Continuous Queries over Data Streams*

Shivnath Babu and Jennifer Widom
Stanford University
{shivnath,widom}@cs.stanford.edu
http://www-db.stanford.edu/stream

Abstract

In many recent applications, data may take the form of continuous *data streams*, rather than finite stored data sets. Several aspects of data management need to be reconsidered in the presence of data streams, offering a new research direction for the database community. In this paper we focus primarily on the problem of query processing, specifically on how to define and evaluate *continuous queries* over data streams. We address semantic issues as well as efficiency concerns. Our main contributions are threefold. First, we specify a general and flexible architecture for query processing in the presence of data streams. Second, we use our basic architecture as a tool to clarify alternative semantics and processing techniques for continuous queries. The architecture also captures most previous work on continuous queries and data streams, as well as related concepts such as triggers and materialized views. Finally, we map out research topics in the area of query processing over data streams, showing where previous work is relevant and describing problems yet to be addressed.

1 Introduction

Traditional database management systems (DBMSs) expect all data to be managed within some form of persistent *data sets*. For many recent applications, the concept of a continuous *data stream* is more appropriate than a data set. By nature, a stored data set is appropriate when significant portions of the data are queried again and again, and updates are small and/or relatively infrequent. In contrast, a data stream is appropriate when the data is changing constantly (often exclusively through insertions of new elements), and it is either unnecessary or impractical to operate on large portions of the data multiple times.

Several applications naturally generate data streams as opposed to data sets: financial tickers, performance measurements in network monitoring and traffic management, log records or click-streams in web tracking and personalization, manufacturing processes, data feeds from sensor applications, call detail records in telecommunications, and others. Because today's database systems are ill-equipped to perform any kind of special storage management or query processing for data streams, heavily stream-oriented applications tend to use a DBMS largely as an offline storage system, or not at all. Like other relatively recent new demands on data management (e.g., triggers, objects), it would be beneficial to provide stream-oriented processing as an integral part of a DBMS. Several aspects of data management need to be reconsidered in the presence of data streams. The *STREAM* (ST*anford st*RE*am dat*A M*anagement*) project at Stanford is addressing the new demands imposed by data streams on data management and processing techniques.

In this paper we focus on defining a solid framework for query processing in the presence of continuous data streams. We consider in particular *continuous queries* [TGNO92], which are queries that are issued once and then logically run continuously over the database (in contrast to traditional *one-time* queries which are run once to completion over the current data sets). In network traffic management, for example, continuous queries may be used to monitor network behavior online in order to detect anomalies (e.g., link congestion) and their cause (e.g., hardware failure, denial-of-service attack). Continuous queries may also be used to support load balancing or other network performance adjustments [DG00]. In financial applications, continuous queries may be used to monitor trends and detect fleeting opportunities [Tra]. Both of these applications are characterized by a need for continuous queries that go well beyond simple element-at-a-time processing, by rapid data streams, and by a need for timely online answers.

The organization of the rest of the paper is as follows:

- In Section 2 we provide a broad survey of previous work relevant to data stream processing and continuous queries. Although there has been only a handful of papers addressing the topic directly, a number of papers in related areas contain useful techniques and results.
- In Section 3 we introduce a concrete example to motivate our discussion of continuous queries over data streams.
- In Section 4 we define a general and flexible architecture for query processing in the presence of data

*This work was supported by the National Science Foundation under grant IIS-9811947, by NASA Ames under grant NCC2-5278, and by a Microsoft graduate fellowship.

streams. Also in Section 4 we use our basic architecture to specify alternative semantics for continuous queries, and to classify previous related work. We also use the architecture to clarify how continuous queries over data streams relate to triggers and materialized views.

- In Section 5 we map out, in some detail, a number of open research topics that must be addressed in order to realize flexible and efficient processing of continuous queries over data streams.

- Sections 6 and 7 discuss our vision of and plans for a general-purpose *Data Stream Management System* (*DSMS*).

2 Related Work

In this section we provide a general discussion of past work that relates in some way to continuous queries and/or data streams. A more technical analysis of some of the work will be provided in Section 4.3, after we present our basic architecture.

Continuous queries were an important component of the *Tapestry* system [TGNO92], which performed content-based filtering over an append-only database of email and bulletin board messages. The system supported continuous queries expressed using a quite restricted subset of SQL, in order to make guarantees about efficient (incremental) evaluation and append-only query results. The notion of continuous queries for a much wider spectrum of environments is formalized in [Bar99]. The *XFilter* content-based filtering system [AF00] performs efficient filtering of XML documents based on user profiles. The profiles are expressed as continuous queries in the *XPath* language [XPA99]. *Xyleme* [NACP01] is a similar content-based filtering system that enables very high throughput with a restricted query language. The *Tribeca* stream database manager [Sul96] provides restricted querying capability over network packet streams. We will revisit much of this work in Section 4.3.

The *Chronicle data model* [JMS95] introduced append-only ordered sequences of tuples (*chronicles*), a form of data stream. They defined a restricted view definition language and algebra that operates over chronicles together with traditional relations. The view definition restrictions, along with restrictions on the sequence order within and across chronicles, guarantees that the views can be maintained incrementally without storing any of the chronicles.

Two recent systems, *OpenCQ* [LPT99] and *NiagaraCQ* [CDTW00], support continuous queries for monitoring persistent data sets spread over a wide-area network, e.g., web sites over the internet. OpenCQ uses a query processing algorithm based on incremental view maintenance, while NiagaraCQ addresses scalability in number of queries by proposing techniques for grouping continuous queries for efficient evaluation. Within the same project as NiagaraCQ, reference [STD$^+$00] discusses the problem of providing *partial results* to long-running queries on the internet, where it is acceptable to provide an answer over some portion of the input data. The main technical challenge is handling blocking operators in query plans. As will be seen, our architecture provides a framework that captures and classifies all of these issues.

