Data Stream Management Issues – A Survey*

Lukasz Golab   M. Tamer Özsu

School of Computer Science
University of Waterloo
Waterloo, Canada
{lgolab, tozsu}@uwaterloo.ca

Technical Report CS-2003-08
April 2003

University of
Waterloo

*This research is partially supported by the Natural Sciences and Engineering Research Council (NSERC) of Canada.

Abstract

Traditional databases store sets of relatively static records with no pre-defined notion of time, unless timestamp attributes are explicitly added. While this model adequately represents commercial catalogues or repositories of personal information, many current and emerging applications require support for on-line analysis of rapidly changing data streams. Limitations of traditional DBMSs in supporting streaming applications have been recognized, prompting research to augment existing technologies and build new systems to manage streaming data. The purpose of this paper is to review recent work in data stream management systems, with an emphasis on data models, continuous query languages, and query evaluation and optimization techniques. We also give examples of streaming queries in various applications and review related work in modeling lists and sequences.

1 Introduction

Traditional databases have been used in applications that require persistent data storage and complex querying. Usually, a database consists of a set of unordered objects that are relatively static, with insertions, updates and deletions occurring less frequently than queries. Queries are executed when posed and the answer reflects the current state of the database. However, the past few years have witnessed an emergence of applications that do not fit this data model and querying paradigm. Instead, information naturally occurs in the form of a sequence (stream) of data values; examples include sensor data [13, 68], Internet traffic [45, 87], financial tickers [22, 104], on-line auctions [5], and transaction logs such as Web usage logs and telephone call records [26].

A data stream is a real-time, continuous, ordered (implicitly by arrival time or explicitly by timestamp) sequence of items. It is impossible to control the order in which items arrive, nor is it feasible to locally store a stream in its entirety. Likewise, queries over streams run continuously over a period of time and incrementally return new results as new data arrive. First defined in the Tapestry system [89], these are known as *long-running*, *continuous*, *standing*, and *persistent* queries [22, 66]. The unique characteristics of data streams and continuous queries dictate the following requirements of data stream management systems:

- The data model and query semantics must allow order-based and time-based operations (e.g. queries over a five-minute moving window).
- The inability to store a complete stream suggests the use of approximate summary structures, referred to in the literature as *synopses* [1] or *digests* [104]. As a result, queries over the summaries may not return exact answers.
- Streaming query plans may not use blocking operators that must consume the entire input before any results are produced.
- Due to performance and storage constraints, backtracking over a data stream is not feasible. On-line stream algorithms are restricted to making only one pass over the data.
- Applications that monitor streams in real-time must react quickly to unusual data values.
- Long-running queries may encounter changes in system conditions throughout their execution lifetimes (e.g. variable stream rates).
- Shared execution of many continuous queries is needed to ensure scalability.

Proposed data stream systems resemble the abstract architecture shown in Figure 1. An input monitor regulates the input rates, perhaps by dropping packets if the system is unable to keep up. Data are typically stored in three partitions: temporary working storage (e.g. for window queries), summary storage for stream synopses, and static storage for meta-data (e.g. physical location of each source). Long-running queries are registered in the query repository and placed into groups for shared processing, though one-time queries over the current state of the stream may also be posed. The query processor communicates with the input monitor and may re-optimize the query plans in response to changing input rates. Results are streamed to the users or temporarily buffered. Users may then refine their queries based on the latest results.

In this paper, we review recent work in data stream processing, including data models, query languages, continuous query processing, and query optimization. Related surveys include Babcock et al. [8], which discusses issues in data stream processing in the context of the STREAM project, and a tutorial by Garofalakis et al. [40], which reviews algorithms for data streams.

The remainder of this paper surveys requirements of streaming applications (Section 2), models and query languages for data streams (Section 3), streaming operators (Section 4), query processing and optimization (Section 5), and related data models and query languages

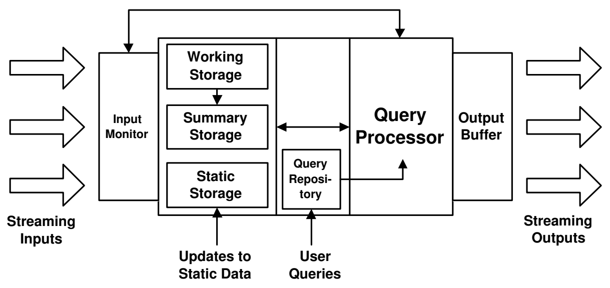

Figure 1: Abstract reference architecture for a data stream management system.

(Section 6). We conclude in Section 7 with a list of current academic projects related to data stream management.

2 Streaming Applications

We begin by reviewing a collection of current and proposed data stream applications in order to define a set of query types that a data stream management system should support. More examples may be found in the Stream Query Repository [85] and in NEXMark [93]—a proposed benchmark for data stream systems.

2.1 Sensor Networks

Sensor networks may be used for geophysical monitoring, highway congestion monitoring, movement tracking, medical monitoring of life signs, and supervision of manufacturing processes. These applications involve complex filtering and activation of an alarm upon discovering unusual patterns in the data. Aggregation and joins over multiple streams are required to analyze data from many sources, while aggregation over a single stream may be needed to compensate for individual sensor failures (due to physical damage or depletion of battery power). Sensor data mining may require access to some historical data. Representative queries include the following:

- Activate a trigger if several sensors in the same area report measurements that exceed a given threshold.
- *Drawing temperature contours on a weather map:* Perform a join of temperature streams (on the temperature attribute) produced by weather monitoring stations. Join the results with a static table containing the latitude and longitude of each station, and connect all points that have reported the same temperature with lines.
- Analyze a stream of recent power usage statistics reported to a power station (group by location, e.g. city block) and adjust the power generation rate if necessary [23].

