-minimizing operator scheduling policies such as Chain [Babcock et al., 2003]. Aurora [Abadi et al., 2003] supports flexible scheduling of operators via its Train scheduling technique [Carney et al., 2003]. It also employs the load shedding when an overload is detected. In [Cherniack et al., 2003], they point towards ideas for developing a distributed version of the Aurora and Medusa systems, including fault tolerance, distribution and load balancing. TelegraphCQ [Chandrasekaran et al., 2003] provides a very fine-grained adaptivity by routing each tuple individually through its network of operators. While offering maximal flexibility, this comes with the overhead of having to manage the query path taken on an individual tuple basis and of having to recompute intermediate results.

These systems also consider the constraint-exploiting query optimization, in particular, they all incorporate various forms of sliding window semantics to bound the state of stateful operators. In addition, the STREAM system also exploits static k-constraints to reduce the resource requirements [Babu and Widom, 2004]. However, none of these systems considers punctuations which can be used to model both static and dynamic constraints in the stream context. Further optimization opportunities enabled by the interactions between different types of constraints are also not found in these systems.

1.3 CAPE: Adaptivity and Constraint Exploitation

In this chapter, we will describe CAPE, a **C**onstraint-Aware **A**daptive Stream **P**rocessing **E**ngine [Rundensteiner et al., 2004], that we have developed to effectively evaluate continuous queries in highly dynamic stream environments. CAPE adopts a novel architecture that offers highly adaptive services at all levels of query processing, including reactive operator execution, adaptive operator scheduling, runtime query plan restructuring and across-machine plan redistribution. In addition, unlike other systems that under resource limitation duress load shedding and thus affect the accuracy of the query result [Abadi et al., 2003], CAPE instead focuses on maximally serving precise results by incorporating optimizations enabled by a variety of constraints. For instance, the CAPE operators are designed to exploit dynamic constraints such as punctuations [Ding et al., 2004; Tucker et al., 2003] in combination with time-based constraints such as sliding windows [Carney et al., 2002; Hammad et al., 2003; Kang et al., 2003; Motwani et al., 2003] to shrink the runtime state and to produce early partial results.

This chapter now describes four core services in CAPE that are constraint-exploiting and highly adaptive in nature:

- The *constraint-exploiting reactive query operators* exploit constraints to reduce resource requirements and to improve the response time. These operators employ an adaptive execution logic to react to the varying stream environment.
- The *introspective execution scheduling framework* adaptively selects one algorithm from a pool of scheduling algorithms that helps the query to best meet the optimization objectives.
- The *online query plan reoptimization and migration* restructures the query plan at runtime to continuously converge to the best possible route that input data goes through.
- The *adaptive query plan distribution framework* balances the query processing workload among a cluster of machines so to maximally exploit available CPU and memory resources.

We introduce the CAPE system in Section 2. The design of the CAPE query operators is described in Section 3. Sections 4, 5 and 6 present our solutions for operator scheduling, online plan reoptimization and migration, and plan distribution respectively. Finally, we conclude this chapter in Section 7.

2. CAPE System Overview

CAPE embeds novel adaptation techniques for tuning different levels of query evaluation, ranging from intra-operator execution, operator scheduling,

query plan structuring, to plan distribution. Each level of adaptation is able to yield maximally optimized performance in certain situations by working on their own. However, none of them is able to handle all kinds of variations that may occur in a stream environment. In addition, the improper use of all levels of adaptation may cause either optimization counteraction or oscillating re-optimizations, which should both be avoided. Hence an important task is to coordinate different levels of adaptations, guiding them to function properly on their own and also to cooperate with each other in a well-regulated manner. CAPE not only incorporates novel adaptation strategies for all aspects of continuous query evaluation, but more importantly, it employs a well-designed mechanism for coordinating different levels of adaptation.

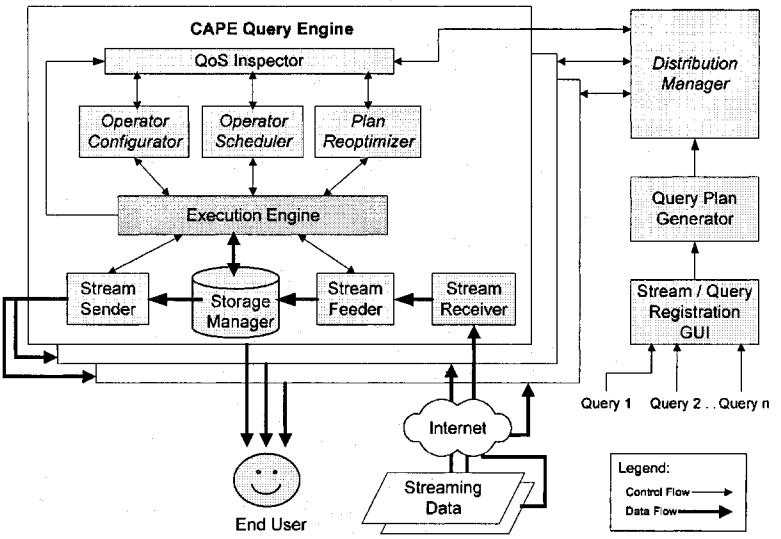


Figure 5.1. CAPE System Architecture.

In the system architecture depicted in Figure 5.1, the key adaptive components are Operator Configurator, Operator Scheduler, Plan Reoptimizer and Distribution Manager. Once the Execution Engine starts executing the query plan, the QoS (Quality of Service) Inspector will regularly collect statistics from the Execution Engine at each sampling point. All the above four adaptive components then use these statistics along with QoS specifications to determine if they need to adjust their behavior.

To synchronize adaptations at all levels, we have designed a ***heterogeneous-grained adaptation schema***. Since these adaptations deal with dissimilar runtime situations and have different overheads, they are invoked in CAPE under different frequencies and conditions. The current adaptation components in

CAPE and the granularities of adaption are shown in Figure 5.2, with the adaption interval increasing as we go from the inner to the outer layers of the onion shape.

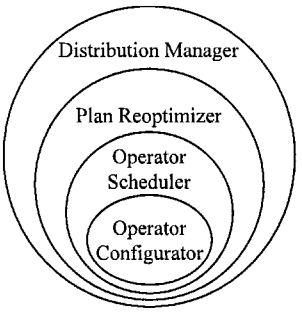


Figure 5.2. Heterogeneous-grained Adaptation Schema.

The intra-operator adaptation incurs the lowest overhead so that it functions within an operator's execution time slot. Our event-driven intra-operator scheduling mechanism enables the operators, especially the stateful and blocking ones, to adjust their execution at runtime through the Operator Configurator. The Operator Scheduler is able to adjust the operator processing order after a run of a single operator or a group of operators, called a scheduling unit. After a scheduling unit finishes its work, the scheduler will check the QoS metrics for the operators and decide which operator to run next or even switch to a better scheduling strategy. This is a novel feature unique to our system. The Plan Reoptimizer will wait for the completion of several scheduling units and then check the QoS metrics for the entire query plan residing on its local machine to decide whether to restructure the plan. The Distribution Manager, which potentially incorporates the highest costs in comparison with other adaptive components due to across-machine data transfers, is invoked the least frequently, i.e., it is assigned the longest decision making interval. If a particular machine is detected to be overloaded, the Distribution Manager will redistribute one or multiple query plans among the given cluster of machines. While the Plan Reorganizer migrate the old plan to a new plan structure, the Distribution Manager instead migrates a query plan from one machine to another machine.

3. Constraint-Exploiting Reactive Query Operators

As described in Section 1.1, uncertainties may exist in many aspects of a streaming environment, including the data arrival rate, the resource availability, etc. The operators in CAPE are designed to react to such variations by adapting their behavior appropriately [Ding et al., 2004]. Moreover, these operators

exploit various constraints to optimize their execution without sacrificing the precision of the query result [Ding et al., 2003].

In this section we use the design of a join operator as an example to illustrate the optimization principles inherent in the CAPE operator design. We highlight in particular two features unique to CAPE: the adaptive operator execution logic and the exploitation of punctuations.

For clarity of presentation, we use a join over two streams S_1<A, B_1> and S_2<A, B_2> with the join condition S_1.A = S_2.A. The schema of the join result is <A, B_1, B_2>. We assume that each tuple or punctuation has a timestamp field *TS* that records its arrival time. We also assume that tuples and punctuations in both streams have a global ordering on their timestamp.

3.1 Issues with Stream Join Algorithm

Pipelined join operators have been proposed for delivering early partial results in processing streaming data [Haas and Hellerstein, 1999; Mokbel et al., 2004; Urhan and Franklin, 2000; Wilschut and Apers, 1993]. These operators build one state per input stream to hold the already-processed data.. As a tuple comes in on one input stream, it is used to *probe* the state of the other input stream. If a match is found, a result is produced. Finally the tuple is *inserted* into the state of its own input stream. In summary, this join algorithm completes the process of each tuple by following the *probe-insert* sequence.

Some issues may arise with this algorithm. As tuples continuously accumulate in the join states, the join may run out of memory. To prevent data loss, part of the state needs to be flushed to disk. This may cause many expensive I/O operations when we try to join new tuples with those tuples on disk. As more data is paged to disk, the join execution will be slowed down significantly. In addition, the join state may potentially consumes infinite storage.

3.2 Constraint-Exploiting Join Algorithm

In most cases it is not necessary to maintain all the historical data in the states. Constraints such as sliding windows [Carney et al., 2002; Hammad et al., 2003; Kang et al., 2003; Motwani et al., 2003] or punctuations [Ding et al., 2004; Tucker et al., 2003] can be utilized by the join to detect and discard no-longer-needed data from the states. This way the join state can be shrunk in a timely manner, thereby reducing and even eliminating the need of paging data to disk.

We first consider the sliding window, a time-range constraint. Assume that in the join predicate, two time-based windows W_1 and W_2 are specified on streams S_1 and S_2 respectively. A new tuple from S_1 can only be joined with tuples from S_2 that arrived within the last W_2 time units. So can new tuples from S_2. Hence the join only needs to keep tuples that have not yet expired from the

window. Any new tuple from one stream can be used to remove expired tuples from the other stream. Accordingly, the probe-insert execution logic should be extended to add a third operation that *invalidates* tuples based on the sliding window constraints.

Punctuations are value-based constraints embedded inside a data stream. A punctuation in a stream is expressed as an ordered set of patterns, with each pattern corresponding to one attribute of the tuple from this stream. The punctuation semantics define that no tuple that arrives *after* a punctuation will match this punctuation. As the join operator receives a punctuation from one stream, it can then purge all already-processed tuples from the other stream that match this punctuation because these tuples no longer contribute to any future join results. In response, a new operation, namely *purge*, that discards tuples according to punctuations, needs to be added into the join execution logic.

Below we use an example query in an online auction application to briefly illustrate how these constraints are used to optimize the evaluation of this query. As shown in Figure 5.3, in this auction system, the items that are open for auction, the bids placed on the items and the registered users are recorded in the *Item*, *Bid* and *Person* streams respectively. When the open duration for a certain item expires, the auction system can insert a punctuation into the Bid stream to signal the end of bids for that item, e.g., the punctuation <1082, ∗, ∗, ∗> on item 1082 in the figure. Figure 5.3 also shows a stream query in CQL language [Arasu et al., 2003] and a corresponding query plan. For each person registered with the auction system, this query asks for the count of distinct categories of all the items this person has bid on within 12 hours of his registration time.

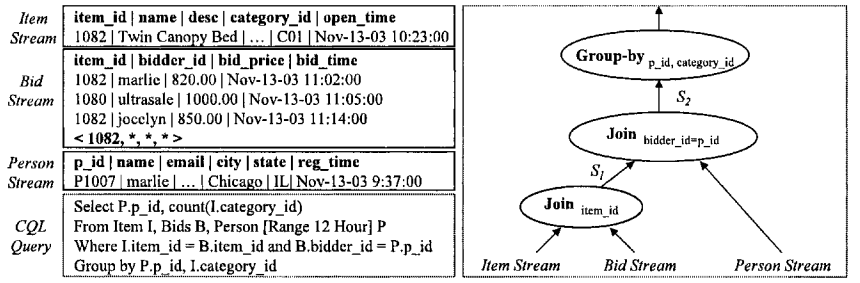


Figure 5.3. Example Query in Online Auction System.

In the first join (Item ⋈ Bid) in the query plan, when a punctuation is received from the Bid stream, the tuple with the matching *item_id* from the Item stream can be purged because it will no longer join with any future Bid tuples. The second join (S_1 ⋈ Person) applies a 12-hour window on Person. Tuples from stream S_1 can be used to invalidate expired tuples from the Person stream.

Besides utilizing punctuations to optimize their own execution, the operators can also propagate punctuations to help other operators. In the above example, when a Person tuple moves out of the window, no more join results will be produced for this person. A punctuation can then be propagated to trigger the group-by operator to emit a result for this person. This way the group-by operator is able to produce real-time results instead of being blocked indefinitely.

3.3 Optimizations Enabled by Combined Constraints

Either punctuation or sliding window can help shrink the join state. We now show that when they are present simultaneously, further optimizations can be enabled. Such optimizations are not achievable if only one constraint type occurs. We first present a theorem as the foundation of these optimizations.

THEOREM 5.1 *Assume t_i is announced by a punctuation to be the last tuple ever in stream S_i that has value a_k for join attribute A. Once t_i expires from the sliding window, no more join results with $A{=}a_k$ will be generated thereafter.*

Based on this theorem, we derive a *tuple dropping invariant* for dropping new tuples that won't contribute to the join result. This further reduces the join workload without compromising the precision of the join result.

DEFINITION 5.2 **(Tuple Dropping Invariant.)** *Let t_i be the last tuple from stream S_i that ever contains value a_k for the join attribute A and let latestTS be the timestamp of the latest tuple being processed thus far. Drop tuple t_j from stream S_j ($j \neq i$) if $t_i.TS <$ latestTS - W_i and $t_j.A = t_i.A$ and $t_j.TS \geq$ latestTS.*

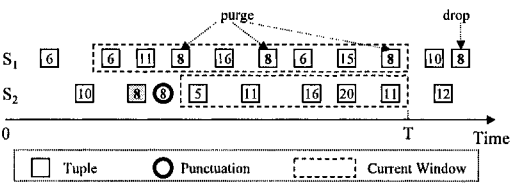


Figure 5.4. Dropping Tuples Based on Constraints.

Figure 5.4 shows an example of applying the tuple dropping invariant. The last tuple with join value 8 in stream S_2 expires from the window at time T. Hence, the tuple dropping invariant is satisfied. In the figure, four tuples in stream S_1 are shown to have join value 8. Three of them arrived before time T so that they have joined with the matching tuples from S_1 and have been purged by the purge operation. Another tuple with this join value is arriving after time T. This tuple can then be dropped with no need to be processed based on the tuple dropping invariant.

Now we consider how the combined constraints assist the punctuation propagation in the join operator. Assume the join receives a punctuation $<a_1, *>$ from S_1, which declares that no more tuples from S_1 will have value a_1 for attribute A. The join, however, may not be able to immediately output a punctuation $<a_1, *, *>$ because tuples with A=a_1 are still potentially coming from S_2. This may render future results with A=a_1. Only when the join state contains no tuple with A=a_1 from stream S_1, we can safely output this punctuation.

We observe that without sliding window, we can only propagate punctuations in a very restrictive case, i.e., punctuations are specified on the join attribute. When the punctuation on a certain join value has been received from both streams, we know that all results with this join value have been produced. The join is then able to propagate this punctuation.

In the presence of both punctuations and sliding windows, a more efficient propagation strategy can be achieved in the *invalidation* operation of the join algorithm. As we invalidate expired tuples from the window, we also invalidate punctuations. When a punctuation from one stream moves out of the window, all tuples from this stream that match this punctuation must have all expired from the window. Therefore the propagation condition is satisfied and this punctuation becomes propagable. Also note that the punctuations propagated by this strategy are not necessary to be on the join attribute.

3.4 Adaptive Component-Based Execution Logic

As described in Section 3.2, the join algorithm may involve numerous tasks: (1) *memory join*, which probes in-memory join state using a new tuple and produces results for any matches, (2) *state relocation*, that moves part of the in-memory state to disk when running out of memory, (3) *disk join*, that retrieves data from disk into memory for join processing, (4) *purge*, that purges no-longer-useful data from the state according to punctuations, and (5) *invalidation*, that removes expired tuples from the state based on the sliding window.

The frequencies of executing each of these tasks may be rather different due to performance considerations. Memory join is executed as long as new tuples are ready to be processed. This guarantees the join result to be delivered as soon as possible. State relocation is applied only when the memory limit is reached. This way the I/O operations are reduced to a minimum. Disk join also involves I/O operations. Hence it is scheduled only when the memory join cannot proceed due to the delays in data delivery. The purge incurs overhead in searching for tuples that satisfy the purge invariant. Depending on how frequently the punctuations arrive, we may choose to run the purge task after receiving a certain number of punctuations (*purge threshold*) or when the memory usage reaches the limit. Similarly, the invalidation task also incurs

overhead in searching for expired tuples. We may decide to conduct this task after processing a certain number of new tuples (*invalidation threshold*).

Due to the dynamic nature of the streaming environment, the threshold associated with these tasks may vary over time. For example, as other queries enter and leave the system, the memory limit of a operator may be decreased and increased accordingly. The traditional join algorithm that follows a fixed sequence of operations is inappropriate for achieving such fine-tuned execution logic. In response, we have devised a component-based adaptive join algorithm to cope with the dynamic streaming environment. For this algorithm, we design five components for accomplishing the five tasks listed above. We also employ an *event-driven* framework to adaptively schedule these components according to certain changes that may affect the performance of the operator.

As shown in Figure 5.5, the memory join is scheduled as long as new tuples are ready to be processed. Meanwhile, an *event generator* monitors a variety of runtime parameters that serve as the component triggering conditions. These parameters include the memory usage of the join state, the number of punctuations that arrived since the last purge, etc. When a parameter reaches the corresponding threshold, e.g., when the memory usage reaches the *memory limit*, a corresponding event will be invoked. Then the memory join is suspended and the registered listener to the invoked event, i.e., one of the components, will be scheduled to run. After the listener finishes its work, the memory join will be resumed. We have defined the following events for the join operator:

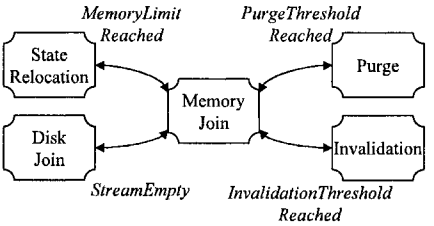


Figure 5.5. Adaptive Component-Based Join Execution Logic.

1 *StreamEmpty* signals that both input streams currently contain no tuple.

2 *PurgeThresholdReached* signals that the number of unprocessed punctuations has reached the *purge threshold.*

3 *MemoryLimitReached* signals the memory used by the join state has reached the *memory limit.*

4 *InvalidationThresholdReached* signals that the number of newly-processed tuples since the last invalidation has reached the *invalidation threshold.*

Table 5.1. Example Event-Listener Registry.

Events	*Conditions*	*Listeners*
StreamEmpty	Activation threshold: 70%.	Disk Join
PurgeThresholdReached	None.	Purge
MemoryLimitReached	There exist unprocessed punctuations.	Purge
MemoryLimitReached	There are no unprocessed punctuations.	State Relocation

The join operator maintains an *event-listener registry*. Each entry in the registry lists the event being generated, additional conditions to be checked and the listener (component) which will be executed to handle the event. The registry is initiated at the static query optimization phase and can be updated at runtime. The thresholds used for invoking the events are specified in the *event generator*. They can be changed at runtime. Table 5.1 shows an example registry for a join with no sliding window applied. Hence, no entry in the registry corresponds to the invalidation component.

3.5 Summary of Performance Evaluation

From our experimental study on the join operator in CAPE [Ding et al., 2004; Ding et al., 2003], we have obtained the following observations:

- By only exploiting punctuations, in the best case the join state consumes nearly constant memory. The shrinkage in state also helps improve the tuple output rate of the join operator because the probe operation can now be done more efficiently.

- In terms of the sliding window join, if the window contains a large amount of tuples, by in addition exploiting punctuations, the memory consumption of the join state is further reduced and the tuple output rate increases accordingly due to the tuple dropping.