The *Alert* system [SPAM91] provides a mechanism for implementing *event-condition-action* style triggers in a conventional SQL database, by using continuous queries defined over special append-only *active tables*. In Section 4.3.3 we will discuss how Alert and trigger systems in general relate to continuous queries over data streams.

Clearly there is a relationship between continuous queries and the well-known area of *materialized views* [GM95], since materialized views are effectively queries that need to be reevaluated or incrementally updated whenever the base data changes. There are several differences between materialized views and continuous queries: continuous queries may stream rather than store their results, they may deal with append-only input relations, they may provide approximate rather than exact answers, and their processing strategy may adapt as characteristics of the data stream change. Nevertheless, much work on materialized views is captured by our architecture and is relevant to our proposed approach; see Section 4.3.4. Of particularly importance is work on *self-maintenance* [BCL89, GJM96, QGMW96]—ensuring that enough data has been saved to maintain a view even when the base data is unavailable—and the related problem of *data expiration* [GMLY98]—determining when certain base data can be discarded without compromising the ability to maintain a view.

The *Telegraph* project [AH00, HF$^+$00, UF01] shares some target applications and basic technical ideas with our problem, although the general approach is different. Telegraph uses an *adaptive* query engine to process conventional (one-time) queries efficiently under volatile and unpredictable environments (e.g., autonomous data sources over the internet, or sensor networks). The *Tukwila* system [IFF$^+$99] also supports adaptive query processing, in order to perform dynamic data integration over autonomous data sources. Adaptive query processing is likely to be useful for continuous queries over data streams, as discussed in Section 5.

Some work considers traditional data sets but treats them like (finite) data streams, processing the data in a single pass and possibly providing intermediate or "early" query results. For example, *online aggregation* [HHW97,

HH99] is a technique for handling long-running aggregation queries, continually providing a running aggregate with improving probabilistic error bounds. In more theoretical work, [HRR98] studies basic tradeoffs in processing finite data streams, specifically among storage requirements, number of passes required, and result approximations. The problem of computing approximate *quantiles* (equi-height histograms) over numeric data streams of unknown length is addressed in [MRL99] and [GK01].

Recently there has been increasing interest in *data reduction* techniques, where the general goal is to trade accuracy for performance in massive disk-resident data sets, with some obvious possible applications to data streams. A good survey appears in [B$^+$97]. In related work, *synopsis data structures* [GM99] provide a summary of a data set within acceptable levels of accuracy while being much smaller in size, and a framework for extracting synopses (*signatures*) from data streams is proposed in [CFPR00]. A variety of approximate query answering answering techniques have been developed based on data reduction and synopsis techniques including samples [AGPR99, AGP00, CMN99], histograms [IP99, PG99], and wavelets [CGRS00, VW99]. Reference [GKS01] develops histogram-based techniques to provide approximate answers for *correlated aggregate queries* over data streams. Reference [GKMS01] presents a general approach for building small-space summaries over data streams to provide approximate answers for many classes of aggregate queries.

There has been some initial work addressing data streams in the data mining community. In terms of building classical data mining models over a single data stream, reference [Hid99] considers *frequent itemsets* and *association rules*, reference [GMMO00] considers *clustering*, and references [DH00, HSD01] consider *decision trees*. The only work we know of addressing multiple data streams appears in [YSJ$^+$00], which develops algorithms to analyze *co-evolving time sequences* to forecast future values and detect correlations and outliers.

Finally, stream data management and query processing techniques are likely to draw on work in sequence databases (e.g., [SLR94]), time-series databases (e.g., [FRM94]), main-memory databases (e.g., [Tea99]), and real-time databases (e.g., [KGM95]).

3 A Concrete Example

Let us consider a representative application to illustrate the need for continuous queries over data streams and why conventional DBMS technology is inadequate. Consider the domain of *network traffic management* for a large network, e.g., the backbone network of an Internet Service Provider (ISP) [DG00]. Network-traffic-management applications typically process rapid, unpredictable, and continuous data streams, including packet traces and network performance measurements. Due to the inadequacy of conventional DBMSs to provide the kind of online continuous query processing that would be most beneficial in this domain, current traffic-management tools are either restricted to offline query processing or to online processing of simple hard-coded continuous queries, often avoiding the use of a DBMS altogether. A traffic-management system that could provide online processing of ad-hoc continuous queries over data streams would allow network operators to install, remove, and modify appropriate monitoring queries to support effective management of the ISP's network.

As a concrete example, consider an ISP that collects packet traces from two links (among others) in its network. The first link, called the *customer link*, connects the network of a customer to the ISP's network. The second link, called the *backbone link*, connects two routers within the ISP's network. Each packet trace is a continuous stream of packet headers observed on the corresponding link. For simplicity, we assume that a packet header comprises the five fields listed in Figure 1. We use PT_c and PT_b to denote the packet traces collected from the customer and backbone links respectively.

Field name	Description
saddr	IP address of packet sender
daddr	IP address of packet destination
id	Identification number given by sender so that destination can uniquely identify each packet
length	Length of packet
timestamp	Time when packet header was recorded

Figure 1: Record structure of a packet header.

A first simple continuous query (Q_1) computes the load on the backbone link averaged over one minute periods and notifies the network operator if the load exceeds a threshold T. A SQL version of Q_1 using two self-explanatory functions is:

```
Q1: Select    notifyoperator(sum(length))
    From      PT_b
    Group By  getminute(timestamp)
    Having    sum(length) > T
```

Although Q_1's functionality might be achievable using triggers in a conventional DBMS, performance concerns may dictate special techniques. For instance, if the PT_b stream is coming very fast (e.g., packets in an optical link), the only feasible approach might be to compute an approximate answer to Q_1 by sampling the data, something conventional triggers are certainly not designed for.