2.2 Network Traffic Analysis

Ad-hoc systems for analyzing Internet traffic in near-real time are already in use, e.g. [27, 45, 87]. As in sensor networks, joining data from multiple sources, packet monitoring, packet filtering, and detecting unusual conditions (e.g. congestion or denial of service) are required. Support for historical queries and on-line mining is also needed, perhaps to compare current traffic traces with stored patterns corresponding to known events such as a denial-of-service attack. Other requirements include monitoring recent requests for popular URLs or finding those customers who consume the most bandwidth. These are particularly important as Internet traffic patterns are believed to obey the Power Law distribution, whose consequence is that a considerable amount of bandwidth is consumed by a small set of "heavy" users. The following are typical queries in network traffic analysis:

- *Traffic matrices*: Determine the total amount of bandwidth used by each source-destination pair and group by distinct IP address, subnet mask, and protocol type. Note that IP traffic is statistically multiplexed, therefore a traffic stream must be logically demultiplexed in order to reconstruct the underlying TCP/IP sessions [27]. Moreover, dividing the stream into sessions involves temporal semantics, e.g. a session ends if the two parties have not sent packets to each other for more than one minute.
- Compare the number of distinct source-destination pairs in the (logical) streams containing the second and third steps, respectively, of the three-way TCP handshake. If the counts differ by a large margin, then a denial-of-service attack may be taking place and permissions to connect are not being acknowledged by the (spoofed) clients.

2.3 Financial Tickers

On-line analysis of stock prices involves discovering correlations, identifying trends and arbitrage opportunities, and forecasting future values [104]. Traderbot, a typical Web-based financial ticker, allows users to pose queries such as these [90]:

- *High Volatility with Recent Volume Surge*: Find all stocks priced between \$20 and \$200, where the spread between the high tick and the low tick over the past 30 minutes is greater than three percent of the last price, and where in the last five minutes the average volume has surged by more than 300%.
- *NASDAQ Large Cap Gainers*: Find all NASDAQ stocks trading above their 200-day moving average with a market cap greater than \$5 Billion that have gained in price today between two and ten percent since the opening, and are within two percent of today's high.
- *Trading Near 52-week High on Higher Volume*: Find all stocks whose prices are within two percent of their respective 52-week highs that trade at least one million shares per day.

2.4 Transaction Log Analysis

On-line mining of Web usage logs, telephone call records, and Automated Bank Machine transactions also conform to the data stream model. The goal is to find interesting customer behaviour patterns, identify suspicious spending behaviour that could indicate fraud, and forecast future data values. As in other streaming applications, this requires joining multiple streams, complex filtering, and statistical analysis. The following are some examples:

- Find all Web pages on a particular server that have been accessed in the last fifteen minutes with a rate that is at least 40% greater than the running daily average.
- Examine Web server logs in real-time and re-route users to backup servers if the primary servers are overloaded.
- *Roaming diameter* [26]: Mine cellular phone records and for each customer, determine the greatest number of distinct base stations used during one telephone call.

2.5 Analysis of Requirements

The preceding examples show significant similarities in data models and basic operations across applications. There are some differences, but these are related to workload characteristics (e.g. stream arrival rates or amount of historical data to be stored) and not to the underlying model. We list below a set of fundamental continuous query operations over streaming data, keeping in mind that new streaming applications, possibly with additional requirements, may be proposed in the future.

- *Selection*: All streaming applications require support for complex filtering.
- *Nested aggregation* [42]: Complex aggregates, including nested aggregates (e.g. comparing a minimum with a running average) are needed to compute trends in the data.
- *Multiplexing and demultiplexing*: Physical streams may need to be decomposed into a series of logical streams and conversely, logical streams may need to be fused into one physical stream (similar to group-by and union, respectively).
- *Frequent item queries*: These are also known as *top-k* or *threshold* queries, depending on the cut-off condition.
- *Stream mining*: Operations such as pattern matching, similarity searching, and forecasting are needed for on-line mining of streaming data.
- *Joins*: Support should be included for multi-stream joins and joins of streams with static meta-data.
- *Windowed queries*: All of the above query types may be constrained to return results inside a window (e.g. the last 24 hours or the last one hundred packets).

3 Data Models and Query Languages for Streams

As demonstrated above, data stream applications require support for continuous queries and order-related operators such as moving windows. In this section, we survey proposed data models and query languages for streaming applications.

3.1 Data Models

A real-time data stream is a sequence of data items that arrive in some order and may be seen only once [57]. Since items may arrive in bursts, a data stream may instead be modeled as a sequence of lists of elements [91, 92]. Individual stream items may take the form of relational tuples or instantiations of objects. In relation-based models (e.g. STREAM [75]), items are transient tuples stored in virtual relations, possibly horizontally partitioned across remote nodes. In object-based models (e.g. COUGAR [13] and Tribeca [87]), sources and item types are modeled as (hierarchical) data types with associated methods.

Stream items may arrive out of order and/or in a preprocessed form, giving rise to the following list of possible models [46]:

1. *Unordered cash register*: Items from various domains arrive in no particular order and without any preprocessing.

2. *Ordered cash register*: Individual items from various domains are not preprocessed but arrive in some known order.

3. *Unordered aggregate*: Individual items from the same domain are preprocessed and only one item per domain arrives, in no particular order.

4. *Ordered aggregate*: Individual items from the same domain are preprocessed and one item per domain arrives in some known order.