- The adaptive execution logic enables the join operator to continue outputting results even when the data delivery experiences temporary delay. In addition, the appropriate purge threshold and invalidation threshold settings help the join operator to achieve a good balancing between the memory overhead and the tuple output rate.

4. Adaptive Execution Scheduling

Rather than randomly select operators to execute or leave such execution ordering up to the underlying operating system, stream processing systems aim to have fine-grained control over the query execution process. In response, scheduling algorithms are being designed that decide on the order in which

the operators are executed. They target specific optimization goals, such as increasing the output rate or reducing the memory usage.

4.1 State-of-the-Art Operator Scheduling

Current stream processing systems initially employed traditional scheduling algorithms borrowed from the realm of operating systems [Dan and Towsley, 1990; Zahorjan and McCann, 1990], such as Round-Robin and FIFO. More recently, customized algorithms designed specifically for continuous query evaluation have been proposed, including Chain [Babcock et al., 2003] in STREAM and Train [Carney et al., 2003] in Aurora. The Chain scheduling strategy is designed with the goal of minimizing intermediate queue sizes, thereby minimizing the memory overhead. However, it is not targeting at meeting other Quality of Service (QoS) requirements. The Train scheduling algorithms, four in total, are variations each tuned for a particular QoS criterion.

We have experimentally compared these popular algorithms under a variety of stream workloads within the CAPE testbed. This experimental study [Sutherland et al., 2004a] reveals that each of these algorithms is good at improving the system performance in one specific manner, e.g., Chain for reducing memory usage and FIFO for increasing the result output rate. Thus, even though many scheduling algorithms exist in the literature, there is no one algorithm that a system can utilize to satisfy the diversity of system requirements common to stream systems. In other words, it is difficult to design a scheduling algorithm that always functions effectively even when experiencing a wide variety of changing conditions, including changing QoS requirements, the addition of new queries or runtime query plan reoptimization (See Section 5).

The existing stream systems usually select one scheduling algorithm at the beginning of the query execution and then stick with it. This overlooks the fact that as the stream environment experiences changes, the initially optimal scheduling algorithm may become sub-optimal over time. One possible solution to this dilemma may be to put a human administrator in charge of the decision on selecting scheduling algorithms. However, it is often impossible for an administrator to know a priori which scheduling algorithm to pick, or more challenging even which algorithm to turn on or off at runtime based on the behavior of the system. It is exactly this issue of automating the scheduling algorithm selection that we address in CAPE.

4.2 The ASSA Framework

We have designed a framework for the *Adaptive Selection of Scheduling Algorithms* (ASSA for short) [Sutherland et al., 2004a]. The ASSA architecture is depicted in Figure 5.6. ASSA is equipped with a library of scheduling algorithms, ranging from well established ones like Round Robin and FIFO to

recently proposed ones like Chain and Train. As new scheduling algorithms are developed, they can easily be plugged into this library.

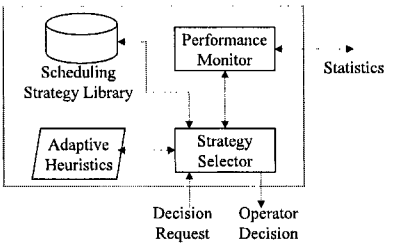


Figure 5.6. Architecture of ASSA Scheduler.

In a nutshell, having several algorithms at its avail with each targeting different QoS requirements, the *Strategy Selector* dynamically selects one scheduling algorithm from the library for scheduling the execution of the operators in a plan. For this, the *Performance Monitor* must establish some measures by which the algorithms can be compared and ranked in terms of their expected effectiveness. ASSA learns about the impact of each algorithm on the query system by observing how well each algorithm has done thus far during execution. This learned knowledge is encoded into a score for each algorithm.

The *Strategy Selector* then utilizes those learned scores to guide the selection of the candidate algorithm. It applies a lightweight *Adaptive Heuristic*, also called Roulette Wheel heuristic [Mitchell, 1996], that selects the next candidate algorithm to use for scheduling based on its perceived potential to fulfill the current set of QoS requirements. ASSA then simply asks this selected algorithm to pick the next operator to execute. Lastly this decision is then reported to the Execution Engine which carries out the control of the actual execution flow.

4.3 The ASSA Strategy: Metrics, Scoring and Selection

We now describe the specified QoS requirements that the ASSA selector utilizes to assess the effectiveness of a scheduler. We also discuss the fitness score assigned to each scheduler to capture how well it performed relative to the other algorithms.

Quality of Service Requirements. Our system allows for the system administrator to specify the desired execution behavior as a composition of several metrics. A QoS requirement consists of three components: the statistic, quantifier, and weight. The statistic corresponds to the metric that is to be controlled. Performance metrics considered include throughput (the number of result tuples produced), memory requirements, and freshness of results (the

amount of time a tuple stays in the system). The quantifier, either *maximize* or *minimize*, specifies what the administrator wants to do with this preference. The weight is the relative importance of each requirement, with the sum of all weights equal to 1. We combine all of the QoS requirements into a single set called a QoS specification. This specification is our indicator of how we want the system to perform overall. Table 5.2 shows an example QoS specification. Here, the administrator has specified that the system should give highest priority to minimizing the queue size and next highest to maximizing the output rate.

Table 5.2. An example QoS specification.

Statistic	Quantifier	Weight
Input Queue Size	minimize	0.75
Output Rate	maximize	0.25

QoS requirements guide the adaptive execution by encoding a goal that the system should pursue. Without these preferences, the system will not have any criteria by which to determine how well or poorly a scheduler is performing. The requirements specify the desired behavior in relative terms, such as maximize the output rate or minimize the queue size(s) and their relative importance. Absolute requirements are too dependent on data arrival patterns and in fact in many cases are simply not achievable.

Scoring the Scheduling Algorithms. During execution, the Execution Engine will update the statistics that are related to the QoS requirements. Once updated, the system needs to decide how well the previous scheduler, S_{old}, has performed, and compare this performance to that of the other scheduling algorithms. To accomplish this, a function is developed to quantify how well an algorithm is performing for a particular QoS metric. First, the system calculates the mean and the spread of the values of each of the statistics specified in the service preferences for each category. Next, using the statistics from S_{old}, the relative mean of each of the statistics is calculated and then normalized.

The scoring function weighs the individual QoS metrics for relative importance (by multiplying by its corresponding weight w_i) and then normalizes the collected statistics for those metrics such that one algorithm can be ranked against another. We compute an algorithm's overall score, *scheduler_score*, by combining the relative performance for all of the QoS metrics into one QoS specification. The score assigned to an algorithm is not based solely on the previous time that it was used, but rather it is an exponentially smoothed average value over time. By comparing S_{old}'s *scheduler_score* with the scores for the

other algorithms, the adapter is in a position to select the next most promising scheduling candidate.

Guidelines for Adaptation. Several guidelines are considered when using the scores to determine the next scheduling algorithm. Initially, all scheduling algorithms should be given a chance to "prove" themselves. Otherwise the decision would be biased against the algorithms that have not yet run. Therefore, at the beginning of execution, we allow some degree of exploration on the part of the adapter. Second, not switching algorithms periodically during execution (i.e., greedily choosing the next algorithm to run) could result in a poor performing algorithm being run more often than a potentially better performing one. Hence, we periodically explore alternate algorithms. Third, switching algorithms too frequently could cause one algorithm to impact the next and skew the latter's results. For example, using Chain could cause a glut of tuples in the input queues of the lower priority operators. If a batch-tuple strategy were to be run next, its throughput would initially be artificially inflated because of the way Chain operated on the tuples. More generally, when a new algorithm is chosen, it should be used for enough time such that its behavior is not significantly over-shadowed by the previous algorithm. For this, we empirically set delay thresholds before reassessing the potential of a switch to be undertaken.

Adaptive Selection Process. After each algorithm is given a score, the system needs to decide if the current scheduling algorithm performed well enough that it should be used again or if better performance may be achieved by changing algorithms. Considering Guideline 1 above, initially running each algorithm in a round robin fashion is the fairest way to start adaptive scheduling.

Once each algorithm has had a chance to run, there are various heuristics that could be applied to determine if it would be beneficial to change the scheduling algorithm. In an effort to consider all scheduling algorithms while still probabilistically choosing the best fit we adopted the Roulette Wheel strategy. This strategy assigns each algorithm a slice of a circular "roulette wheel" with the size of the slice being proportional to the individual's score. Then the wheel is spun once and the algorithm under the wheel's marker is selected to run next. This strategy was chosen because it is lightweight and does not cause significant overhead. In spite of its simplicity, this strategy is shown to significantly outperform single scheduling strategies (See Section 4.4). While this strategy may initially choose poor scheduling algorithms, over time it should fairly choose a more fit algorithm. The strategy also allows for a fair amount of exploration and thus it prevents one algorithm from dominating.

4.4 Summary of Performance Evaluation

An extensive experimental study on performance of ASSA can be found in [Sutherland et al., 2004a]. We now briefly summarize the overall observations from this study:

- For the special case of a QoS specification consisting of only one single metric, ASSA indeed picks the one most optimal algorithm from all available algorithms in the library.
- For a complex QoS specification combining multiple requirements, ASSA also significantly improves performance over the run of any individual algorithm by working with some combination of algorithms.
- ASSA is able to react to QoS requirements even as they are changed at runtime by the system administrator.
- The overhead for the adaptation itself, i.e., the score calculation and the switching among algorithms, is shown to be negligible.
- ASSA is shown to be general, i.e., new scheduling solutions developed in the future can be plugged into the library of ASSA at any time.

5. Run-time Plan Optimization and Migration

Query plan optimization is critical for improving query performance. In a stream processing system, data is not present at the time when a query starts but is streaming in as time goes by. The long-running continuous queries have to withstand fluctuations in stream workload and data characteristics. Therefore, compared to static query processing system, a stream processing system has a much more pressing need to re-optimize the continuous query plans at run-time. A run-time plan optimization procedure takes three steps:

- ***Step 1:*** The optimizer decides *when* to invoke the optimization procedure. Too frequent optimization creates extra burden on the system resources, and too infrequent optimization may skip good optimization opportunities and hurt the system performance as well. The timing of the optimization is critical and needs to be carefully tuned. We present the solution in CAPE for this issue in Section 5.1.
- ***Step 2:*** The optimizer constructs a new query plan that is semantically equivalent to the currently running plan yet more efficient in terms of system resource consumption or performance. This is done by applying heuristics and rewriting rules to the old query plan based on gathered system statistics. We will discuss the optimization heuristics in CAPE in Section 5.2.

- ***Step 3:*** The optimizer migrates the old running query plan to the new plan that it has chosen. We refer to this process as *dynamic plan migration*. A novel feature of the run-time optimizer in CAPE, not yet offered by other stream engines, is that it can efficiently at run-time migrate a stream query plan even if it contains stateful operators. This dynamic plan migration step is the critical service that enables optimization to occur at runtime for stream query processing. We will discuss dynamic plan migration in Sections 5.3 and 5.4.

5.1 Timing of Plan Re-optimization

In the CAPE system, the plan optimization procedure can be invoked in two modes: the *periodic mode* and the *event-driven mode*.

In the periodic mode, the optimizer is invoked at a pre-specified optimization interval. As mentioned in Section 2, we adopt a heterogeneous-grained adaptation framework, in which each adaptation technique is assigned an adaptation interval based on its overhead and its perceived potential gain. Since the cost of the plan re-structuring is usually between the costs of the operator scheduling (which is very low effort) and the across-machine plan re-distribution (which is a more involved effort), so is its adaptation interval. The optimization interval can also be tuned dynamically based on certain changes in streaming data arrival rates or data distributions.

The CAPE system also identifies types of events that represent critical optimization opportunities that are unique to a stream processing system, as detailed in Section 5.2. Whenever one of the events occurs, it will trigger the optimizer running in the periodic mode to switch to the event-driven mode. The optimizer then immediately reacts to the triggering event by taking the corresponding actions typically in the form of applying customized heuristics. Once the optimization has completed, the optimizer returns back to its default mode, i.e., the periodic mode.

5.2 Optimization Opportunities and Heuristics

Many commonly used heuristics and rewriting rules in static database are also applicable for continuous query optimization. In CAPE, an optimizer in the periodic mode for example applies the following heuristics:

- The optimizer pushes down the select and project operators to minimize the amount of data traveling through the query plan, unless sharing of partial query plans dictates a delay of such early filtering.

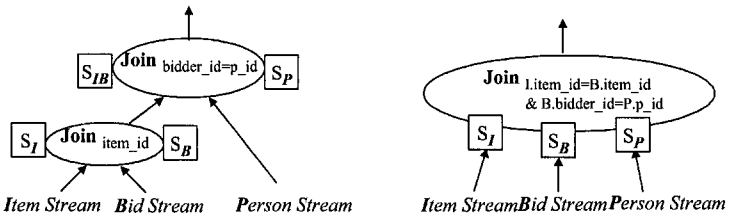
- The optimizer merges two operators into one operator whenever possible, such as merging two select operators or merging a group-by operator with

an aggregate operator, to reduce scan of data via shared data access and to avoid context switching.

- The optimizer switches two operators based on their selectivities and processing overhead. If Sel_i and Cost_i represent the selectivity and the processing cost of an operator op_i, then operators op_i and op_j with op_i consuming data produced by op_j can be switched if $(1 - Sel_i)/Cost_i > (1 - Sel_j)/Cost_j$.

We have also identified several new optimization opportunities that are unique to the stream processing system and its dynamic environment. We have incorporated these stream-specific optimization heuristics into CAPE as well.

Register/De-register Continuous Queries. A stream processing system often needs to execute numerous continuous queries at the same time. Sharing among multiple queries can save a large amount of system resources. In addition, queries may be registered into or de-registered from the system at any time. The above features can affect the decision of the optimizer. As an example, assume the system currently has one query plan with one select and one join operator, and after a while another query is registered which contains the same join as the first query but no select. In this case, the optimizer can pull up the select operator so the two queries can share the results from the join operator. Later if the second query is de-registered from the system, the optimizer may need to push the select down again. So the events of query registering/de-registering create new optimization opportunities that CAPE utilizes to trigger heuristics for query-sharing optimizations.



(a) Binary Join Tree (b) Multi-way Join

Figure 5.7. A Binary Join Tree and A Multi-way Join Operator.

Multi-Join Queries. Choosing the optimal order of multiple join operators has always been a critical step in query optimization. There are two popular methods to process a continuous query with multiple joins: a binary join tree as in traditional (static) databases, and a single *multi-way join* operator [Golab and Ozsu,

2003; Viglas et al., 2003]. For the two joins $Join_{I.item\_id=B.item\_id\ \&\ B.bidder\_id=P.p\_id}$ in the query defined in Figure 5.3, Figures 5.7 (a) and (b) depict a query plan composed of several binary joins and out of one multi-way join operator respectively. A binary join tree stores all intermediate results in its intermediate states, so no computation will be done twice. On the other hand, a multi-way join operator does not save any intermediate results, so all useful intermediate results need to be recomputed. A binary join tree saves CPU time by sacrificing memory, while a multi-way join sits on the opposite end of the spectrum. Dynamically switching between these two achieves different balancing between CPU and memory resources. The CAPE system monitors the usage of CPU and memory while processing multiple joins. When the ratio of CPU to memory is greater or smaller than some pre-defined threshold, the optimizer enters the event-driven mode and switches between these two methods accordingly.

Punctuation-Driven Plan Optimization. The characteristics of punctuations available to the query also affect the plan structuring decision making. Considering the example query shown in Figure 5.3, the two join operators in the query plan are commutative, hence rendering two semantically equivalent plans. In the plan shown in Figure 5.3, some join results of the first join may not be able to join with any Person tuple in the second join because they don't satisfy the sliding window constraint applied to the Person stream. If we choose to do (Bid⋈Person) first, the sliding window constraint will drop expired tuples early so to avoid unnecessary work in the later join. However, in this plan, both join operators need to propagate punctuations on the Person.p_id attribute to help the group-by operator. This incurs more propagation overhead than the first plan in which only the second join needs to propagate punctuations. The optimizer in CAPE will choose the plan with less cost by considering these factors related to punctuation-propagation costs and punctuation-driven unblocking.

5.3 New Issues for Dynamic Plan Migration

Dynamic plan migration is the key service that enables plan optimization to proceed at runtime for stream processing. It is a unique feature offered by the CAPE system. Existing migration methods instead adopt a *pause-drain-resume* strategy that pauses the processing of new data, drains all old data from the intermediate queues in the existing plan, until finally the new plan can be plugged into the system.

The *pause-drain-resume* migration strategy is adequate for dynamically migrating a query plan that consists of only *stateless* operators (such as select and project), in which intermediate tuples only exist in the intermediate queues. On the contrary, a *stateful* operator, such as join, must store all tuples that have been processed thus far to a data structure called a *state* so to be able to join them with future incoming tuples. Several strategies have been proposed to purge

unwanted tuples from the operator states, including window-based constraints [Carney et al., 2002; Hammad et al., 2003; Kang et al., 2003; Motwani et al., 2003] and punctuation-based constraints [Ding et al., 2004; Tucker et al., 2003] (See Section 3). In all of these strategies the purge of the old tuples inside the state is driven by the processing of new tuples or new punctuations from input streams.

For a query plan that contains *stateful* operators, the draining step of the pause-drain-resume step can only drop the tuples from the intermediate queues, not the tuples in the operator states. Those could only be purged by processing new data. Hence there is a dilemma. CAPE offers a unique solution for stateful runtime plan migration. In particular, the we have developed two alternate migration strategies, namely *Moving State Strategy* and *Parallel Track Strategy*. These strategies are now introduced below.

5.4 Migration Strategies in CAPE

Below, the term *box* is used to refer to the plan or sub-plan selected for migration. The migration problem can then be defined as the process of transferring an old box containing the old query plan to a new box containing the new plan. The old and new query plans must be equivalent to each other, including identical sets of box input and output queues, as shown in Figure 5.8.

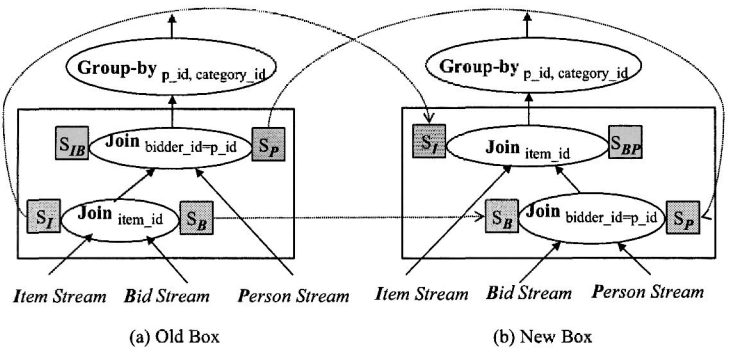


Figure 5.8. Two Exchangeable Boxes.

Moving State Strategy. The moving state strategy first pauses the execution of the query plan and drains out tuples inside intermediate queues, similar to the above *pause-drain-resume* approach. To avoid loss of any useful data inside states, it then takes a three-step approach to safely move old tuples in old states directly into the states in the new box. These steps are *state matching*, *state moving* and *state recomputing*.

State matching determines the pairs of states, one in the old and one in the new box, between which tuples can be safely moved. If two states have the same state ID, which are defined as the same as its tuples' schema, we say that those two states are *matching states*. In Figure 5.8, states (S_I, S_B, S_P) exist in both boxes and are matching states. During the step of *state moving*, tuples are moved between all pairs of matching states. This is accomplished by creating a new cursor for each matching new state that points to its matching old state, such that all tuples in the old state are shared by both matching states. The cursors for the old matching states are then deleted. In the *state recomputing* step, the unmatched states in the new box are computed recursively from the leaves upward to the root of the query plan tree. Since the two boxes have the same input queues, the states at the bottom of the new box always have a matching state in the old box. Using the example shown in Figure 5.8, we have identified an unmatched state S_{BP} in the new box. We can recompute S_{BP} by joining the tuples in S_B and S_P.

Once the moving state migration starts, no new results are produced by the targeted migration box inside the larger query plan until the migration process is finished. Of course, the remainder of the query plan continues its processing. This way the output stream may experience a duration of temporary silence. For applications that desire a smooth and constant output, CAPE offers a second migration strategy called the parallel track strategy. This alternate strategy can still deliver output tuples even during migration.