A more complex continuous query (Q_2) finds the fraction of traffic on the backbone link coming from the customer network. Q_2 is an example of an ad-hoc continuous query that a network operator might register to check in response to congestion, whether the customer is a likely cause.

Q_2: (Select count (*)
From PT_c As C, PT_b As B
Where C.saddr = B.saddr and C.daddr = B.daddr
and C.id = B.id) /
(Select count (*) From PT_b)

Q_2 joins streams PT_c and PT_b on their keys to count the number of common packets on the links. Since unbounded intermediate storage could potentially be required for joining two continuous data streams, the network operator might want the system to compute an approximate answer. Possible approximation methods are to allocate a fixed amount of storage and maintain *synopses* of the two streams (recall Section 2), and/or exploit application semantics—such as a high probability that joining tuples occur within a certain time window—to bound the required storage.

A final example continuous query (Q_3) monitors the top 5% source-to-destination pairs in terms of traffic on the backbone link. (We use the SQL3 `With` construct [UW97] for ease of expressing the query.)

Q_3: With Load As
(Select saddr, daddr, sum(length) as traffic
From PT_b
Group By saddr, daddr)
Select saddr, daddr, traffic
From Load As L_1
Where (Select count(*)
From Load as L_2
Where L_2.traffic $<$ L_1.traffic) $>$
(Select $0.95 \times$count(*) From Load)
Order By traffic

Processing Q_3 over the continuous data stream PT_b is especially challenging due its overall complexity and the presence of `Group By` and `Order By` clauses, which are normally "blocking" operators in a query execution plan.

Note that in addition to the issues discussed in each example, all three example queries are likely to benefit from adaptive query processing [AH00], given the unpredictable nature of network packet streams.

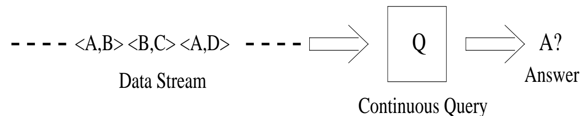

Figure 2: A continuous query Q over a single data stream.

4 Architecture for Continuous Queries

Now that we have seen a concrete example motivating data streams and continuous queries, the remainder of the paper addresses the general problem. We begin in Section 4.1 by motivating, through an extremely simple scenario, some of the most basic issues that arise when processing continuous queries over data streams. Then in Section 4.2 we present our architecture, which allows us in Section 4.3 to classify previous work in continuous queries, and to relate continuous queries to triggers and materialized views. We consider data streams that adhere to the relational model (i.e., streams of tuples), although many of the ideas and techniques are independent of the data model being considered.

4.1 Motivation

Let us consider the simplest possible scenario to illustrate the differences between querying data streams and traditional stored data sets. Suppose we have a single, continuous stream of tuples and a single query Q we are interested in answering over the stream, as illustrated in Figure 2. Q is a continuous query—we issue it once and it operates continuously as new tuples appear in the stream—and suppose we are interested in the exact answer to Q (as opposed to an approximation). Let us further suppose that the data stream is *append-only*—it has no updates or deletions—so we can think of the stream as an unbounded append-only database D. Even in this simplest of cases, there are different possible ways to handle Q, with different ramifications:

(1) Suppose we want to always store and make available the current answer A to Q. Since the "database" D may be of unbounded size, the size of A also may be unbounded (e.g., if Q is a selection query).

(2) Suppose instead we choose not to store answer A, but rather to make new tuples in A available when they occur, e.g., as another continuous data stream. Although we no longer need unbounded storage for A, we still may need unbounded storage for keeping track of tuples in the data stream in order to determine new tuples in A (e.g., if Q is a self-join).

Let us further complicate the problem by considering deletions and updates:

(3) Even if the stream is append-only, there may be updates or deletions to tuples in answer A (e.g., if Q is a group-by query with aggregation). Now, in case (2) above we may need to somehow update and delete tuples in our output data stream, in addition to generating new ones.

(4) In the most general scenario, the input data stream also may contain updates or deletions. In this case, typically more—possibly much more—of the stream needs to be stored in order to continuously determine the exact answer to Q.

One way to address these issues is to restrict the expressiveness of Q and/or impose constraints on characteristics of the data stream so that we can guarantee that the size of Q's answer A is bounded, or that the amount of extra storage needed to continuously compute A is bounded. Previous work on continuous queries, e.g., [JMS95, TGNO92, Bar99], has tended to take this approach. Another possibility is to relax the requirement that we always provide an exact answer to Q, which relates to the area of *approximate query answering* discussed in Sections 2 and 3.

In this paper we do not specifically advocate one of these approaches. Instead, we specify a general and flexible architecture that makes the choices above, and their ramifications, explicit. We further use our basic architecture to explain how continuous queries relate to triggers and materialized views, and to define a number of open research problems in processing continuous queries over data streams.

4.2 Architecture

We now introduce our general architecture for processing continuous queries over data streams, illustrated in Figure 3. For now let us consider a single continuous query Q with answer A, operating over any number of incoming data streams. Multiple continuous queries can be handled within our architecture (as implied in the figure), and we will discuss some of the interesting issues that arise in this context in Section 5.4. We also assume that the query is over data streams only, although mixing streams and conventional relations poses no particular problems.

When query Q is notified of a new tuple t in a relevant data stream, it can perform a number of actions, which are not mutually exclusive:

(i) It can determine that because of t there are new tuples in the answer A. If it is known that a new tuple a in A will remain in A "forever," then Q may send tuple a to the *Stream* component illustrated in Figure 3. In other words, *Stream* is a data stream containing tuples appended to A, similar to case (2) discussed in Section 4.1.