In many cases, only an excerpt of a stream is of interest at any given time, giving rise to window models, which may be classified according the the following three criteria [17, 42]:

1. *Direction of movement of the endpoints:* Two fixed endpoints define a *fixed window*, two sliding endpoints (either forward or backward, replacing old items as new items arrive) define a *sliding window*, while one fixed endpoint and one moving endpoint (forward or backward) define a *landmark window*. There are a total of nine possibilities as each of the two endpoints could be fixed, moving forward, or moving backward.

2. *Physical vs. logical:* Physical, or *time-based* windows are defined in terms of a time interval, while logical, (also known as *count-based* or *tuple-based*) windows are defined in terms of the number of tuples.

3. *Update interval:* Eager re-evaluation updates the window upon arrival of each new tuple, while batch processing (lazy re-evaluation) induces a "jumping window". If the update interval is larger than the window size, the result is a series of non-overlapping *tumbling windows* [14].

3.2 Continuous Query Semantics

Any monotonic persistent query that is incrementally updatable may be implemented as a continuous query over a traditional database. In an append-only database, all conjunctive queries are monotonic: once a tuple is added, it either satisfies the query or it does not and the satisfaction condition does not change over time. In contrast, adding negation may violate monotonicity (e.g. select from a stream of e-mail messages all those messages that have not yet received a reply)[1]. Similarly, if the database is not append-only, then no query is monotonic as updated tuples may cease to satisfy a given query.

Less restrictive semantics of monotonic and non-monotonic continuous queries over data streams have been derived by Arasu et al. in [5]. Assuming for simplicity that time is represented as a set of natural numbers and that all continuous queries are re-evaluated at each clock tick, let $A(Q,t)$ be the answer set of a continuous query Q at time t, τ be the current time, and 0 be the starting time. The answer set of a monotonic continuous query Q at time τ is:

$$A(Q,\tau) = \bigcup_{t=1}^{\tau} (A(Q,t) - A(Q,t-1)) \cup A(Q,0) \qquad (1)$$

That is, it suffices to re-evaluate the query over newly arrived items and append qualifying tuples to the result. In contrast, non-monotonic queries may need to be recomputed from scratch during every re-evaluation, giving rise to the following semantics:

$$A(Q,\tau) = \bigcup_{t=0}^{\tau} A(Q,t) \qquad (2)$$

3.3 Stream Query Languages

Three querying paradigms for streaming data have been proposed in the literature. Relation-based systems use SQL-like languages to query timestamped relations, usually with enhanced support for windows and ordering. Object-based languages also resemble SQL, but include support for streaming abstract data types (ADTs) and associated signal processing methods. Procedural systems construct queries by defining data flow through

[1]Note that it may be possible to remove negation from some queries with a suitable rewriting.

various operators. We describe the three groups of languages below and give example queries to illustrate their differences, at times using simplified syntax to improve readability. A summary is provided in Table 1.

3.3.1 Relation-Based Languages

Three proposed relation-based languages are CQL [5, 75], StreaQuel [15, 17], and AQuery [65]. CQL (Continuous Query Language) is used in the STREAM system, and includes sliding windows and operators that translate relations to streams. It is possible to `PARTITION` a window on an attribute and specify the width of a window (e.g. `ROWS 100` or `RANGE 100 MINUTES`). For example, if a stream `S` contains telephone call records, the following query computes the average length of the ten most recent long-distance calls for each customer:

```
SELECT AVG(S.call_length)
FROM   S [PARTITION BY S.customer_id
       ROWS 10
       WHERE S.type = 'Long Distance']
```

Queries over entire streams may specify `[UNBOUNDED]` or `[NOW]` in the window type, with the latter being used for monotonic queries (e.g. selections) that need not consider any old items. Moreover, there are three relation-to-stream operators, which can be used to explicitly specify the query semantics (as defined in Equations (1) and (2)). Additionally, the sampling rate may be explicitly defined, e.g. ten percent, by following a reference to a stream with the statement `10 % SAMPLE`.

StreaQuel, the query language of TelegraphCQ, also provides advanced windowing capabilities. Each query definition is followed by a for-loop construct with a variable `t` that iterates over time. The loop contains a `WindowIs` statement that specifies the type and size of the window. Let `S` be a stream and let `ST` be the start time of a query. To specify a sliding window over `S` with size five that should run for fifty days, the following for-loop may be appended to the query:

```
for(t=ST; t<ST+50; t++)
   WindowIs(S, t-4, t)
```

Changing to a landmark window could be done by replacing `t-4` with some constant in the `WindowIs` statement. Changing the for-loop increment condition to `t=t+5` would cause the query to re-execute every five time units.

AQuery consists of a query algebra and an SQL-based language for ordered data. Table columns are treated as arrays, on which order-dependent operators such as next, previous (abbreviated prev), first, and last may be applied. For example, a continuous query over a stream of stock quotes that reports consecutive price differences of IBM stock may be specified as follows:

```
SELECT price - prev(price)
FROM Trades ASSUMING ORDER timestamp
WHERE company = 'IBM'
```

The clause `ASSUMING ORDER` defines the ordering field of the table. Note that performing this query in conventional sequence languages (discussed in Section 6) requires a self join of the `Trades` relation with a copy of itself that is shifted by one position.