Parallel Track Strategy. The basic idea for the *parallel track migration strategy* is that at the migration start time, the input queues and output queue are connected and shared between the old box and the new box. Both boxes are then being executed in parallel until all old tuples in the old box have been purged. During this process, new outputs are still being continually produced by the query plan. When the old box contains only *new* tuples, it is safe to discard the old box. Because the new box has been executed in parallel with the old box from the time the migration first starts, all the new tuples now in the old box exist in the new box as well. So if the old box is discarded at this time, no useful data will be lost.

A valid migration strategy must ensure that no duplicate tuples are being generated. Since the new box only processes *new* tuples fed into the old box at the same time, all output tuples from the new box will have only *new* sub-tuples. However, the old box may also generate the all-new tuple case, which may duplicate some results from the new box. To prevent this potential duplication, the root operator of the old box needs to avoid joining tuples if all of them are *new* tuples. In this way, the old box will not generate the all-new tuples.

Cost of Migration Strategies Detailed cost models to compute the performance overhead as well as migration duration periods have been developed [Zhu et al., 2004]. This enables the optimizer in CAPE to compute the cost of these two

strategies based on gathered system statistics, and then dynamically choose the strategy that has the lowest overhead at the time of migration.

The two migration strategies have been embedded into the CAPE system. While extensive experimental studies comparing them can be found in [Zhu et al., 2004], a few observations are summarized here:

- Given sufficient system resources, the moving state strategy tends to finish the migration stage quicker than parallel track.
- However, if the system has insufficient processing power to keep up with the old query plan, the parallel track strategy, which can continuously output results even during the migration stage, is observed to have a better output rate during the migration stage.
- Overall, the costs of both migration strategies are affected by several parameters, including the stream arrival rates, operator selectivities and sizes of the window constraints.

6. Self-Adjusting Plan Distribution across Machines

While most current stream processing systems (STREAM [Motwani et al., 2003], TelegraphCQ [Chandrasekaran et al., 2003], and Aurora [Abadi et al., 2003]) initially have employed their engine on a single processor, such an architecture is bound to face insurmountable resource limitations for most real stream applications. A distributed stream architecture is needed to cope with the high workload of registered queries and volumes of streaming data while serving real-time results [Cherniack et al., 2003; Shah et al., 2003]. Below we describe our approach towards achieving a highly scalable framework for distributed stream processing, called Distributed CAPE (D-CAPE in short) [Sutherland et al., 2004b].

6.1 Distributed Stream Processing Architecture

D-CAPE adopts a Shared-Nothing architecture shown to be favorable for pipelined parallelism [DeWitt and Gray, 1992]. As depicted in Figure 5.9, D-CAPE is composed of a set of query processors (in our case CAPE engines) and one or more distribution managers. We separate the task of distribution decision making ("control") from the task of query processing to achieve maximal scalability. This allows all query engines to dedicate 100% of their resources to the query processing. The distribution decision making is encapsulated into a separate module called the Distribution Manager. A Distribution Manager typically resides on a machine different from those used as query processors, though this is not mandatory.

D-CAPE is designed to efficiently distribute query plans and continuously monitor the performance of each query processor with minimal communica-

tion between the controller and query processors. At runtime, during times of heavy load or if it is determined by D-CAPE that reallocation will boost the performance of the system, query operators are seamlessly reallocated to different query processors. By multi-tiering distribution managers, we can exploit clusters of machines in different locations to process different workloads.

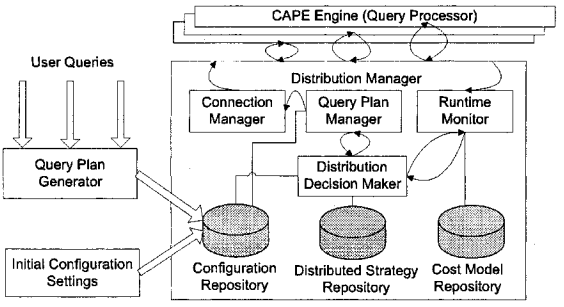


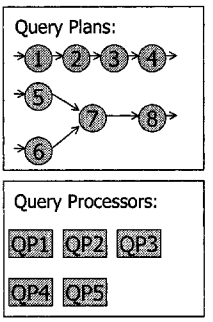
Figure 5.9. Distribution Manager Architecture.

As illustrated in Figure 5.9, the Distribution Manager is composed of four core components and three repositories. The *Runtime Monitor* listens for periodic statistics reported by each query processor, and records them into the *Cost Model Table*. These run-time statistics form the basis for determining the *workload* of processors and for deciding operator reallocation. The *Connection Manager* is responsible for physically sending a sequence of appropriate connection messages according to our connection protocol to establish operators to machines. The *Query Plan Manager* manages the query plans registered by the user in the system. The *Distribution Decision Maker* is responsible for deciding *how* to distribute the query plans. There are two phases to this decision. First, an initial distribution is created at startup using static information about query plans and machine configurations. Second, at run-time query operators are reallocated to other query processors depending on how well the query processors are perceived to be performing by the Decision Maker.

The Distribution Manager is designed to be light-weight. Only incremental changes of the set of query plans are sent to the query processors to reduce the amount of time the Distribution Manager spends communicating with each processor at run-time. Our empirical evaluation of the Distribution Manager shows that the CPU is rarely used, primarily, only when calculating new distribution ([Sutherland et al., 2004b]). Furthermore, the network traffic the DM creates is minimal. In short, this design of D-CAPE is shown to be highly scalable.

6.2 Strategies for Query Operator Distribution

Distribution is defined as the physical layout of query operators across a set of query processors. The initial distribution of a query plan based only on static information at query startup time is shown to directly influence the query processing performance. The initial distribution depends only on two pieces of information: the queries to be processed and the machines that have the potential to do the work. The Distribution Decision Maker accepts both the description of the query processors and query plans as inputs and returns a table known as a *Distribution Table* (Figure 5.10). This table captures the assignment of each query plan operator to the query processor it will be executing on.



Distribution Table

Operator	QP
Operator 1	QP 1
Operator 2	QP 1
Operator 3	QP 2
Operator 4	QP 3
Operator 5	QP 4
Operator 6	QP 4
Operator 7	QP 4
Operator 8	QP 5

Figure 5.10. Distribution Table.

The methodology behind how the table is created depends on the Distribution Pattern utilized by the Decision Maker. This allows us the flexibility to easily plug in any new Distribution Pattern into the system. Two distribution algorithms that were initially incorporated into D-CAPE are:

- **Round Robin Distribution.** It iteratively takes the next query operator and places it on the still most "available" query processor, i.e., the one with the fewest number of assigned operators. This ensures fairness in the sense that each processor must handle an equal number of operators, i.e., an equivalent workload.

- **Grouping Distribution.** It takes each query plan and creates sub-plans for each query by maximally grouping neighboring operators together so to construct connected subgraphs. This aims to minimize network connections since adjacent operators with joint "pipes" are for the most part kept on the same processor. Then it divides these connected subgraphs among the available query processors.

After a distribution has been recorded into the distribution table, then the Connection Manager distributes the query plan among the query processors. Once the Connection Manager has completed the initial setup, query execution can begin on the cluster of query processors, now no longer requiring any interaction from the Distribution Manager.

6.3 Static Distribution Evaluation

Our work is one of the first to report experimental assessments on a working distributed stream processing system. For details, the readers are referred to [Sutherland et al., 2004b]. Below we list a summary of our findings:

- D-CAPE effectively parallelizes the execution of queries, improving performance even for small query plans and for lightly loaded machines, without ever decreasing performance beyond the central solution.
- The total throughput is improved when using more query processors over that when using less processors. This is because we can assign a larger CPU time slice to each operator.
- The larger the query plans in terms of number and type of operators, the higher a percentage of performance improvement is achievable when applying distribution (on the order of several 100%). In many cases while the centralized CAPE fails due to resource exhaustion or a lack of processing power, the distribution solution continues to prevail.
- The Grouping Distribution generally outperforms the Round Robin by several fold. In part, this can be attributed to the Grouping Distribution being "connection-aware", i.e., due to it minimizing the total amount of data sent across the network and the number of connections.

6.4 Self-Adaptive Redistribution Strategies

When we first distribute a query plan, we only know static information such as shape and size of the query plan, the number of input streams, and data about the layout of the processing cluster. Dynamic properties such as state size, selectivity, and input data rates are typically not known until execution. Worse yet, these run-time properties tend to change over time during execution.

Due to such fluctuating conditions, D-CAPE is equipped with the capability to monitor in a non-obtrusive manner its own query performance, to self-reflect and then effectively redistribute query operators at run-time across the cluster of query processors. We will allow for redistribution among *any* of the query processors, not just adjacent ones, in our computing cluster.

Towards this end, we require a measure about the relative *query processor workload* that is easily observable at runtime. One such measure we work with

is the rate at which tuples are emitted out of each processor onto the network. This dynamically collected measure is utilized by the on-line redistribution policy in D-CAPE for deciding *if*, *when* and *how* to redistribute.

Algorithm 1 Overall Steps for Redistribution.

```
1: costTable ← costModel.getTable()
2: maxCost ← costTable.getMaxCost()
3: minCost ← costTable.getMinCost()
4: if max − min > redistributionPercent
5:  while !valid(newDistribution) do
6:  newDistribution ← RedistributionPolicy.redistribute()
7: end while
8: differenceTable ← newDistribution − currentTable
9: connectNewDistribution(differenceTable)
10: currentTable ← newDistribution
11: end if
```

While new policies can be easily plugged into D-CAPE framework, one of the redistribution policies we found to be effective in D-CAPE is called the *degradation redistribution policy*. This policy alleviates load on machines that have shown a degradation in cost since the last time operators were allocated to the machine. If the cost has degraded beyond a certain threshold, we aim to stop this degradation by moving the 'most costly' operators to other query processors. This policy gives higher preference to those operators that will remove a network connection from the overall distribution of operators — driven by our empirical observation of the direct impact of higher connection loads on the resulting system performance.

In general, any of the redistribution policies in D-CAPE, including the degradation policy above, employs the steps detailed in Algorithm 1 for realizing the desired re-distribution. These steps use our our handshake protocol between the Distribution Manager and the designed for moving query operators between processors The cost of moving an operator has been shown to be negligible in our system due to this carefully designed connection protocol. Intuitively, since we create the connections for the data to flow *before* we start sending the data, we are able to "flip a switch" and in the eyes of the query processor, turn off one operator and turn it on on another machine instantaneously.

6.5 Run-Time Redistribution Evaluation

Our results confirm that dynamic redistribution is a viable and even necessary option for handling the performance degradation observed at runtime. Our results illustrate that redistribution can tune the execution if the initial distribution is found to be bad or if it turns bad over time. While a detailed experimental study can be found in [Sutherland et al., 2004b], key experimental observations are shown below.

- The overhead for redistributing an operator or even a complete sub-plan across machines is found to be negligible. This allows D-CAPE to perform reallocation at a high frequency, if deemed necessary.

- Even strategies that achieve good initial distribution patterns such as the Grouping Distribution can still experience a further performance boost when undergoing runtime redistribution.

- On-line operator re-allocation has been shown to improve performance over time compared to only working with static distributions.

- Initial static distribution decisions significantly affect the performance in the long-term even when continuing to dynamically apply reallocation.

7. Conclusion

In this chapter, we have presented a streaming query processing system named CAPE. We reviewed the query optimization techniques that are unique to CAPE, including the heterogeneous-grained adaptation framework and constraint-exploiting techniques. The adaptation technologies we illustrated include the adaptive operator execution logic, self-healing adaptive operator scheduling, runtime query plan re-optimization and migration, and self-adjusting query plan distribution across machines. CAPE employs these technologies to effectively evaluate continuous queries in highly dynamic streaming environments.

References

D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, August 2003.

A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *DBPL*, pages 1–19, Sep 2003.

B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *ACM SIGMOD*, pages 253–264, 2003.

S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. *ACM Transactions on Database Systems*, 39(3), Sep 2004.

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 *VLDB*, pages 215–226, August 2002.

S. Chandrasekaran, O. Cooper, A. Deshpande, M. Franklin, J. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah.

TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, pages 269–280, Jan 2003.

M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *CIDR*, 2003.

D. Carney and U. Cetintemel and A. Rasin et al. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

A. Dan and D. Towsley. An approximate analysis of the lru and fifo buffer replacement schemes. In *ACM SIGMETRICS*, pages 143–152, 1990.

D. J. DeWitt and J. Gray. Parallel database systems: The future of high performance database systems. *Communications of the ACM*, 35(6):85–98, 1992.

L. Ding, N. Mehta, E. A. Rundensteiner, and G. T. Heineman. Joining punctuated streams. In *EDBT*, pages 587–604, March 2004.

L. Ding, E. A. Rundensteiner, and G. T. Heineman. MJoin: A metadata-aware stream join operator. In *DEBS*, June 2003.

L. Golab and M. T. Ozsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB*, pages 500–511, Sep 2003.

P. Haas and J. Hellerstein. Ripple joins for online aggregation. In *ACM SIGMOD*, pages 287–298, June 1999.

M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *VLDB*, pages 297–308, Sep 2003.

J. Kang, J. F. Naughton, and S. D. Viglas. Evaluating window joins over unbounded streams. In *ICDE*, pages 341–352, March 2003.

S. Madden and M. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE*, pages 555–566, Feb 2002.

M. Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1996.

M. F. Mokbel, M. Lu, and W. G. Aref. Hash-merge join: A non-blocking join algorithm for producing fast and early join results. In *ICDE*, pages 251–262, Mar/Apr 2004.

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 approximation in a data stream management system. In *CIDR*, pages 245–256, Jan 2003.

E. A. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, and N. Mehta. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB Demo*, Aug/Sep 2004, to appear.

M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

Stanford University. Stream query repository. http://www-db.stanford.edu/stream/sqr/, Dec 2002.

T. Sutherland, B. Pielech, and E. A. Rundensteiner. Adaptive scheduling framework for a continuous query system. Technical Report WPI-CS-TR-04-16, Worcester Polytechnic Institute, April 2004.

T. Sutherland and E. A. Rundensteiner. D-cape: A self-tuning continuous query plan distribution architecture. Technical Report WPI-CS-TR-04-18, Worcester Polytechnic Institute, July 2004.

P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras. Exploiting punctuation semantics in continuous data streams. *IEEE Transactions on Knowledge and Data Engineering*, 15(3):555–568, May/June 2003.

T. Urhan and M. Franklin. XJoin: A reactively scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, 2000.

S. Viglas, J. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information. In *VLDB*, pages 285–296, Sep 2003.

A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):103–128, 1993.

J. Zahorjan and C. McCann. Processor scheduling in shared memory multiprocessors. In *ACM SIGMETRICS*, pages 214–225, 1990.

Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic plan migration for continuous queries over data streams. In *ACM SIGMOD*, June 2004.

Chapter 6

EFFICIENT SUPPORT FOR TIME SERIES QUERIES IN DATA STREAM MANAGEMENT SYSTEMS

Yijian Bai, Chang R. Luo, Hetal Thakkar, and Carlo Zaniolo
Computer Science Department
UCLA
bai,lc,hthakkar,zaniolo@cs.ucla.edu

Abstract There is much current interest in supporting continuous queries on data streams using generalizations of database query languages, such as SQL. The research challenges faced by this approach include (i) overcoming the expressive power limitations of database languages on data stream applications, and (ii) providing query processing and optimization techniques for the data stream execution environment that is so different from that of traditional databases. In particular, SQL must be extended to support sequence queries on time series, and to overcome the loss of expressive power due to the exclusion of blocking query operators. Furthermore, the query processing techniques of relational databases must be replaced with techniques that optimize execution of time-series queries and the utilization of main memory. The Expressive Stream Language for Time Series (ESL-TS) and its query optimization techniques solve these problems efficiently and are part of the data stream management system prototype developed at UCLA.

1. Introduction

There is much ongoing research work on data streams and continuous queries [Babcock et al., 2002; Golab and Ozsu, 2003]. The Tapestry project [Barbara, 1999; Terry, 1992] was the first to focus on the problem of 'queries that run continuously over a growing database'. Recent work in the TelegraphCQ project [Chandrasekaran and Franklin, 2002; Madden et al., 2002] focuses on efficient support for continuous queries and the computation of traditional SQL-2 aggregates that combine streams flowing from a network of nodes. The Tribeca system focuses on network traffic analysis [Sullivan, 1996] using operators

adapted from relational algebra. The OpenCQ [Liu et al., 1999] and Niagara Systems [Chen et al., 2000] support continuous queries to monitor web sites and similar resources over the network, while the Chronicle data model uses append-only ordered sets of tuples (chronicles) that are basically data streams [Jagadish et al., 1995].

The Aurora project [Carney et al., 2002] aims at building data management systems that provide integrated support for

- Data streams applications, that continuously process the most current data on the state of the environment.
- Applications on stored data (as in traditional DBs)
- Spanning applications that combine and compare incoming live data with stored data. This requires balancing real-time requirements with efficient processing of large amounts of disk-resident data.

The learning curve and complexity of writing spanning applications can be minimized if SQL is used on both data bases and data streams. This observation justifies the choice of SQL as query language made by most research projects on data streams; however, these projects often underestimate the challenges faced by SQL in this new role. For instance, the design of a general-purpose data stream language and system is the stated objective of the CQL [Arasu et al., 2002] project, which introduces several SQL-based constructs with rigorous semantics [Arasu et al., 2002]. Yet, CQL appears to be effective only for simple queries, and lacks the ability of supporting mining queries, sequence queries, and even some of the monotonic queries expressible in SQL which are discussed next[1].

The expressive power challenge faced by continuous query languages was elucidated in [Law et al., 2004] where it was shown that (i) queries can be expressed by nonblocking computations iff they are monotonic, and that (ii) relational algebra (RA) and SQL are not relationally complete on data streams, since some monotonic queries[1] of relational algebra can only be expressed by RA or SQL by their blocking operators (which must be disallowed on data streams). Moreover, the seriousness of SQL problems proven by the theory are surpassed by those experience in practice, where we find that SQL cannot support many important classes of applications, including data mining and sequence queries.

The limitations of SQL with time-series queries are well-known, and have been the focus of many database research projects aiming at supporting time-series analysis and the search for interesting patterns in stored sequences [Informix, 1998; Ramakrishnan et al., 1998; Seshadri et al., 1994; Seshadri and Swami, 1995; Seshadri, 1998; Perng and Parker, 1999]. Informix [Informix, 1998] was the first among commercial DBMSs to provide special libraries

for time-series, that they named datablades; these libraries consist of functions that can be called in SQL queries. While other database vendors were quick to embrace it, this procedural-extension approach lacks expressive power and amenability to query optimization. To solve these problems, the SEQ and PREDATOR systems introduce a special sublanguage, called SEQUIN for queries on sequences [Seshadri et al., 1994; Seshadri and Swami, 1995; Seshadri, 1998]. SEQUIN works on sequences in combination with SQL working on standard relations; query blocks from the two languages can be nested inside each other, with the help of directives for converting data between the blocks. SEQUIN's special algebra makes the optimization of sequence queries possible, but optimization between sequence queries and set queries is not supported; also its expressive power is still too limited for many application areas. To address these problems, SRQL [Ramakrishnan et al., 1998] augments relational algebra with a sequential model based on sorted relations. Thus sequences are expressed in the same framework as sets, enabling more efficient optimization of queries that involve both [Ramakrishnan et al., 1998]. SRQL also extends SQL with some constructs for querying sequences.

SQL/LPP is a system that adds time-series extensions to SQL [Perng and Parker, 1999]. SQL/LPP models time-series as attributed queues (queues augmented with attributes that are used to hold aggregate values and are updated upon modifications to the queue). Each time-series is partitioned into segments that are stored in the database. The SQL/LPP optimizer uses pattern-length analysis to prune the search space and deduce properties of composite patterns from properties of the simple patterns.

SQL-TS [Sadri et al., 2001b; Sadri et al., 2001a] introduced simple and yet powerful extensions of SQL for finding patterns in sequences, along with techniques generalizing the Knuth-Morris-Pratt (KMP) algorithm [Knuth et al., 1977] to support the optimization of such queries. The ESL-TS system, discussed next, extends those constructs to work on data streams, rather than stored data, and uses a novel implementation and optimization architecture that exploits the native extensibility of ESL and the Stream Mill system.