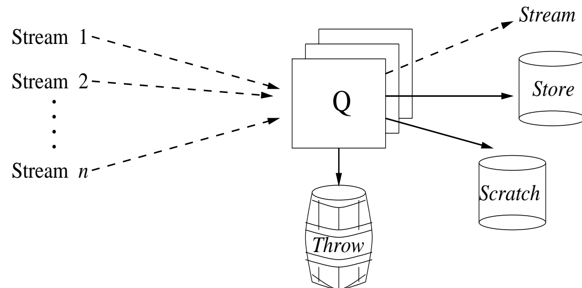

Figure 3: Architecture for processing continuous queries over data streams.

(ii) If a new tuple a is determined to be in A, but may at some time no longer be in A, then a is added to the *Store* component illustrated in Figure 3. In other words, together *Stream* and *Store* define the current query answer A. If our goal is to minimize storage for the query result, then we want to make sure that tuples are sent to *Stream* rather than *Store* whenever possible.

(iii) The new stream tuple t may cause the update or deletion of answer tuples in *Store*. Answer tuples might also be moved from *Store* to *Stream*.

(iv) We may need to save t, or save data derived from t, so that in the future we are assured of being able to compute our query result. In this case, t (or the data derived from it), is sent to the *Scratch* component of Figure 3. Combined with action (iii), we might also move data from *Store* to *Scratch*.

(v) We may not need t now or later, in which case t is sent to the *Throw* component of Figure 3. Note that *Throw* does not require any actual storage (unless we are interested in archiving unneeded data).

(vi) As a result of the new stream tuple t, we may take data previously saved in *Scratch* (or *Store*) and send it to *Throw* instead. If our goal is to minimize storage, we want to make sure that unneeded data is sent to *Throw* whenever possible, rather than *Scratch*.

4.3 The Architecture and Related Work

In this section we revisit the issues and scenarios discussed in Section 4.1, revisit the related work discussed in Section 2, and consider triggers and materialized views. In all cases we use our basic architecture as a tool for detailed understanding and comparisons.

4.3.1 Query Processing Scenarios

Let us consider query processing scenarios (1)–(4) from Section 4.1 in light of the architecture specified in Section 4.2. In scenario (1), we want to always store Q's entire current answer A. In terms of our architecture, (1) says that *Stream* is empty, *Store* always contains A, and *Scratch* contains any data that may be required to keep the answer in *Store* up-to-date. In the example case where Q is a selection query, *Store* may be of unbounded size, while *Scratch* is empty. Conversely in scenario (2) we want to make A available exclusively as a data stream, i.e., *Stream* streams the entire answer to A while *Store* is empty. In the example case where Q is a self-join, we can send all answer tuples to *Stream* since they will remain in the result forever, but *Scratch* may need to grow without bound.

Scenario (3) covers the case where answer A can have updates and deletions even when the input streams are append-only, e.g., a query that performs grouping and aggregation. Scenario (4) further extends to the case where the input streams may have updates and deletions. As an example, suppose Q is a group-by query over a single data stream with a *min* aggregation function. Since *min* is monotonic for insertions, in scenario (3) A is maintained in *Store*, and *Scratch* can remain empty. However, in scenario (4) unbounded storage is required for *Scratch* to ensure that the *min* values over the entire stream can always be computed. In both cases, the only time answer tuples can be sent to *Stream*, or moved from *Store* to *Stream*, is when it is known that for some group there will be no further insertions, updates, or deletions of tuples falling into that group.[1]

4.3.2 Previous Related Work

We now revisit some of the related work discussed in Section 2, characterizing it in terms of our basic architecture. Note that citations are not repeated in this section except when needed to identify the work being discussed. Also note that some of the related work from Section 2 is revisited instead in Section 4.3.3 on triggers or Section 4.3.4 on views.

Recall that the Tapestry system supports restricted continuous queries over append-only data sets. In Tapestry, a continuous query Q is rewritten into its *minimum bounding monotone query* Q^M, which is then rewritten into an *incremental query* Q^I. As a monotone continuous query, Q^M has the property that its answer changes only by addition of new tuples, so in terms of our architecture all answer tuples can be sent to *Stream* and *Store* is empty. The incremental version Q^I of the query is meant to improve the efficiency of computing new answer tuples when new input tuples are appended, but there is no mechanism for guaranteeing that *Scratch* will not grow without bound.

The work in [STD$^+$00] on maintaining partial results for long-running queries is similar to Scenario (3) in Section 4.1. It maintains the current partial result in *Store* and any extra needed information in *Scratch*. Our discussion of new query processing techniques in Section 5.3 is relevant to the problem addressed in [STD$^+$00], and we believe that based on these techniques it is possible to exploit monotonicity more aggressively to improve upon the algorithm in [STD$^+$00], reducing the data saved in *Scratch*. *OpenCQ* and *NiagaraCQ* consider Scenario (4) in Section 4.1, but they are geared towards data sets that change primarily through in-place updates. Thus, they do not address the problem of *Store* or *Scratch* growing without bound.

A number of systems perform tuple-at-a-time processing over their input data streams: each time a new stream element arrives, the element is moved directly to either *Stream* or *Throw*, without consulting any other data in the stream. Packet routing and simple network algorithms have this characteristic [Tan96], although for network traffic management more sophisticated stream processing is needed, as seen in Section 3. The *XFilter* and *Xyleme* systems discussed in Section 2 also perform element-at-a-time processing although the elements are XML documents.