3.3.2 Object-Based Languages

One approach to object-oriented stream modeling is to classify stream contents according to a type hierarchy. This method is used in the Tribeca network monitoring system, which implements Internet protocol layers as hierarchical data types [87]. Another possibility is to model the sources as ADTs, as in the COUGAR system for managing sensor data [13]. Each type of sensor is modeled by an ADT, whose interface consists of signal-processing methods supported by this type of sensor. The proposed query language has SQL-like syntax and also includes a `$every()` clause that indicates the query re-execution frequency; however, few details on the language are available in the published literature, so we do not include a summary of COUGAR's query language in Table 1. For a simple example, a query that runs every sixty seconds and returns temperature readings from all sensors on the third floor of a building could be specified as follows:

```
SELECT R.s.getTemperature()
FROM R
WHERE R.floor = 3 AND $every(60).
```

3.3.3 Procedural Languages

An alternative to declarative query languages is to let the user specify how the data should flow through the system. In the Aurora system [14], users construct query plans via a graphical interface by arranging boxes (corresponding to query operators) and joining them with directed arcs to specify data flow, though the system may later re-arrange, add, or remove operators in the optimization phase. Aurora includes several operators that are not explicitly defined in other languages: *map* applies

Language/ system	Motivating applications	Allowed inputs	Basic operators	Supported windows type	Supported windows base	Supported windows execution	Custom operators?
AQuery	stock quotes, network traffic analysis	sorted relations	relational, “each”, order-dependent (first, next, etc.)	fixed, landmark, sliding,	time and count	not discussed in [65]	via “each” operator
Aurora	sensor data	streams only	σ,π,$\cup$,$\bowtie$, group-by, resample, drop, map, window sort	fixed, landmark, sliding	time and count	streaming	via map operator
CQL/ STREAM	all-purpose	streams and relations	relational, relation-to-stream, sample	currently only sliding	time and count	streaming	allowed
StreaQuel/ TelegraphCQ	sensor data	streams and relations	relational	all types	time and count	streaming or periodic	allowed
Tribeca	network traffic analysis	single input stream	σ, π, group-by, union aggregates	fixed, landmark, sliding	time and count	streaming	allows custom aggregates

Table 1: Summary of existing and proposed data stream languages.

a function to each item (this operator is also defined in AQuery, where it is called “each”), *resample* interpolates values of missing items within a window, while *drop* randomly drops items if the input rate is too high.

3.3.4 Comments on Query Languages

Table 1 summarizes the proposed streaming query languages. Note that periodic execution refers to allowing the users to specify how often to refresh results. All languages (especially StreaQuel) include extensive support for windowing. In comparison with the list of fundamental query operators in Section 3.3, all required operators except top-k and pattern matching are explicitly defined in all the languages. Nevertheless, all languages allow user-defined aggregates, which should make it possible to define pattern-matching functions and extend the language to accommodate future streaming applications.

It appears that relation-based languages with additional support for windowing and sequence operators are the most popular paradigm at this time. Notably, in CQL, a (window excerpted from a) stream is a relation and relation-to-stream operators are needed to convert query output to streams, while in StreaQuel, all query inputs and outputs are streaming.

4 Implementing Streaming Operators

While proposed streaming languages may resemble standard SQL, their implementation, processing, and optimization present novel challenges. In this section, we highlight the differences between streaming operators and traditional relational operators, including non-blocking behaviour, approximations, and sliding windows. Note that simple operators such as projection and selection (that do not keep state information) may be used in streaming queries without any modifications.

4.1 Non-Blocking Operators

4.1.1 Windowing and Pipelining

Recall that some relational operators are blocking. For instance, prior to returning the next tuple, the Nested Loops Join (NLJ) may potentially scan the entire inner relation and compare each tuple therein with the current outer tuple. Some operators, such as joins [53, 94, 98, 100] and simple aggregates [55, 99], have non-blocking counterparts. For example, a pipelined symmetric hash join [100] builds hash tables on-the-fly for each of the participating relations. Hash tables are stored in main memory and when a tuple from one of the relations arrives, it is inserted into its table and the other tables are probed for matches. It is also possible to incrementally output the average of all the items seen so far by maintaining the cumulative sum and item count. When

Method	Functions	References
Counting	Order statistics, frequent items	[30, 48, 73]
Hashing	Distinct value counts, frequent items	[34, 38]
Sampling	Order statistics, distinct value counts, frequent items, testing near-sortedness	[30, 34, 37, 43, 73, 74]
Sketches	Frequency moments, distinct value counts, aggregates, histograms, frequent items	[2, 18, 25, 32, 36, 39, 42, 46, 59]
Wavelets	Aggregates	[41, 46, 49]

Table 2: Approximate data stream algorithms classified according to method of generating synopses.

a new item arrives, the item count is incremented, the new item's value is added to the sum, and an updated average is computed by dividing the sum by the count. There remains the issue of memory constraints if an operator requires too much working memory, so a windowing scheme may be needed to bound the memory requirements.

4.1.2 Exploiting Stream Constraints

Another way to unblock query operators is to exploit constraints over the input streams. Schema-level constraints include synchronization among timestamps in multiple streams, clustering (duplicates arrive contiguously), and ordering [11]. If two streams have nearly synchronized timestamps, an equi-join on the timestamp can be performed in limited memory: a *scrambling bound* B may be set such that if a tuple with timestamp τ arrives, then no tuple with timestamp greater than $\tau - B$ may arrive later [75].

Constraints at the data level may take the form of control packets inserted into a stream, called *punctuations* [91, 92]. Punctuations are constraints (encoded as data items) that specify conditions for all future items. For instance, a punctuation may arrive asserting that all the items henceforth shall have the A attribute value larger than ten. This punctuation could be used to partially unblock a group-by query on A since all the groups where $A \leq 10$ are guaranteed not to change for the remainder of the stream's lifetime, or until another punctuation arrives and specifies otherwise. Punctuations may also be used to synchronize multiple streams in that a source may send a punctuation asserting that it will not produce any tuples with timestamp smaller than τ [5]. There are several open problems concerning punctuations—given an arbitrary query, is there a punctuation that unblocks this query? If so, is there an efficient algorithm for finding this punctuation?