The paper is organized as follows. In the next section, we introduce the time-series constructs of the ESL-TS language, which is, in Section 3, compared with languages proposed in the past for similar queries. In Section 4, we discuss the native extensibility mechanisms of ESL that we use in the implementation of ESL-TS, described in Section 5. In Section 6, we provide a short overview of the query optimization techniques used in such implementation.

2. The ESL-TS Language

Our Expressive Stream Language for Time Series (ESL-TS) supports simple SQL-like constructs to specify input data stream and search for complex sequential patterns on such streams.

Suppose we have a log of the web pages clicked by a user during a session as follows:

STREAM **Sessions(SessNo, ClickTime, PageNo, PageType)** ORDER BY **ClickTime;**

Here input pages are explicitly sequenced by **ClickTime** using the ORDER BY clause. A user entering the home page of a given site starts a new session that consists of a sequence of pages clicked; for each session number, **SessNo**, the log shows the sequence of pages visited—where a page is described by its timestamp, **ClickTime**, number, **PageNo** and type **PageType** (e.g., a content page, a product description page, or a page used to purchase the item).

The ideal scenario for advertisers is when users (i) see the advertisement page for some item in a content page, (ii) jump to the product-description page with details on the item and its price, and finally (iii) click the 'purchase this item' page. This advertisers' dream pattern can be expressed by the following ESL-TS query, where 'a', 'd', and 'p', respectively, denote an ad page, an item description page, and a purchase page:

EXAMPLE 6.1 *Using the* **FROM** *clause to define patterns*

```
SELECT Y.PageNo, Z.ClickTime
FROM Sessions
     PARTITION BY SessNO AS (X, Y, Z)
WHERE X.PageType='a'
     AND Y.PageType='d'
     AND Z.PageType='p'
```

Thus, ESL-TS is basically identical to SQL, but for the following additions to the FROM clause .

- A PARTITION BY clause specifies that data for the different sessions are processed separately (i.e., as if they arrived in separate data streams.) The semantics of this construct is basically the same as the PARTITION BY construct used in SQL:1999 windows [Zemke et al., 1999], which is also supported in the languages proposed by many data stream projects [Babcock et al., 2002]. In this example, the PARTITION BY clause specifies that data for each **SessNO** are processed as separate streams. The pattern AS **(X, Y, Z)** specifies that, for each **SessNO**, we seek a sequence of the three tuples **X, Y, Z** (with no intervening tuple allowed) that satisfy the conditions stated in the WHERE clause.

- The AS clause, which in SQL is mostly used to assign aliases to the table names, is here used to specify a sequence of tuple variables from the specified table. By (X, Y, Z) we mean three tuples that immediately follow each other.

  Tuple variables from this sequence can be used in the WHERE clause to specify the conditions and in the SELECT clause to specify the output.

In the SELECT clause, we return information from both the Y tuple and the Z tuple. This information is returned immediately, as soon as the pattern is recognized; thus it generates another stream that can be cascaded into another ESL-TS statement for processing.

2.1 Repeating Patterns and Aggregates

A key feature of ESL-TS is its ability to express recurring patterns by using a star operator. For instance, to determine the number of pages the user has visited before clicking a product description page (denoted by 'd') we simply write:

EXAMPLE 6.2 *Number of pages visited before the product description page is clicked, provided that this count is below 20*

```
SELECT SessNo, count(*A)
FROM Sessions
        PARTITION BY SessNO
        AS (*A, B)
WHERE A.PageType <> 'd'
        AND B.PageType = 'd'
        AND count(*A) < 20
```

Thus, *A identifies a **maximal** sequence of clicks to pages other than 'product' pages. Then, count(*A) tallies up those pages and, after checking that the count is less than 20, returns SessNo and the associated count to the user. The maximality of the star construct is important to avoid ambiguity and the possible explosion of matches.

ESL-TS supports a rich set of aggregates, as needed for time series analysis [Linoff and Berry, 1997]; aggregates supported includes rollups, running aggregates, moving-window aggregates, online aggregates, and user-defined aggregates inherited from the AXL/ATLaS system [Wang and Zaniolo, 2000]. Aggregates can only be applied to sequences defined by stars, and come in two very distinct flavors:

1 final aggregates applicable only after the star computation has completed, and

2 continuous aggregates that apply during the star computation.

For instance, **count(*A)** in Example 6.2 is a final aggregate: a sequence of pages is accepted, until a '**p**' page terminates the sequence. At that point, the condition **count(*A) < 20** is evaluated, and if satisfied the sequence is accepted and **SessNo** and **count(*A)** for that session are returned, otherwise the sequence is rejected. Example 6.3 illustrates the use of continuous aggregates—i.e., those that return the current value of the aggregates during the computation, as per online aggregates [Hellerstein et al., 1997]. It also illustrates how ESL-TS benefits from its ability of using standard SQL queries in combination with queries on sequences.

The previous queries were based on examples discussed in [Sadri et al., 2001b]. Let us now consider examples inspired by current data stream testbeds [Babu, 2002]. Assume we have an incoming stream **speed** sent by sensors placed on stations along the highway, which measures the average speed of cars once every minute. Also we have a database table **stations** that has descriptions of the stations, such as "Close to Exit 111".

```
STREAM speed(stationId, speed, speedTime) ORDER BY speedTime;
TABLE stations(stationId, location);
```

A good way of determining traffic condition is to find out **jam** locations along the highway. The **jam** condition is defined as a series of decreasing speeds, which leads to a more than 70% speed reduction, from some starting speed higher than 50 mph, within a time span of at most 6 minutes (we have assumed one measurement per minute). Example 6.3 uses continuous aggregates to detect such locations. The aggregate **ccount** is the online version of count, i.e., a continuous count that returns a new value for each new input. Thus, the condition **ccount(X)<=6** is satisfied for the first 6 elements in the sequence and, upon failing on the 7^{th} element, it brings the star sequence to completion. In general, continuous aggregates can be returned at various points during the computation of the sequence, as online aggregates do [Hellerstein et al., 1997]; thus, they can also be used in the conditions that determine whether the current tuple must be added to the star sequence being recognized.

The two different kinds of aggregates are syntactically distinguished by the fact that, the argument of a final aggregate is prefixed by the star; while there is no star in the argument of continuous aggregates. This query also uses the aggregate **LAST**; this a built-in aggregate that always returns the final value in the star sequence (thus, in Example 6.3 it is used to return the last value of **speed** in the sequence ***Y**.)

EXAMPLE 6.3 *Find out the* **jam** *locations along the highway*

```
SELECT A.location, LAST(Y).speedTime
FROM stations AS A, speed
     PARTITION BY stationId AS (X, *Y)
WHERE X.speed > 50
     AND Y.speed < Y.previous.speed
```

```
AND LAST(*Y).speed < 0.3*X.speed
AND ccount(Y) <= 6
AND X.stationId = A.stationId
```

Notice that, to retrieve the description of station locations, we use standard SQL to access database table **stations**. Also notice that we use the WHERE clause to specify conditions on both the values of attributes and those of aggregates. This is a simplification of traditional SQL (that would instead require HAVING for conditions on aggregates). This simplification is very beneficial for the users, and it has been adopted in more recent query languages such as XQuery [Boag et al., 2003].

The simplification is made possible by the lack of ambiguity associated with the sequential processing of patterns such as ***Y**. The processing is as follows: for each new tuple (i) the current values of attributes and continuous aggregates (i.e., those without the star, such as **ccount(Y)**) are evaluated and all the applicable conditions in the WHERE clause are tested, and (ii) if said conditions evaluate to true, then the computation of the star continues with the next tuple. Otherwise the evaluation of ***Y** completes and the final aggregates such as **count(*Y)** are computed and their values are used to test the applicable conditions in the where clause.

In general, therefore, we treat conditions on starred aggregates like conditions in the HAVING clause of standard SQL. Thus, for Example 6.2, the statement WHERE **count(*A) < 20** is treated like HAVING **count(A) < 20.**

Finally, the meaning of an aggregate such as **avg(*A)** would become undefined if ***A** were to contain zero or more elements, and therefore we require one or more elements in a the star construct. Therefore, ESL-TS wants to achieve both users' convenience and rigorous semantics; a formal logic-based semantics for the language constructs was presented in [Sadri, 2001].

As a more sophisticated example, say we want to find out the course of a traffic accident from the **speed** stream. We can compute a **diff** stream from **speed** stream with the following schema[2]:

```
STREAM diff(stationId, speed_diff, speedTime) ORDER BY speedTime;
```

A tuple in **diff** specifies speed difference between cars at the current station and cars at the next station. Under normal traffic, the difference remains under a rather low value. Whenever an accident happens, we will see a sudden increase of this difference; here, we define it as a more than 2 times increase within a time span of 6 minutes. After the accident is cleared, the difference drops back to a range within 10% of the stable condition. In this query, ***Y** is the pattern when the sudden speed difference jump happens, and once the increase stops the pattern ***Z** starts to match. ***Z** matching fails when the speed difference comes back to 10% of previous difference, or 60 minutes have elapsed, at which point the pattern is returned to the user.

EXAMPLE 6.4 *Detection of traffic accidents*

```
SELECT X.stationId, FIRST(Y).speedTime,
       LAST(Z).speedTime, LAST(Z).speed_diff
       FROM diff
       PARTITION BY stationId
       AS (X, *Y, *Z)
WHERE X.speed_diff <= 15
       AND Y.speed_diff > Y.previous.speed_diff
       AND LAST(*Y).speed_diff > 2*X.speed_diff
       AND ccount(Y) <= 6
       AND Z.speed_diff > 1.1*X.speed_diff
       AND ccount(Z) <= 60
```

2.2 Comparison with other Languages

The following example illustrates a search pattern on streams that has been previously proposed by other languages on stored sequences.

EXAMPLE 6.5 *Given a stream of events (time, name, type, magnitude) consisting of Earthquake and Volcano events, retrieve Volcano name and Earthquake name for volcano eruptions where the last earthquake (before the volcano) was greater than 7.0 in magnitude.*

```
SELECT V.name, LAST(E).name
FROM events AS (*E, V)
WHERE E.type ='Earthquake' AND V.type = 'Volcano'
       AND LAST(E).magnitude >= 7.0
```

This simple example is easily expressed in all the pattern languages proposed in the past [Ramakrishnan et al., 1998; Seshadri et al., 1994; Seshadri and Swami, 1995; Seshadri, 1998; Perng and Parker, 1999].

However, as illustrated in [Perng and Parker, 1999] most languages have problems with more complex patterns, such as the classical double-bottom queries, where given the table (name, price, time), for each stock find the W-curve (double-bottom). W-curve is a period of falling prices, followed by a period of rising prices, followed by another period of falling prices, followed by yet another period of rising prices. To make sure it is a "real" pattern we will enforce at least 5 prices in each period. In SQL/LPP+, this complex query is handled by the definition of patterns "uptrend" and "downtrend" followed by "doublebottom" as shown next.

EXAMPLE 6.6 *Double Bottom in SQL/LPP+*

```
CREATE PATTERN uptrend AS
       SEGMENT s OF quote WHICH_IS FIRST MAXIMAL, NON-OVERLAPPING
       ATTRIBUTE name AS first(s, 1).name
       ATTRIBUTE b_date AS first(s, 1).time
```

```
        ATTRIBUTE b_price AS first(s, 1).price
        ATTRIBUTE e_date AS last(s, 1).time
        ATTRIBUTE e_price AS last(s, 1).price
WHERE [ALL e IN s] (e.price >= prev(e, 1).price
        AND e.name = prev(e, 1).name)
        AND length(s) >= 5
CREATE PATTERN downtrend AS ...
/*this is similar to uptrend and omitted for lack of space*/
CREATE PATTERN double_bottom AS
        downtrend p1; uptrend p2; downtrend p3; uptrend p4 WHICH_IS ALL,
        NON-OVERLAPPING WHICH_IS NON-OVERLAPPING
        ATTRIBUTE name IS first(p1).name
        ATTRIBUTE b_date IS first(p1).time
        ATTRIBUTE b_price IS first(p1).price
        ATTRIBUTE e_date IS last(p4).time
        ATTRIBUTE e_price IS last(p4).price
WHERE p1.name = p2.name AND p2.name = p3.name
        AND p3.name = p4.name

SELECT db.b_date, db.b_price, db.e_date, db.e_price
FROM double_bottom
```

ESL-TS can express the same pattern in much fewer lines:

EXAMPLE 6.7 *Double Bottom in ESL-TS*

```
SELECT W.name, FIRST(W).time,
        FIRST(W).price, LAST(Z).time, LAST(Z).price FROM quote
        PARTITION BY name
        SEQUENCE BY date
        AS (*W, *X, *Y, *Z)
WHERE W.price <= W.previous.price AND count(*W) >= 5
        AND X.price >= X.previous.price AND count(*X) >= 5
        AND Y.price <= Y.previous.price AND count(*Y) >= 5
        AND Z.price >= Z.previous.price AND count(*Z) >= 5
```

3. ESL and User Defined Aggregates

ESL-TS is implemented as an extension of ESL that is an SQL-based data-stream language that achieves native extensibility and Turing completeness via user-defined aggregates (UDAs) defined in SQL itself rather than in an external procedural language. In fact, using nonblocking UDAs, ESL overcomes the expressive power loss from which all data stream languages suffer because of the exclusion of blocking operators. In [Law et al., 2004], it is shown that (i) all (and only) monotonic queries can be expressed by nonblocking computations, and (ii) using nonblocking UDAs, ESL can express all the computable monotonic functions. The practical benefits achieved by ESL's extraordinary level of theoretical power will become clear in the next section, where we will

show that the pattern-searching constructs of ESL-TS can be implemented by mapping them back into the UDAs of standard ESL.

User Defined Aggregates (UDAs) are important for decision support, stream queries, and other advanced database applications [Wang and Zaniolo, 2002; Babcock et al., 2002; Hellerstein et al., 1997]. ESL adopts from SQL-3 the idea of specifying a new UDA by an INITIALIZE, an ITERATE, and a TERMINATE computation; however, ESL lets users express these three computations by a single procedure written in SQL [Wang and Zaniolo, 2000]— rather than by three procedures coded in procedural languages as prescribed by SQL-3[3]. Example 6.8 defines an aggregate equivalent to the standard AVG aggregate in SQL. The second line in Example 6.8 declares a local table, **state**, where the sum and count of the values processed so far are kept. Furthermore, while in this particular example, **state** contains only one tuple, it is in fact a table that can be queried and updated using SQL statements and can contain any number of tuples. Thus, INITIALIZE inserts the value taken from the input stream and sets the count to 1. The ITERATE statement updates the tuple in **state** by adding the new input value to the sum and 1 to the count. The TERMINATE statement returns the ratio between the sum and the count as the final result of the computation by the INSERT INTO RETURN statement[4]. Thus, the TERMINATE statements are processed just after all the input tuples have been exhausted.

EXAMPLE 6.8 *Defining the standard aggregate average*

```
AGGREGATE myavg(Next Int) : Real
{   TABLE state(tsum Int, cnt Int);
    INITIALIZE : {
      INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
      UPDATE state
        SET tsum=tsum+Next, cnt=cnt+1;
    }
    TERMINATE : {
      INSERT INTO RETURN
        SELECT tsum/cnt FROM state;
    }
}
```

Observe that the SQL statements in the INITIALIZE, ITERATE, and TERMINATE blocks play the same role as the external functions in SQL-3 aggregates. But here, we have assembled the three functions under one procedure, thus supporting the declaration of their shared tables (the **state** table in this example).

deallocated just after the TERMINATE statement is completed. This approach to aggregate definition is very general. For instance, say that we want to support tumbling windows of 200 tuples [Carney et al., 2002]. Then we can write the UDA of Example 6.9, where the RETURN statements appear in ITERATE instead of

TERMINATE. The UDA **tumble_avg**, so obtained, takes a stream of values as input and returns a stream of values as output (one every 200 tuples). While each execution of the RETURN statement produces here only one tuple, in general, a UDA can produce (a stream of) several tuples. Thus UDAs operate as general stream transformers. Observe that the UDA in Example 6.8 is blocking, while that of Example 6.9 is nonblocking. Thus, nonblocking UDAs are easily expressed in ESL, and clearly identified by the fact that *their* TERMINATE *clauses are either empty or absent.* The typical default semantics for SQL aggregates is that the data are first sorted according to the GROUP-BY attributes: thus the very first operation in the computation is a blocking operation. Instead, ESL uses a (nonblocking) hash-based implementation for the GROUP-BY calls of the UDAs. This default operational semantics leads to a stream oriented execution, whereby the input stream is pipelined through the operations specified in the INITIALIZE and ITERATE clauses: the only blocking operations (if any) are those specified in TERMINATE, and these only take place at the end of the computation.

EXAMPLE 6.9 *Average on a Tumbling Window of 200 Tuples*

```
AGGREGATE tumble_avg(Next Int) : Real
{   TABLE state(tsum Int, cnt Int);
    INITIALIZE : {
      INSERT INTO state VALUES (Next, 1);
    }
    ITERATE : {
      UPDATE state
        SET tsum=tsum+Next, cnt=cnt+1;
      INSERT INTO RETURN
        SELECT tsum/cnt FROM state
        WHERE cnt % 200 = 0;
      UPDATE state SET tsum=0, cnt=0;
        WHERE cnt % 200 = 0
    }
    TERMINATE: { }
}
```

ESL supports standard SQL, where the UDAs (defined using SQL) are called in the same way as any other built-in aggregate. As discussed above, both blocking and non-blocking UDAs can be used on database tables, however only non-blocking UDAs can be used on streams, as in the next example. For instance, given an incoming stream which contains bidding data for an online-auction web site:

```
STREAM bid(auction_id, price, bidder_id, bid_time) ORDER BY bid_time;
```

Example 6.10 continuously computes the number of unique bidders for auction with ID 1024 within a time-based sliding window of 30 minutes by applying a non-blocking UDA bidder_wcount on stream **bid** (which will be define in the

next example). The first two lines in Example 6.10 illustrate stream declaration in ESL. The next two lines of Example 6.10 filter the tuples from the stream **bid** using the condition **auction_id=1024**; the tuples that survive the filter are then pipelined to the UDA **bidder_wcount**.

EXAMPLE 6.10 *UDAs and Streams in ESL*

```
STREAM bid(auction_id, price, bidder_id, bid_time)
       ORDER BY bid_time;
SELECT auction_id, bidder_wcount(bidder_id, bid_time, 30)
       FROM bid WHERE auction_id=1024;
```

In Example 6.11, we define an aggregate **bidder_wcount** that continuously returns the count of unique bidders within a sliding window of certain number of minutes, with the window size passed in as a formal parameter. Observe that the formal parameters of the UDA function are treated as constants in the SQL statements. Thus, the INSERT statement in INITIALIZE put into the table **bidders** with the constant **bidder_id** and **bid_time**. In ITERATE, we first add the bidder into the table bidders, if it is a new bidder. Then, if it is an existing bidder, we update the last seen timestamp for that bidder. Next, we delete all bidders last seen before the sliding window starts. Finally, the RETURN statement in ITERATE returns the current count of unique bidders within the sliding window.

EXAMPLE 6.11 *Continuous count of unique bidders within a sliding window of certain number of minutes*

```
AGGREGATE bidder_wcount(bidder_id, bid_time, num_min):(bcount)
{   TABLE bidders(b_id, btime);
    INITIALIZE :{
       INSERT INTO bidders VALUES(bidder_id, bid_time);
    }
    ITERATE:{
       INSERT INTO bidders VALUES(bidder_id, bid_time)
            WHERE bidder_id NOT IN (SELECT bId FROM bidders);
       UPDATE bidders SET btime = bid_time
            WHERE bidder_id = b_id;
       DELETE FROM bidders
            WHERE bid_time > (btime + num_min minutes);
       INSERT INTO RETURN
          SELECT count(b_id) FROM bidders
    }
    TERMINATE : {}
}
```

Observe that, this UDA has an empty TERMINATE, thus it is non-blocking and can be used on streams. It maintains a buffer with minimum number of tuples within the sliding window, those that are needed to ensure all the unique bidders are counted.