Basic online aggregation [HHW97] maintains the current aggregate in *Store* along with an estimate of the error, and an empty *Scratch*. Follow-on work that extends online aggregation to joins [HH99] does need to maintain previously seen tuples in *Scratch*. Finally, the body of work in approximate query answering focuses primarily on making the best possible use of a limited size *Scratch* by storing only small synopses (summaries) of the data. References [GMP97, MRL99, MVW00, Vit85] address the problem of updating the synopses (i.e., *Scratch*) efficiently when the underlying data changes.

4.3.3 Triggers

Triggers, also called *event-condition-action* rules, are used to monitor events and conditions in databases, and to execute actions automatically when specific situations are detected [WC96]. In the Alert system introduced in Section 2, triggers are implemented by means of continuous queries over *active tables*. Each tuple in an active table represents an *event*, which is an update on a conventional stored table. When a new tuple is added to one of the active tables, each continuous query involving the ta-

[1] Note that we are assuming *Stream* is constrained to be append-only, even though in scenario (4) we discuss input streams with updates and deletions. If we allow updates and deletions to *Stream* tuples, then we are always free to send answer tuples to *Stream* instead of *Store*, since we can update or delete them later.

ble is evaluated, and the trigger *action* is invoked on each new tuple in the query result.

Our mapping from triggers to the architecture of Figure 3 is based on (and slightly generalizes) the Alert approach. We assume that events to be monitored are generated as data streams, and we allow continuous queries over any number of data streams together with conventional stored tables. As in Alert, these queries perform event and condition monitoring. For launching trigger actions, like Alert we assume that the desired actions are performed by SQL data manipulation commands and user-defined stored procedures specified as part of the continuous queries (e.g., query Q_1 in Section 3). In terms of our architecture, since there is no query "answer" in triggers, *Stream* and *Store* may remain empty, while *Scratch* is used for any data required to monitor complex events or evaluate conditions. Alternatively, depending on the desired trigger behavior and application interaction, actions could send results to *Stream*.

There are a number of benefits to using continuous queries over data streams to provide trigger functionality. Continuous queries specified on event streams together with conventional tables enable complex multi-table events and conditions to be monitored, equivalent to the most powerful trigger language proposals we know of [WC96]. More importantly, trigger processing would benefit automatically from efficient data management and processing techniques for continuous queries over data streams, such as specialized query optimization techniques (Section 5.3).

4.3.4 Materialized Views

Materialized views, whether in a conventional DBMS or in a *data warehousing* environment [GM95], fall naturally into our architecture. The base data over which the views are defined, if not available in conventional stored tables, is stored in *Scratch*. The view itself is maintained in *Store*. Updates to the base data can be represented as one or more data streams, as discussed in Section 4.3.3 for triggers. In terms of this mapping, work on materialized view self-maintenance and expiration, discussed in Section 2, is geared specifically towards minimizing the size of *Scratch*. Pure self-maintenance guarantees that *Scratch* is empty [BCL89, GJM96], although for many views pure self-maintainability is impossible, so *auxiliary views* must be stored and maintained in *Scratch* [QGMW96]. Data expiration exploits constraints to determine precisely when data can be removed from *Scratch*, although no bounds on the size of *Scratch* are guaranteed. The Chronicle data model discussed in Section 2 for materialized views is designed to ensure bounded storage for *Scratch*, but like pure self-maintainability it restricts the allowable view definitions significantly. To the best of our knowledge, no work on materialized views has addressed the problem of bounding the size of the materialized view itself, so that the size of *Store* also can be bounded.

5 Research Problems

In this section we outline a number of research problems associated with processing continuous queries over data streams. We begin at a relatively global level, becoming more detailed as the section progresses. In several cases the architecture of Section 4.2 is used to make the problems and issues more concrete.

5.1 Basic Problems and Techniques

At the most global level, what sets continuous queries over data streams apart from previous work is a unique combination of:

- **Online processing.** The applications discussed in Section 1 require that continuous queries are processed, well, continuously. Specifically, when new tuples arrive in a data stream they generally must be "consumed" immediately, usually performing one or more of actions (i)–(vi) from Section 4.2. In some applications the tuples may arrive so fast that some of them need to be ignored entirely.

- **Storage constraints.** In the general case for continuous data streams, the amount of storage required for the answer to a continuous query, or to ensure that the answer always can be computed, may be unbounded (recall Section 4.1). Furthermore, even if there is "nearly" unbounded storage available on disk or other tertiary devices, performance requirements may be such that *Store* and/or *Scratch* from Figure 3 need to reside in a limited amount of main memory.

While neither of these problems in isolation is entirely new, dealing with them together, while at the same time offering the full functionality and efficiency of a database query processor, is a new challenge.

Next we mention three basic techniques that have been explored primarily in other contexts within the database or broader Computer Science research community. All of them appear directly relevant to our problem.

- **Summarization.** *Summaries* (or data *synopses*) provide a concise representation of a data set at the expense of some accuracy. As discussed in Section 2, many techniques for summarization have been developed, including *sampling*, *histograms*, and *wavelets*. (See Section 2 for citations.) We expect summarization to play an important role in query processing over data streams due to the storage constraints

discussed above. New issues to resolve in the data stream environment include: (i) how to make guarantees about accuracy of continuous query results based on summaries; (ii) how to maintain summaries efficiently in the presence of very rapid data streams; (iii) what summarization techniques are best for unpredictable data streams. We revisit some of these issues in Section 5.3.

- **Online data structures.** A data structure designed specifically to handle continuous data-flow is typically referred to as an *online data structure* [FW98]. Continuous queries by nature suggest the use of online data structures for query processing.