Function	References
Aggregates	[32, 42]
Distinct value counts	[2, 25, 38, 43]
Frequency moments	[2, 36, 39, 59]
Frequent items	[18, 30, 34, 73]
Histograms	[50, 51]
Order Statistics	[43, 48, 74]
Testing near-sortedness	[37]

Table 3: Approximate data stream algorithms classified according to function.

4.2 Streaming Algorithms

As shown above, unblocking a query operator may be accomplished by re-implementing it in an incremental form, restricting it to operate over a window, and exploiting stream constraints. However, there may be cases where an incremental version of an operator does not exist or is inefficient to evaluate, where even a sliding window is too large to fit in main memory, or where no suitable stream constraints are present. In these cases, compact stream summaries may be stored and approximate queries may be posed over the summaries. This implies a trade-off between accuracy and the amount of memory used to store the summaries. An additional restriction is that the processing time per item (amortized) should be kept small, especially if the inputs arrive at a fast rate. Table 2 classifies approximate algorithms for the infinite stream model according to the method used to summarize the stream, while Table 3 groups the algorithms according to function.

Counting methods, used mainly to compute quantiles and frequent item sets, typically store frequency counts of selected item types (perhaps chosen by sampling) along with error bounds on their true frequencies. Hashing may also be used to summarize a stream, especially when searching for frequent items—each item type may be hashed to n buckets by n distinct hash functions and may be considered a potentially frequent flow if all of its hash buckets are large. Sampling is a well known data reduction technique and may be used to compute various queries to within a known error bound. However, some queries (e.g. finding the maximum element in a stream) may not be reliably computed by sampling.

Sketches were initially proposed by Alon et al. [2] and have since then been used in various approximate algorithms. Let $f(i)$ be the number of occurrences of value i in a stream. A sketch of a data stream is created by taking the inner product of f with a vector of random values chosen from some distribution with a known expectation. Moreover, wavelet transforms (that reduce the underlying signal to a small set of coefficients) have been proposed to approximate aggregates over infinite streams.

4.3 Data Stream Mining

As is the case in traditional query operators, on-line stream mining operators must be incrementally updatable without making multiple passes over the data. Recent results in (possibly approximate) algorithms for on-line stream mining include computing stream signatures and representative trends [26], decision trees [33, 58], forecasting [103], k-medians clustering [19, 52], nearest neighbour queries [63], and regression analysis [23] A comprehensive discussion of similarity detection, pattern matching, and forecasting in sensor data mining may be found in a tutorial by Faloutsos [35].

4.4 Sliding Window Algorithms

Many infinite stream algorithms do not have obvious counterparts in the sliding window model. For instance, while computing the maximum value in an infinite stream is trivial, doing so in a sliding window of size N requires $\Omega(N)$ space—consider a sequence of non-increasing values, in which the oldest item in any given window is the maximum and must be replaced whenever the window moves forward. Thus, the fundamental problem is that as new items arrive, old items must be simultaneously evicted from the window.

4.4.1 Windowed Aggregates

Simple aggregates over sliding windows may be computed in limited memory by dividing the window into small portions (called *basic windows* in [104]) and only storing a synopsis and a timestamp for each portion. When the timestamp of the oldest basic window expires, its synopsis is removed, a fresh window is added to the front, and the aggregate is incrementally re-computed. This method has been used to compute correlations between streams [104] and to find frequently appearing items [29], but results are refreshed only after the stream fills the current basic window; if the available memory is small, the refresh interval is large. Moreover, some statistics may not be incrementally computable from a set of synopses.

One way to stream new results after each new item arrives is to maintain a windowed sample [9] and estimate the answer from the sample. Another is to bound the error caused by delayed expiration of basic windows. Datar et al. [28] show that restricting the sizes of the basic windows to powers of two and imposing a limit on the number of basic windows of each size yields a space-optimal algorithm that approximates simple aggregates to within ϵ using logarithmic space (with respect to the sliding window size). This algorithm has been used to approximately compute the sum [10, 44] as well as variance and k-medians clustering [28].

4.4.2 Windowed Joins

The symmetric hash join [100] and an analogous symmetric NLJ may be extended to operate over two [62] or more [47] sliding windows by periodically scanning the hash tables (or whole windows in case of the NLJ) and removing stale items. Interesting trade-offs appear in that large hash tables are expensive to maintain if tuple expiration is performed too frequently [47].

5 Continuous Query Processing and Optimization

Having surveyed issues in designing continuous query languages and implementing streaming operators, we now discuss problems related to processing and optimizing continuous queries. In what follows, we outline emerging research in cost metrics, query plans, quality-of-service guarantees, and distributed optimization of streaming queries.

5.1 Memory Requirements

As already discussed, some query operators (e.g. joins) may require infinite working memory, even when rewritten into an incremental form. Consequently, a possible first step in processing a continuous query is to decide whether it may be answered exactly in bounded memory or whether it should be approximated based on stream summaries. Computing the memory requirements of continuous queries has been studied by Arasu et al. [4] for monotonic conjunctive queries with grouping and aggregation. Consider two unbounded relational data streams: $S(A, B, C)$ and $T(D, E)$. The query $\pi_A(\sigma_{A=D \wedge A>10 \wedge D<20} (S \times T))$ may be evaluated in bounded memory whether or not the projection preserves duplicates. To preserve duplicates, for each integer i between 11 and 19, it suffices to maintain the count of tuples in S such that $A = i$ and the count of tuples in T such that $D = i$. To remove duplicates, it is necessary to store flags indicating which tuples have occurred such that $S.A = i$ and $T.D = i$ for $i \in [11, 19]$. Conversely, the query $\pi_A(\sigma_{A=D}(S \times T))$ is not computable in finite memory either with or without duplicates.