The power and native extensibility produced by UDAs makes them very useful in a variety of application areas, particularly those, such as data mining, that are too difficult for current O-R DBMSs [Han et al., 1996; Meo et al., 1996; Imielinski and Virmani, 1999; Sarawagi et al., 1998]. The ability of UDAs to support complex data mining algorithms was discussed in [Wang and Zaniolo, 2003], where they were used in conjunction with table functions and in-memory tables to achieve performance comparable to that of procedural algorithms under the cache mining approach. For instance in [Wang and Zaniolo, 2003], a scalable decision-tree classifier was expressed in less than 20 statements. In the next section we describe how UDAs are used to implement SQL-TS.

4. ESL-TS Implementation

The ESL-TS query of Example 6.2 can be recast into the FSM of Figure 6.1, and implemented using the UDA of Example 6.12.

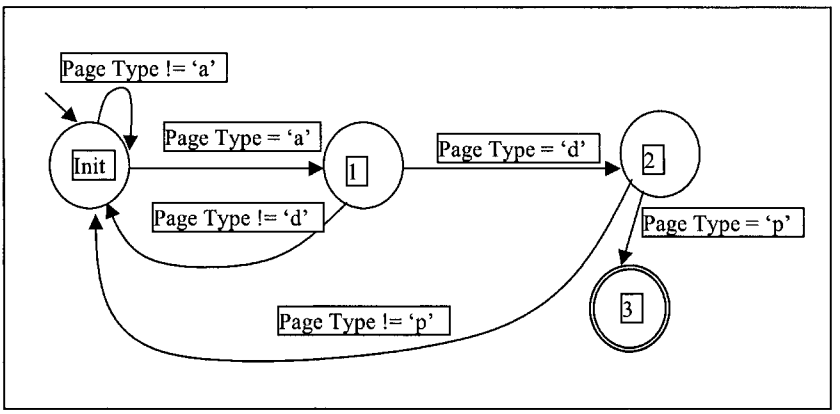


Figure 6.1. Finite State Machine for Sample Query.

We can walk through the code of Example 6.12, as follows:

Lines 2 and **3:** we define local table **CurrentState** that is used to maintain the current state of the FSM, and the table **Memo** that holds the last input tuple.

Line 4: we initialize these tables as the first operation in INITIALIZE.

Line 5 and **6**: we check the first tuple to see if it is 'a'. If, and only if, this the case, the state is advanced to 1 and tuple values updated for state 1,

Line 7: we check if we have the correct input for transitioning to the next state and, in case of failure, we reset the state back to 0— this corresponds to

EXAMPLE 6.12 *Implementation of Example 6.2*

```
1: Aggregate find_pattern(PageNoIn int, ClickTimeIn char(16),
                PageTypeIn char(1)): (PageNo int, ClickTime char(16))
2: {  TABLE CurrentState(curState int);
3:    TABLE Memo(PageNo int, ClickTime char(16), State int);
     INITIALIZE: {
4:     INSERT INTO CurrentState VALUES(0);
       INSERT INTO Memo VALUES ((0, '', 1), (0, '', 2), (0, '', 3));
5:     UPDATE Memo SET PageNo = PageNoIn, ClickTime = ClickTime
         WHERE PageTypeIn = 'a' AND State = 1;
6:     UPDATE CurrentState SET curState = 1 WHERE sqlcode = 0 }
     ITERATE: {
7:     UPDATE CurrentState set curState = 0
         WHERE (curState = 0 AND pageType <> 'a')
         OR (curState = 1 AND pageType <> 'd')
         OR (curState = 2 AND pageType <> 'p');
8:     UPDATE CurrentState SET curState = curState + 1
         WHERE sqlcode > 0 AND ((curState = 0 and pageType = 'a')
         OR (curState = 1 and pageType = 'd')
         OR (curState = 2 and pageType = 'p'));
9:     UPDATE Memo SET PageNo = PageNoIn, ClickTime = ClickTimeIn
         WHERE Memo.State = (SELECT curState FROM CurrentState)
         and sqlcode =0;
10:    INSERT INTO return SELECT Y.PageNo, Z. ClickTime
         FROM CurrentState AS C, Memo AS X, Memo AS Y, Memo
         WHERE C.curState = 3 and Y.st = 2 AND Z.st = 3;
11:    UPDATE CurrentState SET curState=0 WHERE sqlcode = 0}
  }
```

the "Init" state in Figure 6.1. We are now in the ITERATE clause of the UDA, and this clause will be is executed for each subsequent input tuple.

Line 8: If line 7 did not execute (**sqlcode**> 0, indicates the failure of the last statement), then the transition conditions hold, and we advance to the next state.

Line 9: once we transitioned into the next state (**sqlcode** =0 indicates that the last statement succeeded), we need to update the current tuple value for that state.

Line 10: if we are now in the accepting state (State 3), we simply return the tuple values.

Line 11 : once the results are returned, we must reset the FSM to its "Init" state (State 0).

This is realized as follows:

1 We use a special UDA (the same for all ESL-TS queries), called the **buffer_manager**. This UDA passes to **find_pattern** set of tuples, as follows.

The last state of the **find_pattern** UDA is checked, and if this is 0 (denoting backtracking) then teh **buffer_manager** calls **find_pattern** with the "required" old tuples. Otherwise, the **buffer_manager** calls the **find_pattern** UDA on the current input tuple (which it also stores in **buffer** since it might needed later, after backtracking.

2 The **buffer_manager** first sends some old tuples to **find_pattern**, and then take more tuples from the input and give them to **find_pattern** with the expectation that this will resume the computation from the state in which it had left it. Thus **buffer_manager** is reentrant, and remembers the state in which it was last executed[5].

The implementation of the full ESL-TS also supports the star construct, and aggregates. The '*' construct translates to a self-loop in the FSM, and the **find_pattern** UDA was easily extended to handle such self-loops. Finally aggregates are supported, by storing an additional column in the **Memo** for each aggregate in the query (the **.previous** is implemented in a similar fashion). Optimization is discussed in the next section.

The general approach for implementing different FSMs is the same across all different FSMs, therefore we can automate this translation. The resulting UDA and ESL query can be used on both static tables and data streams. Furthermore, native SQL optimizations can be applied to both. Figure 6.1 below illustrates the corresponding FSM.

5. Optimization

The query optimization problems for continuous queries can be very different from the relational-algebra driven approach of traditional databases. Finite state automata based computation models are often used for streaming XML data [Diao and Franklin, 2003], while the generalization of the Knuth, Morris and Pratt (KMP) text search algorithms [Knuth et al., 1977] was proven very effective to mininimize execution cost of SQL-TS [Sadri et al., 2001b]. In ESL-TS, we are extending the KMP algorithm, to optimize memory utilization, and the execution of concurrent queries.

The KMP algorithm provides a solution of proven optimality [Wright et al., 1998] for queries such as that of Example 6.1, which searches for the sequence of three particular constant values. The algorithm minimizes execution by predicting failures and successes in the next search from those in the previous search. The algorithm takes a sequence pattern of length m, $P = p_1 \dots p_m$, and a text sequence of length n, $T = t_1 \dots t_n$, and finds all occurrences of P in T. Using an example from [Knuth et al., 1977], let $abcabcacab$ be our search pattern, and $babcbabcabcaabcabcabcacabc$ be our text sequence. The algorithm starts from the left and compares successive characters until the first mismatch occurs. At each step, the i^{th} element in the text is compared with the

j^{th} element in the pattern (i.e., t_i is compared with p_j). We keep increasing i and j until a mismatch occurs.

j, i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
p_j	a	b	c	a	b	c	a	c	a	b							
				⇑													

For the example at hand, the arrow denotes the point where the first mismatch occurs. At this point, a naive algorithm would reset j to 1 and i to 2, and restart the search by comparing p_1 to t_2, and then proceed with the next input character. But instead, the KMP algorithm avoids backtracking by using the knowledge acquired from the fact that the first three characters in the text have been successfully matched with those in the pattern. Indeed, since $p_1 \neq p_2$, $p_1 \neq p_3$, and $p_1p_2p_3 = t_1t_2t_3$, we can conclude that t_2 and t_3 can't be equal to p_1, and we can thus jump to t_4. Then, the KMP algorithm resumes by comparing p_1 with t_4; since the comparison fails, we increment i and compare t_5 with p_1:

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
t_i	a	b	c	b	a	b	c	a	b	c	a	a	b	c	a	b	c
j					1	2	3	4	5	6	7	8	9	10			
p_j					a	b	c	a	b	c	a	c	a	b			
												⇑					

Now, we have the mismatch when $j = 8$ and $i = 12$. Here we know that $p_1 \ldots p_4 = p_4 \ldots p_7$ and $p_4 \ldots p_7 = t_8 \ldots t_{11}$, $p_1 \neq p_2$, and $p_1 \neq p_3$; thus, we conclude that we can move p_j four characters to the right, and resume by comparing p_5 to t_{12}.

In general, let us assume that the search has failed on position j of the pattern and succeeded before that; it is then possible to compute at compile-time the following two functions (delivering nonnegative integers):

- $shift(j)$: this determines how far the pattern should be advanced in the input, and
- $next(j)$: this determines from which element in the pattern the checking of conditions should be resumed after the shift.

The result is of this optimization is that less backtracking is required, when $shift(j) > 0$, and fewer elements of the pattern need to be checked, when $next(j) > 0$. Thus with KMP, the complexity of searching for the first occurrence of a pattern of length m on a text of length n is is reduce to $O(m + n)$—whereas it would be $O(m \times n)$ without this optimization [Knuth et al., 1977]. The Optimized Pattern Search (OPS) algorithm, proposed in [Sadri et al., 2001b]

represents a significant generalization of the KMP algorithm in as far as we support:

- general predicates, besides the equality predicates of KMP, and
- SQL-TS patterns defined using the star and aggregates, are also fully supported.

In terms of execution speed, the OPS algorithm delivers orders of magnitude improvements over the naive search algorithm, and it therefore is being used in ESL-TS. But, in addition to this, we are using OPS to minimize memory usage. For, instance, let us return to our previous example, where we observed that the first search failed at t_4. Before that, we had succeeded at t_2, and then t_3; now, those successes are sufficient to assure that we could first discard t_2 and then t_3. Likewise, by the time we reach t_{12}, all the positions before that in memory can be discarded. This observation is of great practical value in the case of the star patterns, since a pattern *X can match an input stream of considerable length. Since ESL-TS only allows users to retrieve the start, the end, and aggregates functions on *X, we can drop from memory all the X values as soon as they are scanned.

A final topic of current research in ESL-TS optimization is the interaction between multiple concurrent queries. Currently, the OPS algorithm is based on the logical implications between conditions in different phases of the same query: we are now investigating how to extend our optimization to exploit implications across conditions of different queries.

6. Conclusion

Time series queries occur frequently in data stream applications, but they are not supported well by the SQL-based continuous query languages proposed by most current data stream management systems. In this paper, we have introduced ESL-TS that can express powerful time series queries by simple extensions of SQL. We have also shown that these extensions can be implemented on top of the basic ESL language—thus demonstrating the power of ESL on data stream applications and the the benefits of its native extensibility mechanisms based on UDAs. We also discussed optimization techniques for ESL-TS, and showed that they can be used to minimize execution time and memory for intra-query and inter-query optimization.

Notes

1. These include temporal queries such as until and coalesce, and queries expressible using monotonic aggregation [Law et al., 2004]

2. The computation can be easily expressed in ESL-TS

3. Although UDAs have been left out of SQL:1999 specifications, they were part of early SQL-3 proposals, and supported by some commercial DBMS.

4. To conform to SQL syntax, RETURN is treated as a virtual table; however, it is not a stored table and cannot be used in any other role.

5. "last state" means the value in the CurrentState table of the UDA at the end of the last call to the UDA. If the CurrentState table is outside of the find_pattern UDA then the "last state" can be retrieved from it.

Acknowledgments

The authors wish to thank Reza Sadri for SQL-TS, and Haixun Wang for his contribution to ESL.

References

A. Arasu, S. Babu, and J. Widom. An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University, 2002.

B. Babcock, S. Babu, M. Datar, R. Motawani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

Shivnath Babu. Stream query repository. Technical report, CS Department, Stanford University, http://www-db.stanford.edu/stream/sqr/, 2002.

D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, 1999.

S. Boag, D. Chamberlin, M. F. Fernandez, D. Florescu, J. Robie, J. Simeon, and M. Stefanescu (eds.). Xquery 1.0: An xml query language–working draft 22 august 2003. Working Draft 22 August 2003, W3C, http://www.w3.org/tr/xquery/, 2003.

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 *VLDB*, Hong Kong, China, 2002.

S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *VLDB*, 2002.

J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, pages 379–390, May 2000.

Yanlei Diao and Michael J. Franklin. Query processing for high-volume xml message brokering. In *VLDB 2003*, pages 261–272, 2003.

Lukasz Golab and M. Tamer Özsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

J. Han, Y. Fu, W. Wang, K. Koperski, and O. R. Zaiane. DMQL: A data mining query language for relational databases. In *Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 27–33, Montreal, Canada, June 1996.

J. M. Hellerstein, P. J. Hass, and H. J. Wang. Online aggregation. In *SIGMOD*, 1997.

T. Imielinski and A. Virmani. MSQL: a query language for database mining. *Data Mining and Knowledge Discovery*, 3:373–408, 1999.

Informix. Informix: Datablade developers kid infoshelf. http://www.informix.co.za/answers/english/docs/dbdk/infoshelf, 1998.

H. Jagadish, I. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *PODS*, pages 113–124, 1995.

D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM Journal of Computing*, 6(2):323–350, June 1977.

Y-N Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams In *VLDB*, 2004.

L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE TKDE*, 11(4):583–590, August 1999.

G. Linoff M. J. A. Berry. *Data Mining Techniques: For Marketing, Sales, and Customer Support.* John Wiley, 1997.

Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD*, pages 49–61, 2002.

R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *VLDB*, pages 122–133, Bombay, India, 1996.

C. Perng and D. Parker. SQL/LPP: A Time Series Extension of SQL Based on Limited Patience Patterns In *DEXA*, 1999.

R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. Beyer, and M. Krishnaprasad. Srql: Sorted relational query language, 1998.

Reza Sadri. *Optimization of Sequence Queries in Database Systems*. PhD thesis, University of California, Los Angeles, 2001.

Reza Sadri, Carlo Zaniolo, and Amir M. Zarkesh andJafar Adibi. A sequential pattern query language for supporting instant data minining for e-services. In *VLDB*, pages 653–656, 2001.

Reza Sadri, Carlo Zaniolo, Amir Zarkesh, and Jafar Adibi. Optimization of sequence queries in database systems. In *PODS*, Santa Barbara, CA, May 2001.

S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *SIGMOD*, 1998.

P. Seshadri. Predator: A resource for database research. *SIGMOD Record*, 27(1):16–20, 1998.

Praveen Seshadri, Miron Livny, and Raghu Ramakrishnan. Sequence query processing. In Richard T. Snodgrass and Marianne Winslett, editors, *Proceedings of the 1994 ACM SIGMOD International Conference on Management of Data*, pages 430–441. ACM Press, 1994.

Praveen Seshadri and Arun N. Swami. Generalized partial indexes. In *Proceedings of Eleventh International Conference on Data Engineering 1995*, pages 420–427. IEEE Computer Society, 1995.

M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *VLDB*, 1996.

D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *SIGMOD*, pages 321–330, 6 1992.

Haixun Wang and Carlo Zaniolo. Using SQL to build new aggregates and extenders for object-relational systems. In *VLDB*, 2000.

Haixun Wang and Carlo Zaniolo. Extending sql for decision support applications. In *Proceedings of the 4th Intl. Workshop on Design and Management of Data Warehouses (DMDW)*, pages 1–2, 2002.

Haixun Wang and Carlo Zaniolo. ATLaS: A native extension of sql for data mining. In *SDM*, San Francisco, CA, 5 2003.

C. A. Wright, L. Cumberland, and Y. Feng. A performance comparison between five string pattern matching algorithms. Technical Report, Dec. 1998. http://ocean.st.usm.edu/~cawright/pattern_matching.html.

Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle. Proposal for OLAP functions. In *ISO/IEC JTC1/SC32 WG3:YGJ-nnn, ANSI NCITS H2-99-155*, 1999.

Chapter 7

MANAGING DISTRIBUTED GEOGRAPHICAL DATA STREAMS WITH THE GIDB PORTAL SYSTEM

John T. Sample,[1] Frank P. McCreedy,[1] and Michael Thomas[2]

[1]*Naval Research Laboratory, Stennis Space Center, MS 39529, USA*

[2]*National Guard Bureau - Counter Drug, Atlanta, GA, USA*

Abstract The Naval Research Laboratory (NRL) has developed a portal system, called the Geospatial Information Database (GIDB®) which links together several hundred geographic information databases. The GIDB portal enables users to access many distributed data sources with a single protocol and from a single source. This chapter will highlight the current functionality of the GIDB Portal System and give specific applications to military and homeland security uses.

Keywords: geographical information system, web map service, client, server.

1. Introduction

Recently, numerous geographical data collections have been made available on the Internet. These data collections vary greatly. Some are large collections of imagery, maps and charts. Others are small stores of highly specialized data types, for example, fire hydrant locations in Phoenix, Arizona. In addition, highly volatile data sources, such as weather forecasts, are available. These many data sources offer access to geographical information at an unprecedented level. However, the greater amounts, types and sources of data produce greater complexity in managing the data.

To provide access to the many different types of geographic data sources in a simple and straightforward manner, the Naval Research Laboratory (NRL) has developed the Geospatial Information Database Portal System (GIDB)[1].

The GIDB is unique in its ability to link many different data sources and make them available through a single source and with a uniform protocol.

2. Geographic Data Servers

To better understand the requirements, challenges, and possibilities involved with integrating many different geographic data servers, detailed descriptions of the different kinds of geographic data, servers, and communication protocols are needed.

Figure 7.1. Vector Features for Nations in North America.

2.1 Types of Geographic Data

In general terms there are two main types of geographic data, vector and raster. Vector geographic features use geometrical primitives such as points, lines, curves, and polygons to represent map features such as roads, rivers, nations. Raster geographic data types are generally structures that consist of rectangular arrays of pixels or points of with given values. This can include scanned maps and charts, aerial, satellite, and sonar imagery and terrain and bathymetric grids.

Figures 7.1, 7.2 and 7.3 show examples of vector and raster map data. Figure 7.1 shows a map with vector features representing countries in North America. Figure 7.2 shows shaded relief (elevation) in the form of an image for the same area. Figure 7.3 shows the two types of features together in the same view.

These are the most basic categories of geographic data types and typical of the information provided by most geographic data servers. Three dimensional

Figure 7.2. Shaded Relief for North America.

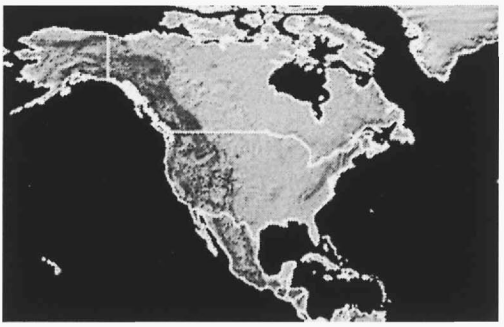


Figure 7.3. Combined View From Figures 7.1 and 7.2.

and multi-media types are also available, but not directly considered in this chapter.

2.2 Types of Geographic Data Servers

As previously mentioned, geographic data servers can be quite varied. Some are quite complex and built on fully functional database management systems (DBMS). Others are simply transport mechanisms for sensor data or other observations.

The most basic types of geographic data servers are often as simple as a web page or FTP (File Transport Protocol) site with geographic data files available. These files can be static or dynamic. Many organizations employ sites such as this. They are simple and inexpensive to setup and straightforward to use. For example, the United States Geological Survey (USGS) and the United States Department of Transportation each have large datasets of public geographic data available for download from their websites. These types of sites usually lack the ability to query based on area of interest (AOI), time or theme. However, they are useful data sources, especially when used with a portal system such as the GIDB, which provides its own query functionality for AOI, time, and theme.

In addition to simple web and FTP sites, there exist special purpose servers that produce small data products which do not require sophisticated interfaces. An example of this type of system could be a server which sends out auto-generated emails with tropical storm warnings. Also, consider a system that makes available the locations of commercial aircraft in route. These types of systems can have limited impact on their own. However, when integrated into a more complete system, they become much more useful.