- **Adaptivity.** We expect continuous queries and the data streams on which they operate to be *long-running*. Unlike during the processing of a simple one-time query, during the lifetime of a continuous query parameters such as the amount of available memory, stream data characteristics, and stream flow rates may vary considerably. While adaptive query processing techniques for more traditional queries have attracted interest recently (see Section 2 for a discussion), the work so far that we are aware of has not considered all of the parameters or kinds of adaptivity (e.g., changing approximations) that arise in a data stream context.

Distilling the basic problems and techniques above, we see that processing continuous queries over data streams entails making fundamental tradeoffs among *efficiency*, *accuracy*, and *storage*. References [AMS96, HRR98] provide some initial contributions from the theory community along these lines, but it is an open problem to understand the implications of these tradeoffs in a real system processing continuous queries for one or more real applications.

Next we will consider in more detail several specific research challenges. We will start in Section 5.2 by briefly discussing the issue of languages for specifying continuous queries. Then in Section 5.3 we focus on query evaluation and optimization, including execution plans and operators for continuous queries. We briefly address research problems associated with multiple continuous queries in Section 5.4.

5.2 Languages for Continuous Queries

Although we certainly do not advocate inventing a new query language for the purpose of specifying continuous queries over data streams—particularly over streams of relational tuples—there are some issues that must be considered. Let us take SQL as an example. Most previous work on continuous queries has restricted the language being considered in order to guarantee certain properties such as bounding the size of *Scratch* (or eliminating it entirely), or ensuring that all query results can be sent to *Stream* and none to *Store*. It appears to be an open problem to determine for arbitrary SQL queries whether these kinds of properties are satisfied, particularly if we accept the use of *Scratch* and *Store* but want to make sure they are bounded in some way. We also believe that for certain applications continuous queries will need to refer to the sequencing aspect of streams. Here SQL with extensions for *ordered relations* [SLR94], or with built-in *time-series* support [FRM94], might be a reasonable choice.

5.3 Query Evaluation and Optimization

In any database system it is the job of the query optimizer to choose in advance the "best" query plan for executing each query, based on a variety of statistics maintained for this purpose. A continuous query processor also should select a "best" execution plan, although we expect that fewer of the decisions will be made in advance due to the long-running nature of continuous queries discussed in Section 5.1. Techniques such as *eddies* [AH00], which construct and adapt query plans on-the-fly, come the closest that we know of to the query execution style we envision. However, that work is still designed for one-time rather than continuous queries, the query execution strategies do not adapt to all relevant parameters in the data stream context, and the notion of adaptivity is geared solely towards online processing.

Let us assume a standard *pipelined* (or *iterator-based*) approach to query processing [Gra93]. One of the fundamental differences between traditional query plans operating over stored relations and plans operating over data streams can be characterized as "push" versus "pull." Specifically, a traditional query plan usually has a tree shape and is executed top-down in a "pull" style: each query operator polls its children for the required input, ultimately accessing stored indexes or relations at the leaves of the query tree. Parallel query plans relax this paradigm to some extent [Gra90], but usually do not use the fully "push-based" model that data streams may demand. In an execution plan for a continuous query over data streams, we expect that it will be the appearance of a new tuple in a relevant stream that sets the plan into action. Of course this idea is not new, but rather a query processing variant on triggers, alerts, and other "active" constructs in databases [WC96].

"Push" versus "pull" aside, let us consider other changes that may be required to adapt traditional query plan operators to the data stream context. We will first consider true pipelined operators (such as selections and joins), then we will consider *blocking* operators (such as aggregation and sorting). Finally we will consider a

new class of operators that may be useful for continuous queries over data streams.

5.3.1 Pipelined operators

The simplest standard pipelined operators, such as selections, can be translated to the data stream context with little modification. However, as soon as we introduce joins we are faced with a choice. We can either: (i) evaluate portions of the query multiple times as in a nested-loop style join, which we assume is undesirable or even impossible in the data stream context; or (ii) use *Scratch* to hold temporary results during query processing, as in a pipelined hash join [WA91].

The case of joins points out that when processing continuous queries over data streams, we not only want our query operators to be pipelined, we also want them to operate with bounded intermediate storage (even in the presence of unbounded streams). For example, we might modify a pipelined join operator to degrade gracefully to an approximate join when the required storage begins to reach limits. Semantic constraints in the spirit of *data expiration* [GMLY98], or online feedback across operators in the spirit of *ripple joins* [HH99], could be applied to compute approximations with minimal loss of information.

As it turns out, the architecture we introduced in Section 4.2 for continuous queries as a whole also applies nicely to individual query plan operators: *Store* and *Scratch* represent the intermediate storage required by an operator, while *Stream* represents the pipelined operator results. Thus, techniques developed at the query level for summarization, approximation, or for moving data from *Scratch* or *Store* to *Stream* or *Throw*, might be applicable recursively to query plan operators. It is important to bear in mind, however, that *Scratch* and *Store* will generally be bounded globally, not on a per-operator basis.

5.3.2 Blocking Operators

A *blocking* operator is one that must obtain its entire input set before it can produce any output—typical examples are sorting and aggregation. In a conventional pipelined query plan, all operators that follow a blocking operator must wait until the operator obtains its entire input and begins producing its results. Obviously blocking operators cannot behave in their conventional fashion in the presence of continuous data streams, since the input is unbounded and the operator would block "forever." Part of the solution to this problem must be based on semantic considerations such as those discussed in Section 4.1—e.g., what is the result of an aggregation or a sort now when more data may be coming later? In addition to techniques such as online aggregation [HHW97, HH99], there has been some work addressing closely-related problems [LPT99, STD+00] that develops techniques based on incremental view maintenance. Developing similar techniques for continuous queries over data streams, and even more fundamentally understanding the semantics implied by the various techniques, remains an open problem.