Interestingly, $\pi_A(\sigma_{A>10}S)$ is computable in finite memory only if duplicates are preserved; any tuple in S with $A > 10$ is added to the answer as soon as it arrives. On the other hand, the query $\pi_A(\sigma_{B<D \wedge A>10 \wedge A<20}(S \times T))$ is computable in bounded memory only if duplicates are removed: for each integer i between 11 and 19, it suffices to maintain the current minimum value of B among all the tuples in S such that $A = i$ and the current maximum value of D over all tuples in T.

5.2 Cost Metrics and Statistics

Traditional DBMSs use selectivity information and available indices to select plans that require fewest disk accesses. This cost metric, however, does not apply to (possibly approximate) continuous queries over infinite streams, where processing cost per-unit-time is more appropriate [62]. Below, we list possible cost metrics for streaming queries along with necessary statistics that must be maintained.

- *Accuracy and reporting delay vs. memory usage*: Allocating more memory for synopses should improve accuracy, while sampling and load shedding [88] decrease memory usage by increasing the error. It is necessary to know the accuracy of each operator as a function of the available memory, and how to combine such functions to obtain the overall accuracy of a plan. Furthermore, batch processing [8] may be done instead of re-evaluating a query whenever a new item arrives, at a cost of increased reporting delay.
- *Output rate*: If the stream arrival rates and output rates of query operators are known, it is possible to optimize for the highest output rate or to find a plan that takes the least time to output a given number of tuples [97]. In related work, Urhan and Franklin discuss scheduling of pipelined hash joins in order to quickly produce the initial portion of the result [95].
- *Power usage*: In a wireless network of battery-operated sensors, energy consumption may be minimized if each sensor's power consumption characteristics (when transmitting and receiving) are known [70, 101].

5.3 Continuous Query Plans

In relational DBMSs, all operators are pull-based: an operator requests data from one of its children in the plan tree only when needed. In contrast, stream operators consume data pushed to the system by the sources. One approach to reconcile these differences, as considered in Fjords [68] and STREAM [8], is to connect operators with queues, allowing sources to push data into a queue and operators to retrieve data as needed. Problems include scheduling operators so as to minimize queue sizes and queuing delays in the presence of bursty streams [7], and maintaining quality-of-service guarantees [14]. Another challenge in continuous query plans deals with supporting historical queries. Designing disk-based data structures and indices to exploit access patterns of stream archives is an open problem [15].

5.4 Processing Multiple Queries

Two approaches have been proposed to execute similar continuous queries together: sharing query plans (e.g. NiagaraCQ [22]) and indexing query predicates (CACQ [71] and PSoup [16]). In the former, queries belonging to the same group share a plan, which produces the union of the results needed by each query in the group. A final selection is then applied to the shared result set. Problems include dynamic re-grouping as new queries are added to the system [20], choosing whether to push selections below joins or pull them above (the latter allows many queries to share one materialized join, but performance may suffer if the join is expensive to maintain) [21], shared evaluation of windowed joins with various window sizes [54], and plan sharing for complex queries.

In the indexing approach, query predicates are stored in a table. When a new tuple arrives for processing, its attribute values are extracted and looked up in the query table to see which queries are satisfied by this tuple. Data and queries are treated as duals, reducing query processing to a multi-way join of the query predicate table and the data tables. The indexing approach works well for simple SPJ queries, but is currently not applicable to, e.g. windowed aggregates [16].

5.5 Query Optimization

5.5.1 Query Rewriting

A useful rewriting technique in relational databases deals with re-ordering a sequence of binary joins in order to minimize a particular cost metric. There has been some preliminary work in join ordering for data streams in the context of the rate-based model [97, 98] and in main-memory sliding window joins [47]. In general, each of the query languages outlined in Section 3 introduces rewritings for its new operators, e.g. selections and projections commute over sliding windows [5, 17].

5.5.2 Adaptivity

The cost of a query plan may change for three reasons: change in processing time of an operator, change in selectivity of a predicate, and change in the arrival rate of a stream [6]. Initial efforts on adaptive query plans include mid-query re-optimization [61] and query scrambling, where the objective was to pre-empt any operators that become blocked and schedule other operators instead [3, 96]. To further increase adaptivity, instead of maintaining a rigid tree-structured query plan, the Eddies approach (introduced in [6], extended to multi-way joins in [79], and applied to continuous queries in [16, 71]) performs scheduling of each tuple separately by routing it through the operators that make up the query plan. In effect, the query plan is dynamically re-ordered to match current system conditions. This is accomplished by tuple routing policies that attempt to discover which operators are fast and selective, and those operators are scheduled first. There is, however, an important trade-off between the resulting adaptivity and the overhead required to route each tuple separately.

5.6 Distributed Query Processing

In sensor networks, Internet traffic analysis, and Web usage logs, multiple data streams are expected to arrive from remote sources, suggesting a distribution of query operators among the participating nodes. We classify distribution strategies according to the envisioned application: some are all-purpose, while others are designed specifically for sensor networks.