The next broad category of geographic servers consists of more comprehensive software systems that can provide a user with a complete, though often specialized, map view. These are usually expensive and complicated server systems, which include a DBMS, fully functional GIS, and some type of map renderer. Most of these systems require users to use a specific client software package to access the server. Several vendors currently provide these types of software; examples are ESRI's ArcIMS and AutoDesk's MapGuide.

These systems provide query functionality for a variety of fields including AOI, time and theme. They display data in different layers on a interactive map view. While very powerful, they are almost always restricted to accessing data that resides under the control of the server. Thus, the system is responsible for data maintenance, backups, updates, etc.

There are other types of geographic data servers; however, the provided examples are sufficient to show the variety of systems currently in use and

indicate the challenges involved in producing a system which can integrate all these types of servers into a comprehensive solution.

2.3 Transport Mechanisms

Transport mechanisms refer to the network communication protocols used to communicate data across the network. Within the current network structure of the Internet there are several levels of abstraction from the hardware level, on which the various protocols reside. For a complete discussion of network/Internet protocols see [Tanenbaum, 2003]. The basic data protocol of the internet is TCP/IP (Transport Control Protocol/Internet Protocol). This is a general purpose protocol for the transport of generally unformatted data. On top of TCP/IP a large variety of other special purpose protocols are built. This is called the "application" layer. Examples of these are HTTP (Hypertext Transport Protocol), FTP, SMTP (Simple Mail Transport Protocol). At this level of abstraction, the data takes on slightly more structure, and requests, responses, and sessions can be managed.

HTTP is by far the preferred method for data transfer in this environment. Virtually all web pages and many web based applications are provided to clients via HTTP. It was designed to handle the typical request/response paradigm of webpage browsing. In some cases, HTTP is used despite its not being the most appropriate protocol. This is because many organizations' security restrictions, firewalls and proxy servers exclude virtually any network traffic that is not HTTP based. Under these restrictions, many systems are forced to use HTTP.

HTTP is a "stateless" protocol; therefore, each interaction is independent from those preceding it. Contrast this to FTP, in which users "log in" for a session, conduct several transactions and then "log out." Also, HTTP does not directly handle delayed responses. Contrast this to SMTP, in which messages can be passed at will between various parties. For example, consider the case in which the response to a request requires several hours (or days) to formulate. In that case the HTTP request/response paradigm is not sufficient. Also, consider that the response might never be truly complete. For example, one might query a server which contains up to date locations of oil tankers. A client system could request to be notified any time updated positions become available; thus the server would need to send out updates frequently. In this case, the server initiates communication with the client. This is a reversal of the typical server-client relationship, and would require a different communication protocol.

Beyond the level of standard communication protocols, there are many more details to be resolved to communicate effectively with a geographic data servers. Authentication details, query languages, and data formats are just examples of particulars which must be agreed upon between those providing data and those requesting data. Unfortunately, most current comprehensive geographic data

Table 7.1. Selected OGC Geographic Standards.

Standard Name	*Description*
Web Map Service (WMS)	Provides four protocols in support of the creation and display of registered and superimposed map-like views of information that come simultaneously from multiple sources that are both remote and heterogeneous.
Web Feature Service (WFS)	The purpose of the Web Feature Server Interface Specification (WFS) is to describe data manipulation operations on OpenGIS® Simple Features (feature instances) such that servers and clients can "communicate" at the feature level.
Geography Markup Language (GML)	The Geography Markup Language (GML) is an XML encoding for the transport and storage of geographic information, including both the geometry and properties of geographic features.
Catalog Interface (CAT)	Defines a common interface that enables diverse but conformant applications to perform discovery, browse and query operations against distributed and potentially heterogeneous catalog servers.
Web Coverage Service (WCS)	Extends the Web Map Service (WMS) interface to allow access to geospatial "coverages" that represent values or properties of geographic locations, rather than WMS generated maps (pictures).

servers provide data in proprietary formats through proprietary protocols to proprietary clients. This can make integrating multiple types of geographic data servers very challenging, if not impossible.

2.4 Geographic Data Standards

Given the variety of the data types, servers and communication protocols, one can see why integrating all these geographical data sources is such an imposing problem. One solution to this problem is the adoption of various standards for geographical data. The OpenGIS Consortium (OGC) has proposed a number of standards for communicating geographic data and metadata. These standards are available from http://www.opengis.org. Table 7.1 lists and describes the most significant OGC standards [OpenGIS].

These standards are generally sufficient to handle the problem of geographic data transfer. However, many organizations lack the funding or personnel to replace or upgrade their existing data servers to comply with these standards. Thus, the variety of proprietary access protocols is likely to persist well into the future.

2.5 Geographic Data Streams

While many geographic data servers can be viewed as traditional database data sources, there are instances in which it is better to model the data source as a stream. There are numerous examples of geographic data sources which behave more like streams than databases. Consider a server which monitors a series of weather stations and outputs weather observations continually. The number of available observations and the frequency of updates vary, but the updates continue indefinitely. Thus, like a stream, this data source is potentially unbounded in size. One could only attempt to obtain a snapshot of the data at a specific moment in time, and that snapshot becomes out-of-date quickly. Furthermore, the size of the snapshot of data might be prohibitively large. The best way to handle this data is to treat it as a stream of information, and sample a practically feasible amount of data from the stream for display.

In the examples section of this chapter, two scenarios will presented in which data sources that behave like streams are integrated into the GIDB framework for traditional data sources.

3. The Geospatial Information Database Portal System

Military and homeland security organizations face a common need for mapping products that are both mission relevant and current. These datasets can include fairly static data types like road maps and aerial photography, but they also have a need for dynamic data types like weather maps. In addition, any system to be utilized by a broad user community must be simple to use and cost effective. The GIDB Portal System is unique in meeting these needs.

Within this user community, required geographic datasets are often compiled, recorded and delivered by non-networked means on a periodic basis. Once received, they are stored and served locally. This process produces datasets that are static and revised infrequently. Furthermore, the costs of warehousing very large datasets locally are prohibitive for many smaller organizations, such as local sheriffs and police departments.

The obvious solution to this problem is to provide users the ability to connect to the datasets they need over a dynamic network connection. Thus they have current datasets, which require no local maintenance. However, this is not a trivial task and includes many significant challenges. This section will describe how the GIDB Portal system solves the technical challenges involved with designing and deploying a system that meets these needs.

3.1 GIDB Data Sources

No single dataset contains all of the geospatial information required for the variety of military and homeland security operations supported by the GIDB

Portal System. Some operations can require data from as many as twenty different sources. These include geospatial data servers from federal, state and local government organizations and others.

Thus, if a typical user wants to combine custom-generated maps with real-time weather data, digitized raster graphics, political boundary and road data and census data, that user would have to spend a considerable amount of time finding each of those data types individually and integrating them together. Due to the diversity of the data suppliers, one encounters many different types of data servers among the available sources. After locating the different data sources, the user must gain access to them via a variety of proprietary access methods. Finally, the data types must be converted into a format that is compatible with the user's GIS of choice. This is a formidable task and hardly practical for everyday use. In emergency situations where speed is a requirement, this is infeasible.

Therefore, the primary technical requirement for the GIDB Portal System is to connect many different data sources via many different access protocols. This also necessitates that the systems be linked through a single location. Also, this linking should be truly dynamic. The portal should maintain an active directory of data sources and "know" when sources are online and offline. Furthermore, it should have fall back mechanisms for cases in which certain data sources are temporarily unavailable. Figure 7.4 shows the relatively simple concept: that multiple distributed data sources can be integrated remotely to appear as a single source. In this figure the listed data sources are just examples of the many types of data sources for which interfaces have been developed.

3.2 GIDB Internals

The core of the GIDB Portal System is the Apache/Tomcat server framework. This configuration was chosen to take advantage of a robust HTTP web server with a Java Servlet engine. The web server component, Apache, manages network connections with a variety of client software packages. The Java Servlet engine, Tomcat, allows the rich Java library of database, networking, GIS and other components to be used.

Linking data sources within this framework is accomplished by first defining a standard set of data types and query types that encapsulate the functions of a generalized geospatial data server. This standard set in the GIDB is fairly broad. It contains data types for vector and raster map data, meta-data, and extensible user defined data types. Queries based on scale and AOI are provided in addition to methods for acquiring metadata and browsing available data types. This framework constitutes the first level of abstraction in connecting many distributed sources and making them appear as one source. In practical form,

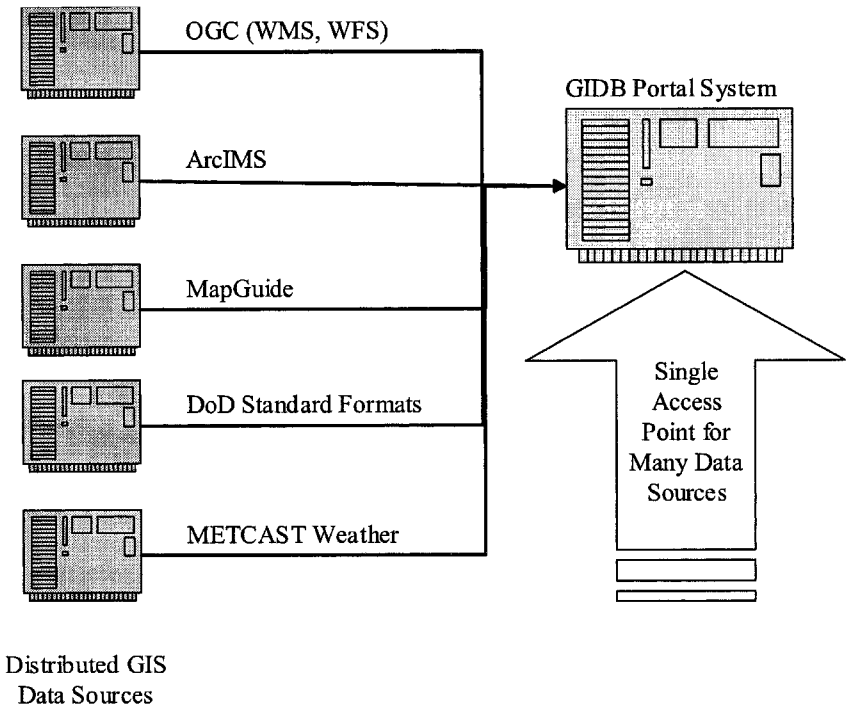


Figure 7.4. GIDB Data Source Architecture.

the standard set of data types and query methods are defined in a collection of Java classes and interfaces.

This collection of Java classes and interfaces encapsulates all that is required to link a networked data source via a proprietary or custom access protocol to the GIDB. In order to link the external data servers, the data types provided by the server are converted to the GIDB standard format, and the query methods required by the GIDB interface are implemented to function with the query methods provided by the external source [Chung et al., 2001]. The completed interface between the GIDB and the external interface is called a "driver." The driver also maintains connections to the various data sources. Most of the connected data sources use HTTP based network communication.

The key feature of the GIDB, one that distinguishes it from other solutions, is that all the code and configuration needed to perform the linkage are located within the GIDB portal system installation. Thus, the provider of the external data source is not required to reconfigure their system in any way. This is the most significant reason for the rapid growth of the number of data servers available in the GIDB. The researchers and developers of the GIDB can configure and catalog data sources around the world without directly interacting with data providers.

Currently a variety of vendor-specific access protocols and data types are supported in the GIDB. These include ESRI ArcIMS, ESRI shape, AutoDesk, Map Objects, METCAST, TerraServer, Census Tiger, and Web Mapping Service. Also, U.S. Department of Defense standard mapping products in VPF (vector product format), CADRG (compressed ARC digitized raster graphic) and CIB (controlled image base) are supported and provided via the GIDB to authorized users.

Given the driver architecture of the GIDB portal system, we can further elaborate on the graphic in Figure 7.4. Consider Figure 7.5, this graphic shows the relationship between each data source to the GIDB. Each data source is connected to the GIDB through a driver component, which converts the native queries and formats of the driver to those used by the GIDB.

3.3 GIDB Access Methods

The previous sections describe how different types of data spread out over multiple data sources can be connected and made available through a single source. This section will detail how this single source distributes data to users through a variety of methods. The GIDB standard interface for linking data sources requires that all the data sources be transformed into a common view within the GIDB. Therefore, from a user perspective, the many different data sources do not appear as many sources with many different access protocols;

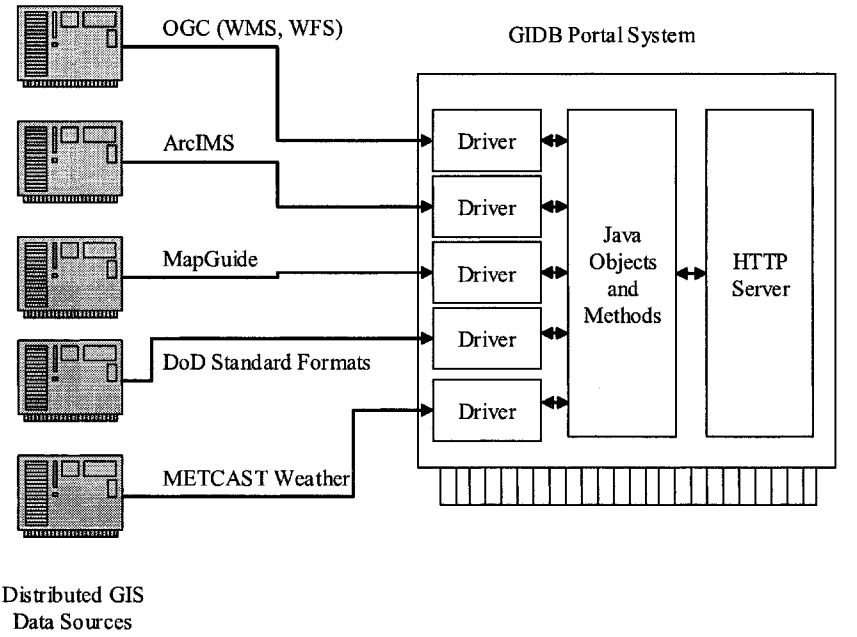


Figure 7.5. Detailed View of GIDB Data Source Architecture.

Table 7.2. GIDB Client Software Packages.

Client Software Package	*Benefits*
Web Browser Based "Thin" Client	1. Requires No Installation 2. Simple to Use 3. ISO 19115 Theme Based Data Access 4. Includes User Registration System
Advanced Client (Full Stand Alone Java Application)	1. Advanced Map Creation Options 2. Includes Encryption and Compression of Mapping Data 3. Includes Ability to Save Generated Maps to Local System 4. Fully Extensible for Data Format, Display, Input and Output
Web Mapping Service Client	1. Allows Compatibility with Web Mapping Service Based Clients and WMS Portals
GIDB API	1. Allows GIDB data sources and themes to be used within independently developed applications

instead, they appear as a single data source with many different categories of data and a single access protocol [Wilson et al., 2003].

From this point, constructing client software to allow access to the GIDB portal is straightforward. The GIDB framework provides all of the available data types and query methods. The core web/application server provides network access to the GIDB through HTTP. Custom protocols are available for accessing the system, but HTTP is the preferred method for simplicity and ubiquity. There are several client software packages available for accessing the GIDB portal. Table 7.2 lists the client packages and their benefits. Figure 7.6 shows the relationship of the various client packages to the GIDB portal. Each client package has access to the same data from a common access point.

3.4 GIDB Thematic Layer Server

The GIDB can successfully connect many different sources and can present the data to a variety of client software packages. However, as the number of data sources has grown from around 10 to over 500, the complexity of browsing through the thousands of available data layers has become unmanageable. This is especially true for novice GIS users who have little or no experience locating needed data and merging it with other data types.

The solution to this problem is to add another layer of abstraction between the user and the original data. Instead of listing all the available data sources

and layers in one long list, we have adopted an alternative method that presents the layers according to ISO 19115 standard themes. Table 7.3 presents the top level themes according to this specification [Peng and Tsou, 2003].

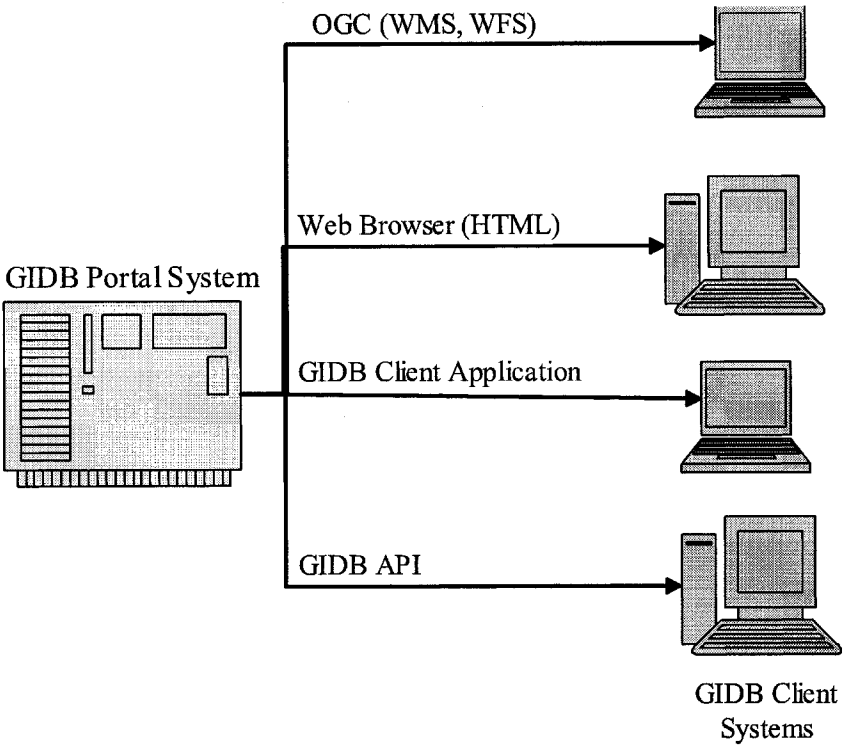


Figure 7.6. GIDB Client Access Methods.

These themes represent the top level presentation of data types to the end user. As an example, under the theme "Biologic and Ecologic Information" in the GIDB the following data layers are listed: Biological Distinctiveness, Ecoregion Divisions, Ecoregion Domains, Ecoregion Provinces, NOAA Mussel Watch, TNC Ecoregions, Terrestrial Ecoregions, Threat to Terrestrial Ecoregions, US Fish & Wildlife Refuges, USGS NDVI Vegetation Index, and WWF Ecoregions. These eleven data layers are representative of over 100 data layers stored in up to seven different data servers. Thus, users of a GIDB client package can quickly get to the required data, with little or no knowledge of the data location or configuration. The list of themes allows users to quickly navigate through the vast amount of data available in the GIDB. This component of the GIDB is called the "Theme Server" and is the most valuable feature to new users.

Table 7.3. ISO 19915 Standard Geographical Themes.

	ISO 19915 Standard Geographical Themes
1.	Administrative and Political Boundaries
2.	Agriculture and Farming
3.	Atmospheric and Climatic Data
4.	Base Maps, Scanned Maps and Charts
5.	Biologic and Ecologic Information
6.	Cadastral and Legal Land Descriptions
7.	Cultural and Demographic Information
8.	Business and Economic Information
9.	Earth Surface Characteristics and Land Cover
10.	Elevation and Derived Products
11.	Emergency Management
12.	Environmental Monitoring and Modeling
13.	Facilities, Buildings and Structures
14.	Fresh Water Resources and Characteristics
15.	Geodetic Networks and Control Points
16.	Geologic and Geophysical Information
17.	Human Health and Disease
18.	Imagery and Photographs
19.	Ocean and Estuarine Resources and Characteristics
20.	Tourism and Recreation
21.	Transportation Networks and Models
22.	Utility Distribution Networks

Much work must be done to make the GIDB Theme Server function effectively. Consider the example in which a user browsing the "Tourism and Recreation" theme selects the "State Park Areas" data layer. If the current map view occupies ten states, then ten servers have to be contacted and ten different data sets have to be requested. The GIDB Theme Server does all this "behind the scenes," and even though the data comes from all over the country, it appears as one source and one layer to the end user. The Theme Server also manages multiple data layers for multiple scales. As a user moves across different map scales, layers are switched to provide the most appropriate data.