5.3.3 Synopsis Operators

We discussed the requirement for summaries or synopses in Section 5.1 and cited some of the most relevant work in Section 2. One approach to incorporating synopsis data structures into a database system is to encapsulate them as basic operators that may appear in query plans. In support of this approach, reference [GM99] shows that different classes of queries are supported efficiently by different synopsis data structures. Thus, the query optimizer could be charged with choosing the best synopsis operator for each purpose under current conditions.

Taking this idea one step further, synopsis query operators could provide the capability to "tune" certain parameters within the operator, such as accuracy and confidence of approximation (e.g., probabilistic confidence bounds for aggregates [HHW97]), and maximum storage required (e.g., a random sample of size N). Particularly relevant in this context are the *semantic synopsis structures* proposed in [BGR01], which summarize a massive disk-resident relation based on error tolerance parameters provided independently for each attribute. If we provide synopsis operators with these types of parameters, then approximate query plans can be constructed carefully based on the query structure and available storage. Of course this power also poses significant challenges for the query optimizer.

5.4 Multiple Continuous Queries

In the paper so far we have assumed a single continuous query over multiple data streams. Let us now consider the more realistic scenario where an application registers multiple continuous queries simultaneously, probably over shared data streams. Because continuous queries are long-running, and some applications may involve a very large number of continuous queries, we expect that some form of *multi-query optimization* [Fin82, Sel88, CDTW00] will be a relevant and perhaps essential technique. There has been some recent work on optimizing multiple continuous queries, focusing either on very large numbers of queries where each query performs element-at-a-time processing [AF00, NACP01], or on subquery merging in the XML context [CDTW00]. In terms of our architecture, the queries in these systems are limited enough that they

always have empty or bounded *Store* and *Scratch* components.

Research yet to be performed includes extending the techniques from [AF00, NACP01, CDTW00] to handle more complex queries, coupling multi-query optimization techniques with approximate query answering, and optimizing the use of bounded-size *Scratch* and *Store* when they are shared among many continuous queries. More generally, the overall problem of understanding and implementing the tradeoffs among efficiency, accuracy, and storage becomes at least one step more complex in the presence of multiple continuous queries.

6 A Data Stream Management System

Our ultimate goal is to build a complete *data stream management system* (*DSMS*), with functionality and performance similar to that of a traditional DBMS, but which allows some or all of the data being managed to come in the form of continuous, possibly very rapid, data streams. In such a system, traditional one-time queries are replaced or augmented with continuous queries, and techniques such as synopsis and online data structures, approximate results, and adaptive query processing become fundamental features of the system. Other aspects of a complete DBMS also need to be reconsidered, including storage management, transaction management, user and application interfaces, and authorization.

Obviously building a complete DSMS—even a research prototype—entails a significant effort. One approach would be to modify or extend an existing DBMS to include the functionality that we envision. However, our approach will be to build a complete DSMS from scratch, so we can fully explore the issues under our own control. We have described many novel and interesting research problems that we expect to encounter along the way.

7 Conclusions and Research Plan

Many recent applications need to process continuous data streams in addition to or instead of conventional stored data sets. In this paper we have specified a general and flexible architecture for processing *continuous queries* in the presence of data streams. We have used our basic architecture as a tool to clarify alternative semantics and processing techniques for continuous queries, as well as to relate past and current work to the general *Data Stream Management System* (*DSMS*) we envision. We have mapped out a number of research topics in the area of query processing over data streams, including new requirements for online, approximate, and adaptive query processing.

At Stanford we have begun to build a complete prototype DSMS called *STREAM* (*STanford stREam datA Manager*). We are focusing initially on:

- A flexible interface for reading and storing data streams—or stream synopses—as part of a hierarchical storage manager.
- A processor for continuous queries specified using SQL or relational algebra including aggregation.
- A client Application Programming Interface (API) for registering continuous queries and receiving query results.

We expect that the development of our prototype system, as well as continuous detailed evaluation of potential applications such as the network monitoring system described in Section 3, will lead to further algorithmic and system research issues. Please visit `http://www-db.stanford.edu/stream`.

Acknowledgements

We are grateful to Jose Blakeley for excellent comments on an initial draft, and to the entire STREAM group at Stanford for many inspiring discussions.

References

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of XML documents for selective dissemination of information. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 53–64, September 2000.

[AGP00] S. Acharya, P. B. Gibbons, and V. Poosala. Congressional samples for approximate answering of group-by queries. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 487–498, May 2000.

[AGPR99] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, June 1999.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[AMS96] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. In *Proc. of the 1996 Annual ACM Symp. on Theory of Computing*, pages 20–29, May 1996.

[B+97] D. Barbara et al. The New Jersey data reduction report. *IEEE Data Engineering Bulletin*, 20(4):3–45, 1997.

[Bar99] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, December 1999.

[BCL89] J. A. Blakeley, N. Coburn, and P. A. Larson. Updating derived relations: Detecting irrelevant and autonomously computable updates. *ACM Trans. on Database Systems*, 14(3):369–400, 1989.

[BGR01] S. Babu, M. N. Garofalakis, and R. Rastogi. SPARTAN: A model-based semantic compression system for massive data tables. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 283–294, May 2001.

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

[CFPR00] C. Cortes, K. Fisher, D. Pregibon, and A. Rogers. Hancock: a language for extracting signatures from data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 9–17, August 2000.

[CGRS00] K. Chakrabarti, M. N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 111–122, September 2000.

[CMN99] S. Chaudhuri, R. Motwani, and V. R. Narasayya. On random sampling over joins. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 1999.