All-purpose strategies aim at decreasing communication costs by performing computations at the sources. These include re-ordering of query operators across sites [24, 84] and specifically, performing simple query functions (e.g. filtering, aggregation or signal compression [64]) locally at a sensor or a network router [27, 56, 69, 72, 102]. For example, if each remote node pre-aggregates its results by sending to the co-ordinator the sum and count of its values, the co-ordinator may then take the cumulative sum and cumulative count, and compute the overall average. Other techniques include selecting leader nodes to stream pre-processed results to the front-end [102], caching [16, 31, 101], and sending updates to the co-ordinator only if new data values differ significantly from previously reported values [76].

Distributed techniques for ad-hoc sensor networks exploit the fact that query dissemination and result collection in a wireless sensor network proceed along a routing tree (or a DAG) via a shared wireless channel. [69, 72]. The two main objectives are decreasing the number of transmissions in order to extend battery life and dealing with poor wireless connectivity. For example, if a sensor reports its maximum local value x in response to a `MAX` query, a neighbouring sensor that overhears this transmission need not respond if its local maximum is smaller than x (assuming that the neighbouring sensor has not powered down). Dealing with poor connectivity includes sending redundant copies of data packets, e.g. a sensor could broadcast its maximum value to several other nodes, not just the node along the path to the root. However, this does not work for other aggregates such as `SUM` and `COUNT`, as duplicate values would contaminate the result. In these cases, a sensor may "split" its local sum and send partial sums to each of its neighbours. Even if one packet is lost, the remainder of the sum should still reach the root.

6 Related Models and Query Languages

While data stream applications have begun to appear in the last several years, there has been some prior work on modeling (off-line) lists, sequences, and time series. We discuss those next.

6.1 List-Based Models

Two types of list models have been defined in the literature: functional and object-oriented. Functional systems (e.g. Tangram [67, 77]) operate on (possibly infinite) lists by means of functional transformations called transducers, of which there are five types:

1. *Enumerators* produce new lists.
2. *Maps* apply a function to each item in a list.
3. *Filters* correspond to selection predicates.
4. *Accumulators* compute aggregates over a list or a sliding window.
5. *Pattern Detectors* consist of regular expressions on values inside a list.

Object-oriented models (e.g. AQUA [86]) define a *LIST* object to be composed of a set of *CELL*s and a set of directed edges joining the cells. Supported operations resemble transducers: *select* (filter), *apply* (map), and *sub_select* (selection of a regular-expression-like pattern within a list). There are also two novel operators: *descendants* and *ancestors*, which return the portion of a list preceding and following a match, respectively. Each operator preserves the ordering of the list.

6.2 Time Series Models

A time-series extension to SQL, called SQL-TS [80], models time series as relations sorted by timestamps and extends SQL by allowing the following constructs to be included in the `FROM` clause:

- A `CLUSTER BY` clause which effectively demultiplexes a stream into separate logical streams.
- A `SEQUENCE BY` clause which sorts the time series on the provided timestamp.
- An `AS` clause used to bind tuples to variable names. As in regular expressions, recurring patterns may be expressed with a star.

Each tuple that is bound to a variable is logically modeled to contain pointers to the previous and next tuples in the time series, forming a doubly linked list. For instance, to find the maximal periods during which the price of a stock fell more than 50 percent, the following SQL-TS query may be used:

```
SELECT X.name, X.date as START_DATE,
       Z.previous.date as END_DATE
FROM   stock_quotes
       CLUSTER BY name
       SEQUENCE BY date
       AS (X, *Y, Z)
WHERE  Y.price < Y.previous.price
       AND Z.previous.price < 0.5 * X.price
```

6.3 Sequence Models

6.3.1 Modeling and Querying Sequences

The SEQ sequence model and algebra were introduced by Seshadri et al. in [81, 82, 83]. SEQ defines an ordering function from the integers (or another ordered domain such as calendar dates) to each item in the sequence. Some operators, such as selection, projection, various set operations, and aggregation (including moving windows) are carried over from the relational model. There are five new operators:

- The *Group* construct is a group-by operator that divides a sequence into sub-sequences.
- The *Positional Offset* returns an output sequence that is identical to the input sequence except that the ordering domain has been shifted by a specified number of positions.
- The *Positional Join* joins two sequences on the ordering attribute.
- The *Collapse* operator is a many-to-one surjective function from one ordering domain to another. For instance, a sequence of daily stock quotes may be collapsed into a sequence of average (or top, or minimum) weekly stock quotes.
- The *Expand* operator is the “inverse” of Collapse. However, note that collapsing and then expanding a sequence S does not return S unless the original sequence is also stored.

Relational equivalences for projections and selections apply, as does the predicate push-down heuristic. There are also additional equivalences between sequence operators. For instance, the positional offset can be “pushed through” any operator and the projection may be pushed through any sequence operator, so long as all the attributes that participate in the sequence operator are included in the projection.

The SEQ model has been implemented in SRQL (Sorted Relational Query Language) [78]. Sequences are implemented as logically or physically sorted multi-sets (relations) and the language attempts to exploit the sort order. Four new operators have been added:

- The *Sequence* operator creates a new sequence by choosing an ordering field(s) for a particular relational table.

- The *Shiftall* operator joins a sequence R with a copy of itself, R' whose ordering field is shifted with respect to R
- The *Shift* operator is similar to *ShiftAll* but only joins a sequence R with the ordering field of R' instead of all the fields of R'.
- *WindowAggregate* computes sliding window aggregates.