For the Theme Server to appropriately link and merge data sets, extensive cataloging of the data must take place ahead of time. This is the most time consuming activity in maintaining the GIDB portal. New servers and data layers are continually appearing and being linked into the system. However, the effort in cataloging and linking in the data sources is well invested. The GIDB portal allows data sources to be configured once and then they are permanently available to all GIDB users.

The Theme Server completes the data type and source abstraction software architecture which makes the GIDB portal the best available system for linking

distributed data sources. The original data is spread out over the world on many different servers, and appears in many different formats requiring multiple access protocols. The GIDB portal provides a single access point, single access protocol, and a coherent theme-based data layout.

4. Example Scenarios

The following examples show configurations of the GIDB Portal System that provide an integrated view of multiple types of data sources. The first scenario is a hypothetical illustration to demonstrate the challenges involved in managing streamed object locations. The second example is a description of the actual manner in which the GIDB manages weather data that it obtains in a variety of ways.

4.1 Serving Moving Objects

To fulfill the requirements of this scenario, the GIDB Client must display objects on the map display as the move around the earth. These objects can include commercial aircraft, military equipment, and ocean going vessels. Suppose that several servers provide this information to the GIDB. The first server, Server 1, is a traditional database system that allows queries at any time. This server contains the locations of oil tankers around the world.

Another source, Server 2, does not permit queries but sends out data messages at a constant rate, and each message contains the locations of ships in a naval battle group. The third and final server, Server 3, sends out messages containing sightings of "enemy" ships. The sightings are not predictable, and are sent out whenever they are available.

At this point it is useful to discuss in more depth, the traditional client/server framework. The client in the client/server relationship is generally the party who initiates communication. This is the case with web browsers, who request web pages from web servers. Likewise, servers traditionally wait for requests to come from clients and then service the requests in some manner.

The first server in this scenario follows the traditional framework. However, the other two do not. They initiate communication when they send out messages. This presents a difficulty for the GIDB. Our framework was designed to deal with passive servers that wait for data requests and then respond with the requested data.

The software structure of the GIDB Server, with modification, is able to accept incoming data messages. However, this is not true for the GIDB client application. It has been designed to act only as a client, and it can only receive data in the form of responses to HTTP requests. This is more a security restriction than a technical requirement. As mentioned in Section 2.3, this type of restriction is fairly common. Servers readily accept many incoming con-

nections; therefore, they are often considered more susceptible than clients to attack or interference. Because of this, the GIDB client application is unable to accept incoming connections.

Since the GIDB client application cannot accept incoming messages from the server, we have to model the effect of server initiated messages. The solution is straightforward. We simply configure the client to request updates or refreshes from the GIDB server at a frequent pace. The GIDB server takes the requests from the client and sends back the latest data available from its Server 1, the GIDB server simply passes on the query from the client to the other server. Figure 7.7 shows diagrams the system relationships in this scenario.

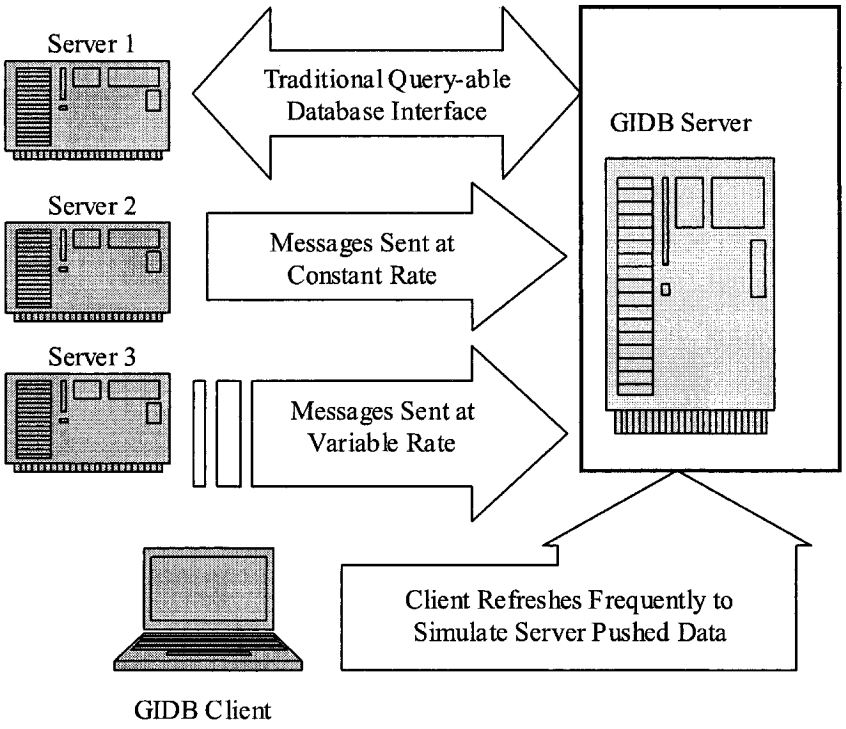


Figure 7.7. Diagram for First Scenario.

In this scenario the GIDB client is able to display the locations of the various objects on a map display as updates of the locations are received/requested by the GIDB server.

4.2 Serving Meteorological and Oceanographic Data

The GIDB currently integrates meteorological and oceanographic data from several different sources. Four sources will be considered in this discussion to demonstrate the variety of ways in which the GIDB must be able to communicate with data servers. The first weather source is METCAST, which is produced by the U.S. Navy's Fleet Numerical Meteorological and Oceanographic Center. METCAST data is provided through a custom retrieval service. Through the service, the GIDB requests an XML (Extensible Markup Language) catalog. This catalog lists all the products which are available on the server. These products include satellite weather imagery, weather observations in text form, and binary output from meteorological numerical models.

The GIDB periodically requests, downloads, and parses the catalog of products. It then adds the list of available products in the catalog to its internal listing of available data types. Thus the user can browse the list of available products from the METCAST server along with the listing of everything else in the GIDB. When the user requests METCAST data, the GIDB converts the users request into one compatible with the METCAST retrieval service. It then requests, downloads, and displays the requested data. Note that the data is never cached or stored on the GIDB; it is simply passed through to the user.

The second source of weather data in the GIDB is provided by NCEP (National Centers for Environmental Prediction). This data is very similar in content to that available from METCAST; however, NCEP does not provide a retrieval service, it simply puts its data on a publicly accessible FTP server. Thus, the GIDB periodically logs in to the server, and browses available data types. From this browse session, a listing of available products is generated. When a GIDB user requests a product from this server, the GIDB initiates another FTP session, and downloads the requested data. The data is then converted to an appropriate format and passed to the user.

The third source of data in this example is a WMS server provided by the Norwegian Meteorological Service. Using the WMS "GetCapabilities" request, the GIDB is able acquire a listing of available products. Thus, when a user requests data from this site, the GIDB requests it using the appropriate WMS formatted request. The GIDB the converts the WMS response to the appropriate format and passes it to the user. In this case the data is not cached or stored in the GIDB.

The previous three cases are similar, for in each case the download of the data is in response to a user request. However, consider the fourth method which the GIDB uses to get weather data. In this case, the GIDB receives periodic emails from NLMOC (Naval Atlantic Meteorology and Oceanography Center). These emails contain geographic data files with storm alerts, locations of weather events, etc. They are sent out by the remote server to the GIDB whenever

updated data becomes available. In this case the data has to be treated as a stream. Specific queries on the data, determinations of the size and scope of the data, and frequency of updates are not available. However, for this data to be part of the GIDB, it has to appear as a traditional data source. In other words, this data source has to allow queries and respond to the user requests. The solution is that the GIDB simply caches all the emails from the source and acts as the database for this source. When the emails are received, the data they contain is converted in the appropriate format and indexed within the GIDB.

While the various methods for acquiring data in this scenario are all fairly straightforward, they are nonetheless quite different from one another. As such, the GIDB serves a significant role in integrating all these sources into a coherent (and virtual) source. To the GIDB user, all four data sources appear to be one source, with one query protocol, and one data format. Furthermore, using the GIDB client application, the user can visualize data from all four sources in one place.

Acknowledgements

The Naval Research Laboratory would like to thank the National Guard Bureau for funding the research and deployment of the GIDB Portal System. Their sponsorship has resulted in the enhancement of the GIDB Portal System to meet an expanded set of homeland security and law enforcement requirements. Furthermore, under this research program, the GIDB Portal System as grown to integrate over 500 data sources, and is used by over 5000 users per month. For more information on the National Guard Bureau's Digital Mapping System, see http://dms.gtri.gatech.edu.

Notes

1. GIDB is a registered trademark of Naval Research Lab.

References

Chung M., R. Wilson, K. Shaw, F. Petry, M. Cobb (2001). Querying Multiple Data Sources via an Object-Oriented Spatial Query Interface and Framework. In *Journal of Visual Languages and Computing*, 12(1), February:37-60.

OpenGIS Documents, Retrieved on 1st June, 2000 from the World Wide Web: http://www.opengis.org/specs/?page=baseline

Peng, Zhong-Ren and Ming-Hsiang Tsou.(2003). *Internet GIS: Distributed Geographic Information Services for the Internet and Wireless Network.* Wiley and Sons, Hoboken, New Jersey.

Tanenbaum, Andrew (2003). *Computer Networks*. Prentice Hall PTR, Upper Saddle, New Jersey.

Wilson, R., M. Cobb, F. McCreedy, R. Ladner, D. Olivier, T. Lovitt, K. Shaw, F. Petry, M. Abdelguerfi (2003). Geographical Data Interchange Using XML-Enabled Technology within the GIDB System. Chapter 13, In *XML Data Management*, Akmal B. Chaudhri, editor, Addison-Wesley, Boston.

Chapter 8

STREAMING DATA DISSEMINATION USING PEER-PEER SYSTEMS

Shetal Shah, and Krithi Ramamritham
Department of Computer Science and Engineering
Indian Institute of Technology Bombay, India
shetals,krithi@cse.iitb.ac.in

Abstract Many characteristics of peer-peer systems make them suitable for addressing the traditional problems of information storage and dissemination. Peer-peer systems give a distributed solution to these problems. Typically, peer-peer systems (research prototypes or commercial systems) have dynamic topologies where peers join and leave the network at any point. However, the information that is stored and queried in these peers is assumed to be static. Most of these current peer-peer systems do not deal with data that is changing dynamically, i.e., data that changes rapidly and unpredictably. This chapter first examines a few of the existing peer-peer systems and the various issues that they address. It then discusses some of the research issues in using peer-peer systems for managing dynamic or streaming data and presents a peer-peer solution for the dissemination of dynamic data.

Keywords: data dissemination, cooperation, data coherence, peer-peer systems.

1. Introduction

The Internet and the web are increasingly being used to disseminate fast changing data such as sensor data, traffic and weather information, stock prices, sports scores, and even health monitoring information [Neoganesh]. These data items are *highly dynamic*, i.e., the data changes continuously and rapidly, *streamed* [1] in real-time, i.e., new data can be viewed as being appended to the old or historical data, and *aperiodic*, i.e., the time between the updates and the value of the updates are not known apriori. Increasingly, users are interested in monitoring such data for on-line decision making. The growth of the Internet has made the problem of managing dynamic data interesting and challenging.

Resource limitations at a source of dynamic data limit the number of users that can be served directly by the source. A natural solution to this is to have a set of repositories which replicate the source data and serve it to users in their geographic proximity. Services like *Akamai* and IBM's edge server technology are exemplars of such networks of repositories, which aim to provide better services by shifting most of the work to the edge of the network (closer to the end users). Although such systems scale quite well for static data, when the data changes rapidly, the quality of service at a repository farther from the data source deteriorates. In general, replication reduces the load on the sources and increases the number of users served, but replication of time-varying data introduces new challenges. When a user gets data from a repository, (s)he expects the data to be in sync with the source. This may not be possible at all times with rapidly changing data due to the inherent delays in the system. Neither is broadcasting all updates to all users a feasible solution as this will increase the number of messages in the system leading to further delays and may even result in bottlenecks. In other words, unless updates to the data are carefully disseminated from sources to repositories (to keep them coherent/in sync with the sources), the communication and computation overheads involved can result in delays and lead to scalability problems, further contributing to loss of data coherence.

In this chapter we look at some of the techniques: algorithms and architectures for the dissemination of fast changing data. We first take a look at some of the existing peer-peer systems[2] and the issues that they address and then turn our attention to a peer-peer system for dynamic data dissemination.

2. Information-based Peer-Peer systems

In this section, we present some information based peer-peer systems[3]. We first take a look at the issues that need to be addressed when building any peer-peer system and then describe how a few of the information-based systems.

2.1 Summary of Issues in Information-Based Peer-Peer Systems

We first summarize the issues that many information based peer-peer systems address and then take a brief look at some of the existing peer to peer systems for information storage. Some of the issues are:

- **Search**: As data is stored on a large scale in peer-peer systems, one needs to be able to search for the required data effectively. This outlook has led to wealth of research on searching in the peer-peer community. Different kinds of information storage applications have led to different kinds of search mechanisms:

- *Centralized versus Distributed*: In peer-peer systems like Napster, the search is done by a centralized component. Queries are directed to this centralized component which returns a list of peers containing the requested data. In other peer-peer systems, like Gnutella, DHTs, there is no centralized component and hence the search is distributed.
- *Exact match searches, keyword searches, wild card searches and other complex searches*: Peer-peer systems like Distributed Hash Tables [Ratnasamy et al., 2001],[Stoica et al., 2001], provide an extremely efficient way to look up an entity in the system. Typically, the search takes a single key and returns a match. They are used in applications which primarily want exact match answers. However, there is a lot of ongoing research to build on this simple operation to handle keywords and wild cards.

  Keyword searches are an inherent part of the design of some peer-peer systems. Given a set of keywords, they return a list of entities that completely or partially match the keywords. Peer-peer systems like Gnutella [Gnutella], Gia [Chawathe et al., 2003] provide this kind of search.
- *Popular vs non-popular*. Some of the search techniques (e.g., Gia [Chawathe et al., 2003]) developed are very efficient for popular content but take time for data which is relatively rare. On the other hand, search mechanisms developed for DHT's take the same time for any data irrespective of its popularity.
- *Exhaustive vs Partial*: This criterion determines the number of matches that the system returns - whether it returns all matches or only the first n matches, $n \geq 1$.

- **Topology**: The topology of the peer-peer system is governed by many factors. Some of them are:
  - *Centralized Component or Completely Distributed*: Some peer-peer systems (like Napster) have a centralized component which might take care of administration and other functionality like keeping track of current peers, etc. In a completely distributed system these functionalities are also distributed amongst the peers.
  - *Dedicated or Dynamic*: Some peer-peer systems consist of dedicated peers which are a part of the system. However, in many peer-peer systems the topology is dynamic, i.e., the peers come and go and hence the topology build algorithms are also adaptive to handle this. Some peer-peer systems follow the middle path wherein

some of the peers are fixed (dedicated), whereas the rest join and leave the network as they desire.

- *Search Criteria*: Just as an application using a peer-peer system governs the search criteria to a large extent, so is the topology of the system governed by search functionality. The peer to peer systems are built in such a manner that the search algorithms are efficient.

- **Fault Tolerance**: In any system, one needs to to be prepared for failures, for e.g., peer crashes and network failures. In addition to this, in a dynamic peer-peer system peers may come and go at any time. Hence, peer-peer systems need to be fault tolerant. A well designed peer-peer system has fault tolerance built into it. Typically, fault tolerance is achieved through redundancy. The exact techniques for achieving fault tolerance vary from system to system.

- **Reputation**: One reason why peer-peer systems, where peers join and leave the network as they desire, are attractive is that the peers get a certain amount of anonymity. However, this leaves the systems open to malicious peers who are interested in sending malicious/invalid data to other peers. One way to detect malicious peers is by assigning reputations to the peers which gives an indication of the trustworthiness of a peer. To do this in a fair and distributed fashion is a challenging task. Finding the reputation of peers is another hot topic of research with various kinds of schemes proposed: (i) global schemes where all peers contribute to calculation of the reputation of a peer, (ii) local schemes where peer reputations are calculated locally and (iii) mix of the two schemes. [Marti and Molina, 2004] proposes a local scheme wherein each peer calculates the reputation of other peers as the % of authentic files it got from that peer. Additionally, authors of [Marti and Molina, 2004] propose algorithms with a mixed flavour of local and global where one can also get reputation information from other *friend* peers or known reputed peers.

- **Load Balancing**: Overloading any part of a system, potentially, leads to poor performance of the system. Unless load balancing is inbuilt in the design of a peer-peer system one can easily come across scenarios where some (or all) the peers in the system, are overloaded, leading to degraded performance. For e.g., in Gnutella, queries are flooded in the network. A large number of queries could lead to the overloading of the nodes, affecting the performance and the scalability of the system.

2.2 Some Existing Peer-Peer Systems

Having presented the issues that peer-peer systems are designed to address, we now examine a few of the existing peer to peer systems, to understand how

they address these issues. We also discuss the specific applications, if any, for which they are targeted.

2.3 Napster

Napster was the first system to use peers for information storage. The target application was the storage and sharing of files containing music. Peers could join and leave the Napster network anytime. Napster had a centralized component which kept track of the currently connected peers and also a list of the data available at each peer. A query for a music file went to the centralized component which returned a list of peers containing this file. One could then download the file from any of peers given in the list. Napster achieved a highly functional design by providing centralized search and distributed download. In Napster as more peers joined in the network, more was the aggregate download capacity of the network. However, having a centralized component in the system increases the vulnerability of the system to overloading and failures.

2.4 Gnutella

Gnutella [Gnutella] is a peer-peer system which is completely distributed. It is an unstructured peer-peer system wherein the topology of the network and the position of files in the network are largely unconstrained. The latest version of Gnutella supports the notion of ultra peers or super peers wherein these nodes have a higher degree (i.e., more number of neighbours[4]) than the rest of the nodes in the network.

Gnutella supports keyword-based queries. Typically, a query matches more than one file in the network. Queries for files are flooded in the network in a limited radius. To locate a file, a node queries each of its neighbours, who in turn query each of their neighbours and so on until the query reaches all the nodes within a certain radius of the original querier. When a peer gets a query, it sends a list of all the files matching the query to the originating node.

As queries tend to flood the network, this schema is fairly fault tolerant. Even if nodes join and leave the network in the interim, the chances that the originator will get back a response is high. Flooding however increases the load on the nodes and hence this scheme does not scale too well for a large number of queries.

2.5 Gia

Gia [Chawathe et al., 2003] is a unstructured peer-peer system like Gnutella with the goal of being scalable, i.e., to handle a higher aggregate query rate and also to be able to function well under increasing system size.

Gia has two kinds of peers, super nodes or high capacity peers and ordinary peers. The capacity of a peer is given by the number of queries that it can handle

per unit time. Each peer controls the number of queries that it receives by use of tokens. Each peer sends tokens to its neighbours. The sum total of the tokens that the node sends represents its query processing rate. A token represents that the peer is willing to accept a single query from its neighbour. In Gia, a node X can direct a query to a neighbour Y only if Y has expressed a willingness to receive queries from X. Also, tokens are not distributed uniformly among neighbours. The distribution takes into account the capacity of the neighbours. This means that a high capacity node will get more tokens from its neighbours than a low capacity node.

In Gia, low capacity peers are placed close to the high capacity peers. Additionally, all peers maintain pointers to contents stored in their neighbours. When two peers become neighbours, they exchange the indices on their data which are periodically updated for incremental changes. Whenever a query arrives at a peer, it tries to find matches with not only data stored at the peer but also using the neighbours' indices. Since the high capacity peers will typically have a lot of neighbours, the chances of finding a match for a query at a high capacity peer is higher.

Search is keyword based search where popular data items are favoured.

Fault tolerance in Gia is achieved with the help of keep alive messages, when a node forwards a query enough times, it sends back a keep alive message to the originator. If the originator does not get any response or keep alive messages in a certain time interval it then reissues the query.

2.6 Semantic Overlay Networks

Semantic Overlay Networks [Crespo and Garcia-Molina, 2002] are networks where peers with similar content form a network. The documents in a peer are first classified using a classifier into concepts and the peers belong to the networks representing those concepts.

A query for a document is again classified into concept(s) and the relevant networks are searched for this document. This approach reduces the number of peers that need to be searched to find a document.