[DG00] N. G. Duffield and M. Grossglauser. Trajectory sampling for direct traffic observation. In *Proc. of the 2000 ACM SIGCOMM*, pages 271–284, September 2000.

[DH00] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. of the 2000 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 71–80, August 2000.

[Fin82] S. J. Finkelstein. Common subexpression analysis in database applications. In *Proc. of the 1982 ACM SIGMOD Intl. Conf. on Management of Data*, pages 235–245, June 1982.

[FRM94] C. Faloutsos, M. Ranganathan, and Y. Manolopoulos. Fast subsequence matching in time-series databases. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 419–429, May 1994.

[FW98] A. Fiat and G. J. Woeginger. *Online Algorithms, The State of the Art*. Springer-Verlag, Berlin, 1998.

[GJM96] A. Gupta, H. V. Jagadish, and I. S. Mumick. Data integration using self-maintainable views. In *Proc. of the 1996 Intl. Conf. on Extending Database Technology*, pages 140–144, March 1996.

[GK01] M. Greenwald and S. Khanna. Space-efficient online computation of quantile summaries. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 58–66, May 2001.

[GKMS01] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: one-pass summaries for approximate aggregate queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, September 2001.

[GKS01] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 13–24, May 2001.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[GM99] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *External Memory Algorithms, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 50, 1999.

[GMLY98] H. Garcia-Molina, W. J. Labio, and J. Yang. Expiring data in a warehouse. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 500–511, August 1998.

[GMMO00] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *Proc. of the 2000 Annual Symp. on Foundations of Computer Science*, pages 359–366, November 2000.

[GMP97] P. B. Gibbons, Y. Matias, and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 466–475, August 1997.

[Gra90] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, pages 102–111, May 1990.

[Gra93] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2):73–170, 1993.

[HF+00] J. M. Hellerstein, M. J. Franklin, et al. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[HH99] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 287–298, June 1999.

[HHW97] J. M. Hellerstein, P. J. Haas, and H. Wang. Online aggregation. In *Proc. of the 1997 ACM SIGMOD Intl. Conf. on Management of Data*, pages 171–182, May 1997.

[Hid99] C. Hidber. Online association rule mining. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 145–156, June 1999.

[HRR98] M. R. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. Technical Report TR-1998-011, Compaq Systems Research Center, Palo Alto, California, May 1998.

[HSD01] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. of the 2001 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, August 2001. (To appear).

[IFF+99] Z. G. Ives, D. Florescu, M. Friedman, A. Y. Levy, and D. S. Weld. An adaptive query execution system for data

integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[IP99] Y. E. Ioannidis and V. Poosala. Histogram-based approximation of set-valued query-answers. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, pages 174–185, September 1999.

[JMS95] H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.

[KGM95] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In S. H. Son, editor, *Advances in Real-Time Systems*, pages 463–486. Prentice Hall, 1995.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):583–590, August 1999.

[MRL99] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Random sampling techniques for space efficient online computation of order statistics of large datasets. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 251–262, June 1999.

[MVW00] Y. Matias, J. S. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, pages 101–110, September 2000.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 437–448, May 2001.

[PG99] V. Poosala and V. Ganti. Fast approximate answers to aggregate queries on a data cube. In *Proc. of the 1999 Intl. Conf. on Scientific and Statistical Database Management*, pages 24–33, July 1999.

[QGMW96] D. Quass, A. Gupta, I. S. Mumick, and J. Widom. Making views self-maintainable for data warehousing. In *Proc. of the 1996 Intl. Conf. on Parallel and Distributed Information Systems*, pages 158–169, December 1996.

[Sel88] T. K. Sellis. Multiple-query optimization. *ACM Trans. on Database Systems*, 13(1):23–52, 1988.

[SLR94] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query processing. In *Proc. of the 1994 ACM SIGMOD Intl. Conf. on Management of Data*, pages 430–441, May 1994.

[SPAM91] U. Schreier, H. Pirahesh, R. Agrawal, and C. Mohan. Alert: An architecture for transforming a passive DBMS into an active DBMS. In *Proc. of the 1991 Intl. Conf. on Very Large Data Bases*, pages 469–478, September 1991.

[STD+00] J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton, and D. Maier. Architecting a network query engine for producing partial results. In *Proc. of the 2000 Intl. Workshop on the Web and Databases*, pages 17–22, May 2000.

[Sul96] M. Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, page 594, September 1996.

[Tan96] A. S. Tanenbaum. *Computer Networks*. Prentice Hall, Upper Saddle River, New Jersey, 1996.

[Tea99] Times-Ten Team. In-memory data management for consumer transactions: The Times-Ten approach. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 528–529, June 1999.

[TGNO92] D. B. Terry, D. Goldberg, D. Nichols, and B. M. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[Tra] Traderbot home page. http://www.traderbot.com.

[UF01] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, September 2001. (To appear).

[UW97] J.D. Ullman and J. Widom. *A First Course in Database Systems*. Prentice Hall, Upper Saddle River, New Jersey, 1997.

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

[VW99] J. S. Vitter and M. Wang. Approximate computation of multidimensional aggregates of sparse data using wavelets. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 193–204, June 1999.

[WA91] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the 1991 Intl. Conf. on Parallel and Distributed Information Systems*, pages 68–77, December 1991.

[WC96] J. Widom and S. Ceri. *Active Database Systems: Triggers and Rules for Advanced Database Processing*. Morgan Kaufmann, San Francisco, California, 1996.

[XPA99] XML path language (XPath) version 1.0, November 1999. W3C Recommendation available at http://www.w3.org/TR/xpath.

[YSJ+00] B. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, and A. Biliris. Online data mining for co-evolving time sequences. In *Proc. of the 2000 Intl. Conf. on Data Engineering*, pages 13–22, March 2000.