Commentary: The work of Semantic Overlay Networks depends heavily on the classifiers and the classification hierarchy. This restricts the data that can be stored in a Semantic Overlay Network to that which can be classified. In fact, if one wants to store various types of data, then one may have to combine more than one classification hierarchy. This may compound the effect of errors in classification.

2.7 Distributed Hash Tables

Distributed Hash Table protocols like Chord and CAN are scalable protocols for lookup in a dynamic peer-peer system with frequent node arrivals and departures.

Chord [Stoica et al., 2001] uses consistent hashing to assign keys to nodes and uses some routing information to locate a few other nodes in the network. Consistent hashing ensures that with high probability, all nodes roughly receive the same number of keys. In an N node network, each node maintains information about only $O(\log N)$ nodes and a lookup requires $O(\log N)$ messages.

In Chord, *identifiers* are logically arranged in the form of a ring called the Chord ring. Both nodes and keys are mapped to identifiers. Nodes are mapped by means of their IP address and keys are mapped based on their values. The hash function is chosen in such a manner that the nodes and the keys don't map to the same hash value. A key k is stored to the first node that is equal to or follows the key on this ring. The node is called the *successor* node of key k.

For efficient lookup, each node maintains information about a few other nodes in the Chord ring by means of "finger tables". In particular, the i^{th} entry in the finger table of node n contains a pointer to the first node that succeeds n by at least 2^{i-1} in the identifier space. The first entry in this finger table is the successor of the node in the identifier space. Finger tables are periodically updated to account for frequent node arrivals and departures.

CAN is another distributed hash table based approach that maps keys to a logical d dimension coordinate space. The coordinate space is split into zones in which each zone is taken care of by one node. Two nodes are neighbours if their coordinate spans overlap along $d-1$ dimensions and abut along one dimension. Every node maintains information about its neighbours.

A <key, value> pair is hashed to a point p in the d dimensional space using the key. The <key, value> pair is then stored in the node owning the zone containing p. To locate the value for a key k, the node requesting it finds the point p that the keys map to using the same hash function and routes the query to the node containing the point via its neighbours.

When a node n wants to join the CAN, it randomly picks up a point p and sends a join message to the CAN with this point p. This is routed to the node in whose zone p currently is. This zone is then split into half, one half belonging to the original node and the other half belonging to the new node. The keys relevant to the new node, exchange hands and the neighbour information of the two nodes and some of the neighbours of the old node is then updated. When a node n leaves the system, it explicitly hands over its zone to one of its neighbours.

Commentary: Peer-peer systems like CAN and Chord mandate a specific network structure and assume total control over the location of data. This results in lack of node autonomy.

3. Multimedia Streaming Using Peer-Peer Systems

The peer-peer systems we have seen so far deal with the issue of storage of files and given a query, their retrieval. We shall now take a look at some of the work that has been done in the field of multimedia streaming data in the presence of peers. As seen from the previous section, peer-peer networks can be very dynamic and diverse. Most of the streaming peer-peer systems address the issue of maintaining the quality of the streaming data in a dynamic heterogeneous peer-peer network. We take a look at two pieces of work, one [Padmanabhan et al., 2002] which builds a content distribution network for streaming and the other [Hefeeda et al., 2003] which is streaming application built on top of a look-up based peer to peer system. A more comprehensive survey of peer-peer systems for media streaming can be found at [Fuentes, 2002].

CoopNet [Padmanabhan et al., 2002] is a content distribution network for live media streaming. CoopNet is a combination of a client-server architecture and a peer-peer system. Normally, the network works as a client server system, however when the server is overloaded then the clients cooperate with each other, acting like peers, for effective media streaming. In [Padmanabhan et al., 2002], the authors present algorithms to maintain live and on-demand streaming under frequent node arrivals and departures.

CollectCast [Hefeeda et al., 2003] is an application level media streaming service for peer-peer systems. To play media files, a single peer may not be able to or may be unwilling to contribute the bandwidth required for the streaming. Downloading the file is a possible solution but media files are typically large in size and hence take a long time to download. CollectCast presents a solution wherein multiple peers participate in streaming data to a peer in a single streaming session to get the best possible quality.

CollectCast works on top of a look-up based peer-peer system. When CollectCast gets a request from a peer for a media file, it first issues a look-up request on the underlying peer-peer system. The peer-peer system returns a set of peers having the media file. Of the set of peers returned by the peer-peer system, CollectCast partitions this set into active peers and standby peers based on the information of the underlying topology. CollectCast uses a path-based topology aware selection wherein available bandwidth and loss rate for each segment of the path between each of the senders and the receivers is predetermined. If multiple senders share the same physical path, this is reflected in the calculation of availability and bandwidth loss.

Once CollectCast partitions the senders into active and standby the rate and data assignment component assigns to send sender in the active list, the rate at which the sender is supposed to send and what part of the actual data is to be sent. The active peers send the data to the receiver in parallel. The rate and data assigned depends on the capability and capacity of the peer.

The data sent by the peers is constantly monitored. In event of a slow rate of transmission on part of the sender or sudden failures, etc., the load is redistributed amongst the remaining active peers. If this is not feasible, it then adds a peer from the stand by to the active list. The monitoring component ensures that data quality to the receiver is not lost in spite of failures in the network.

In addition to addressing the issues mentioned in the previous section, to handle media streaming the peer-peer systems also need to address:

- Merging of Data: In video streaming applications, clips of a movie can be distributed across peers. Or in a file sharing application, a file may be distributed across peers. One needs algorithms to merge this distributed data as one whole when needed. Note that this is not as easy, especially if peers are dynamic.
- Real Time Delivery issues: The peer-peer system has to guarantee the continuous delivery of the media streams in real time. This involves dealing with loss of media stream packets, failures of links and nodes and bottlenecks on the path.

4. Peer-Peer Systems for Dynamic Data Dissemination

The peer-peer systems we have seen so far deal with storage of static data, like music files, media files, etc. A major design consideration while building a peer-peer system is the ability to deal with the dynamics of the network - i.e., peers join and leave the network at any time. In the rest of this chapter we explore the dynamics along yet another axis – where the data itself is dynamic. As said earlier, dynamic data refers to data that changes continuously and at a fast rate, that is *streaming*, i.e., new data can be viewed as being appended to the old or historical data, and is *aperiodic*, i.e., the time between the updates and the value of the updates are not known apriori.

To handle dynamic data in peer-peer systems, in addition to the issues discussed so far, we need to also look at the following issues:

- Dynamics has to be considered along three issues: (i) network is dynamic, i.e., peers come and go, (ii) the data that the peers are interested in is dynamic, i.e., the data changes with time and (iii) the interests of the peers are dynamic (i.e., the data set that the peers are interested in change from time to time).
- The existing searches will also have to take care of one more dimension while dealing with peer-peer data, namely, that of freshness versus latency. One might have to choose an old or stale copy of some data compared to the latest copy if the latency to fetch the latest copy is above an acceptable threshold.

- Since the data is changing continuously, peers will need to be kept informed of the changes - hence we need effective data dissemination/ invalidation techniques.
- Peers may have a different views of the data that is changing. Techniques will be needed to form a coherent whole out of these different views.
- Since the data is changing rapidly and unpredictably, this may lead to load imbalance at the peers. We will need to enhance existing techniques or develop new load balancing techniques to handle this.
- The network should be resilient to failures.

4.1 Overview of Data Dissemination Techniques

Two basic techniques for point-point dissemination of data are widely used in practice. We begin this section with a brief description of these complementary techniques.

The Pull Approach The *pull* approach is the approach used by clients to obtain data on the World Wide Web. Whenever an user is interested in some data, the user sends a request to fetch the page from the data source. This form of dissemination is passive. It is the responsibility of the user to get the data that (s)he is interested in from the source.

The Push Approach Complimenting the pull approach is the *push* approach wherein the user registers with the data sources the data that (s)he is interested in and the source pushes all changes that the user is interested in. For e.g., publish-subscribe systems use push based delivery: users subscribe to specific data and these systems push the changed data/digests to the subscribers.

There are many flavours to these basic algorithms. Some of these are:

- *Periodic Pull*: where pull takes place periodically. This is supported by most of the current browsers.
- *Aperiodic Pull*: here the pull is aperiodic. The time between two consecutive pulls is calculated based on some on estimation techniques [Srinivasan et al., 1998], [Majumdar et al., 2003] [Zhu and Ravishankar, 2004].
- *Periodic Push*: here the source pushes the changes periodically. The periodicity of pushing a particular data item depends on how many users are interested in it, i.e., popularity and how often it changes [Acharya et al., 1997].

- *Aperiodic Push*: here the source pushes to a user whenever a change of "interest" occurs. We'll expand on this in the rest of this section.

Push and pull have complimentary properties as shown in table 8.1. In pull, a client has to keep polling to be up-to-date with the source, and this makes it extremely network intensive. Using some estimation techniques [Srinivasan et al., 1998], [Majumdar et al., 2003], [Zhu and Ravishankar, 2004] one can considerably reduce the the number of required pulls. In push, the onus of keeping a client up-to-date lies with the source - this means that the source has to maintain state about the client making push computationally expensive for dynamic data.

The overheads in both pull and push limit the number of clients that can be handled by the system. Since peer-peer systems have the potential for scalability at low cost, we present a push based peer-peer architecture for data dissemination.

To reduce push overheads, in our system, the work that the source needs to do, to push changes, is distributed amongst peers. Since current technology does not permit a push from the server to the client, we split our system into two parts, peers and simple clients, where the peers push data to each other and the clients pull the data from one or more peers.

Algorithm	**Overheads (Scalability)**		
	Communication	*Computation*	*State Space*
Push	Low	High	High
Pull	High	Low	Low

Table 8.1. Overheads in Push and Pull.

As mentioned earlier, transmission of every single update by a data source to all the users of the data is not a very practical solution. However, typically, not all users of dynamic data are interested in every change of the data. For example, a user involved in exploiting exchange disparities in different markets or an on-line stock trader may need changes of every single cent but a casual observer of currency exchange rate fluctuations or stock prices may be content with changes of a higher magnitude. This brings us to the notion of coherence requirement.

4.2 Coherence Requirement

Consider a user that needs several time-varying data items. To maintain coherency of a data item, it must be periodically refreshed. For highly dynamic data it may not be feasible to refresh every single change. An attempt to do so will result in either heavy network or source overload. To reduce network

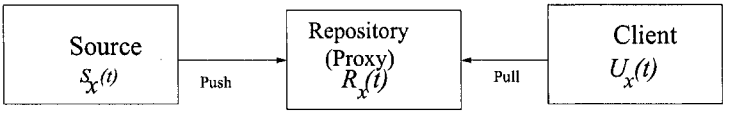


Figure 8.1. The Problem of Maintaining Coherence.

utilization as well as server load, we can exploit the fact that the user may not be interested in every change happening at the source.

We assume that a user specifies a coherence requirement c for each data item of interest. The value of c denotes the maximum permissible deviation of the value that the user has from the value at the source and thus constitutes the user-specified tolerance. Observe that c can be specified in units of *time* (e.g., the item should never be out-of-sync by more than 5 minutes) or *value* (e.g., the stock price should never be out-of-sync by more than a dollar). In this chapter, we only consider coherence requirements specified in terms of the value of the object; maintaining coherence requirements in units of time is a simpler problem that requires less sophisticated techniques (e.g., push every 5 minutes). As shown in Figure 8.1, the user is connected to the source via a set of repositories. The data items at the repositories from which a user obtains data must be refreshed in such a way that the coherence requirements are maintained. Formally, let $S_x(t)$ and $U_x(t)$ denote the value of a data item x at the source and the user, respectively, at time t (see Figure 8.1). Then, to maintain coherence, we should have, $\forall(t), |U_x(t) - S_x(t)| \leq c$.

The *fidelity* of the data seen by users depends on the degree to which their coherence needs are met. We define the fidelity f observed by a user to be the total length of time that the above inequality holds (normalized by the total length of the observations). In addition to specifying the coherence requirement c, users can also specify their fidelity requirement f for each data item so that an algorithm that is capable of handling users' fidelity requirements (as well as the coherence requirements) can adapt to users' fidelity needs.

We would like to mention here that due to the non-zero computational and communication delays in real-world networks and systems, it is impossible to achieve 100% fidelity in practice, even in expensive dedicated networks. The goal of any dissemination algorithm is to meet the coherence requirements with high fidelity in real-world settings.

4.3 A Peer-Peer Repository Framework

The focus of our work is to design and build a dynamic data distribution system that is *coherence-preserving*, i.e., the delivered data must preserve associated coherence requirements (the user specified bound on tolerable imprecision) and *resilient* to failures. To this end, we consider a system in which a set

of repositories cooperate with each other and the sources, forming a dedicated peer-peer network.

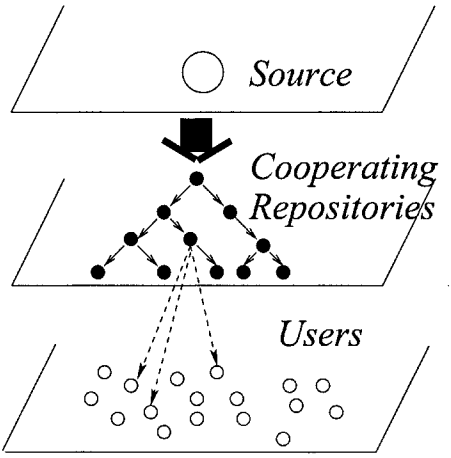


Figure 8.2. The Cooperative Repository Architecture.

As shown in Figure 8.2, our network consist of sources, repositories and clients. Clients (and repositories) need data items at some coherence requirements. Clients are connected to the source via a set of repositories. The architecture uses *push* to disseminate updates to the repositories.

For each data item we build a logical overlay network, as described below. Consider a data item x. We assume that x is served by only one source. It is possible to extend the algorithm to deal with multiple sources, but for simplicity we do not consider this case here. Let repositories $R_1, \ldots, R_n$ be interested in x. The source directly serves some of these repositories. These repositories in turn serve a subset of the remaining repositories such that the resulting network is a tree rooted at the source and consisting of repositories $R_1, \ldots, R_n$. We refer to this tree as the *dynamic data dissemination tree*, or d^3t, for x. The children of a node in the tree are also called the *dependents* of the node. Thus, a repository serves not only its users but also its dependent repositories. The source pushes updates to its dependents in the d^3t, which in turn push these changes to their dependents and the end-users. Not every update needs to be pushed to a dependent—*only those updates necessary to maintain the coherence requirements at a dependent need to be pushed.* To understand when an update should be pushed, let c^P and c^Q denote the coherence requirements of data item x at repositories P and Q, respectively. Suppose P serves Q. To effectively disseminate updates to its dependents, the coherence requirement at a repository

should be *at least as stringent as those of its dependents*:

$$c^p \leq c^q \tag{8.1}$$

Given the coherence requirement of each repository and assuming that the above condition holds for all nodes and their dependents in the d^3t, we now derive the condition that must be satisfied during the dissemination of updates. Let $x_i^s, x_{i+1}^s, x_{i+2}^s, \ldots x_{i+n}^s \ldots$ denote the sequence of updates to a data item x at the source S. This is the data stream x. Let $x_j^p, x_{j+1}^p, \ldots$ denote the sequence of updates received by a dependent repository P. Let x_j^p correspond to update x_i^s at the source and let x_{j+1}^p correspond to update x_{i+k}^s where $k \geq 1$. Then, $\forall m, 1 \leq m \leq k-1, |x_{i+m}^s - x_i^s| < c^p$.
Thus, as long as the magnitude of the difference between last disseminated value and the current value is less than the coherence requirement, the current update is not disseminated (only updates that exceed the coherence tolerance c^p are disseminated). In other words, the repository P sees only a "projection" of the sequence of updates seen at the source. Generalizing, given a d^3t, each downstream repository sees only a projection of the update sequence seen by its predecessor.

In the cooperative repository architecture, repositories filter the data that is streamed to them before forwarding the data to their dependents. Note that, in principle, a repository R can be a dependent of another repository Q for data item x whereas R could obtain data item, y, from Q. In other words the repositories form peer-peer network for selective dissemination of streaming data to each other and to the clients.

Various issues like efficient algorithms to build the dissemination network, how to make the network resilient to failures, how to decrease the response time by effective scheduling are discussed in [Shah et al., 2003]. [Shah et al., 2002] analyzes the degree of cooperation - essentially how much cooperation is good. Interestingly we found that not only is too little cooperation bad but even that too much of it also overloads the network and we had a heuristic to determine the degree of cooperation. In [Agarwal et al., 2004], we handle the issue of which client to connect to which repositories for its data needs.

5. Conclusions

In this chapter we discussed a dissemination system for dynamic data using peers. While we have solutions for some of the problems encountered in peer-peer dynamic data dissemination, there are still some issues that we need to explore: Currently, we have no notion of search in our repository network – when a client needs a data item for the first time, it approaches the data's source to determine which peer to obtain the data from. Also our network consists of

dedicated peers. To extend this to a dynamic peer network with facilities of search, merging of data, etc., is part of our future work.

Notes

1. The term data streams is also used to model situations where in addition the memory available for computation is limited.
2. We would like to mention here that our description of the peer-peer systems is not intended to be comprehensive or complete. Rather it is an attempt on our part to a give a flavour of the different kinds of peer-peer systems that exist and the various issues that are addressed by them.
3. As opposed to, say, peer-peer systems for media streaming as discussed in Section 3 or peer-peer systems for disseminating dynamic data discussed in Section 4.
4. Two peers connected to each other are called neighbours of each other.

References

Acharya, S., Franklin, M. J., and Zdonik, S. B. (1997). Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD Conference.*

Agarwal, S., Ramamritham, K., and Shah, S. (2004). Construction of a temporal coherency preserving dynamic data dissemination network. In *The 25th IEEE International Real Time Systems Symposium.*

Chawathe, Yatin, Ratnasamy, Sylvia, Breslau, Lee, Lanham, Nick, and Shenker, Scott (2003). Making Gnutella-like p2p systems scalable. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 407–418. ACM Press.

Crespo, Arturo and Garcia-Molina, H. (2002). Semantic overlay networks for p2p systems. Technical report, Computer Science Department, Stanford University.

Fuentes, Francisco (2002). An overview of media streaming over peer-to-peer networks. Technical report, Technische Univeritat Munchen.

Gnutella http://www.gnutella.com.

Hefeeda, M., Habib, A., Botev, B., Xu, D., and Bhargava, D. B. (2003). Collectcast: A peer-to-peer service for media streaming. *(Submitted to) ACM Multimedia Systems Journal.*

Majumdar, R., Moudgalya, K., and Ramamritham, K. (2003). Adaptive coherency maintenance techniques for time-varying data. In *The 24th IEEE International Real Time Systems Symposium.*

Marti, Sergio and Molina, Hector Garcia (2004). Limited reputation sharing in p2p systems. In *In ACM Conference on Electronic Commerce.*

Neoganesh http://www.openclinical.org/aisp_neoganesh.html.

Padmanabhan, V., Wang, H., Chou, P., and Sripanidkulchai, K. (2002). Distributing streaming media content using cooperative networking. In *In ACM/IEEE NOSSDAV.*

Ratnasamy, Sylvia, Francis, Paul, Handley, Mark, Karp, Richard, and Shenker, Scott (2001). A scalable content-addressable network. In *Proceedings of the*

2001 conference on Applications, technologies, architectures, and protocols for computer communications, pages 161–172. ACM Press.

Shah, S., Dharmarajan, S., and Ramamritham, K. (2003). An efficient and resilient approach to filtering and disseminating dynamic data. In *Proceedings of the 29th Conference on VLDB.*

Shah, S., Ramamritham, K., and Shenoy, P. (2002). Maintaining coherency of dynamic data in cooperating repositories. In *Proceedings of the 28th Conference on VLDB.*

Srinivasan, R., Liang, C., and Ramamritham, K. (1998). Maintaining temporal coherency of virtual data warehouses. In *The 19th IEEE Real-Time Systems Symposium.*

Stoica, Ion, Morris, Robert, Karger, David, Kaashoek, M. Frans, and Balakrishnan, Hari (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press.

Zhu, S. and Ravishankar, C. (2004). Stochastic consistency, and scalable pull-based caching for erratic data sources. In *Proceedings of the 30th Conference on VLDB.*

Index