These operators implement the SEQ model, with the exception of *Expand* and *Collapse*, which require temporal capabilities beyond SQL's power. For an example taken from [78], consider a sequence of Volcano eruptions with schema `V(time, name)` and a sequence of earthquakes with schema `E(time, name, magnitude)`. Suppose that the sequences are instantiated as seen in Table 4. To find for each volcano eruption the most recent earthquake that was greater than 7 on the Richter scale, the following SRQL query may be posed:

```
SELECT V.name, E.name
FROM Volcano as V, Earthquake as E
WHERE E.time <= V.time
AND (SHIFT(E,1).time > V.time
     OR SHIFT(E,1).time IS NULL)
AND E.magnitude > 7
```

Volcanoes		Earthquakes		
time	name	time	name	magnitude
3	v1	1	e1	8
4	v2	2	e2	2
5	v3	5	e3	8
8	v4	6	e4	9
9	v5	7	e5	8

Table 4: Volcano and Earthquake sequences.

That is, the earthquake times are shifted by one position (e.g. the time of $e1$ becomes 2 and the time of $e2$ becomes 5 and so on until $e5$, whose time becomes NULL) and this shifted time must be more recent than the time of a volcano eruption to guarantee the most recent earthquake. The result of this query is shown in Table 5.

V.name	E.name
v3	e3
v4	e5
v5	e5

Table 5: Result of the earthquake query.

6.3.2 Materialized Views over Sequences

The Chronicle data model includes relations and sequencing attributes, and deals with maintaining materialized views over sequences [60]. It is shown that views are incrementally updatable (i.e. updates take time proportional to the size of the view, not to the length of the underlying sequence) if sequence items arrive in increasing order of the sequence numbers, and if the view definition algebra does not include joins of two sequences, unless these are equi-joins on the sequencing attribute with one of the sequencing fields projected out. Moreover, the algebra must not have a group-by operator with an aggregate that may not be computed incrementally. The Chronicle model does not consider sliding windows.

7 Conclusions

We have shown that designing an effective data stream management system requires extensive modifications of nearly every part of a traditional database, creating many interesting database problems such as adding time, order, and windowing to data models and query languages, implementing approximate operators, combining push-based and pull-based operators in query plans, adaptive query re-optimization, and distributed query processing. Recent interest in these problems has generated a number of academic projects. There exist at least the following systems:

- Aurora [14, 24] is a workflow-oriented system that allows users to build query plans by arranging boxes (operators) and arrows (data flow among operators). Web site: `http://www.cs.brown.edu/research/aurora`.
- COUGAR [12, 13] is a sensor database that models sensors as ADTs and their output as time series. Recent work in the COUGAR project deals with query processing inside the sensor network [101, 102]. Web site: `http://www.cs.cornell.edu/database/cougar`.
- Gigascope [27] is a distributed network monitoring architecture that proposes pushing some query operators to the sources (e.g. routers).
- NiagaraCQ [22] is a continuous query system designed for monitoring dynamic Web content. The

system executes multiple continuous queries (expressed in XML-QL) over streaming data in groups, where each group of similar queries shares an execution plan. Web site: `http://www.cs.wisc.edu/niagara`.

- OpenCQ [66] is another continuous query system for monitoring streaming Web content. Its focus is on scalable event-driven query processing. Web site: `http://disl.cc.gatech.edu/CQ`
- StatStream [104] is a stream monitoring system designed to compute on-line statistics across many streams. Web site: `http://cs.nyu.edu/cs/faculty/shasha/papers/statstream.html`.
- STREAM [75] is an all-purpose relation-based system with an emphasis on memory management and approximate query answering. Web site: `http://www-db.stanford.edu/stream`.
- TelegraphCQ [15] is a proposed continuous query processing system that focuses on shared query evaluation and adaptive query processing. Web site: `http://telegraph.cs.berkeley.edu`.
- Tribeca [87] is an early on-line Internet traffic monitoring tool.

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, S. Ramaswamy. Join synopses for approximate query answering. In *Proc. ACM Int. Conf. on Management of Data*, 1999, pp. 275–286.

[2] N. Alon, Y. Matias, M. Szegedy. The Space Complexity of Approximating the Frequency Moments. In *Proc. 28th ACM Symp. on Theory of Computing*, 1996, pp. 20–29.

[3] L. Amsaleg, M. J. Franklin, A. Tomasic, T. Urhan. Scrambling Query Plans to Cope with Unexpected Delays. In *Proc. Int. Conf. on Parallel and Distributed Information Systems*, 1996, pp. 208–219.

[4] A. Arasu, B. Babcock, S. Babu, J. McAlister, J. Widom. Characterizing Memory Requirements for Queries over Continuous Data Streams. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2002, pp. 221–232.

[5] A. Arasu, S. Babu, J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. Stanford University Technical Report 2002-57, November 2002. Available at `http://dbpubs.stanford.edu:8090/pub/2002-57`.

[6] R. Avnur, J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proc. ACM Int. Conf. on Management of Data*, 2000, pp. 261–272.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. To appear in *Proc. ACM Int. Conf. on Management of Data*, June 2003.

[8] B. Babcock, S. Babu, M. Datar, R. Motwani, J. Widom. Models and Issues in Data Streams. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2002, pp. 1–16.

[9] B. Babcock, M. Datar, R. Motwani. Sampling from a Moving Window over Streaming Data. In *Proc. 13th SIAM-ACM Symp. on Discrete Algorithms*, 2002, pp. 633–634.

[10] B. Babcock, M. Datar, R. Motwani, L. O'Callaghan. Maintaining Variance and k-Medians over Data Stream Windows. To appear in *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, June 2003.

[11] S. Babu, J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. Stanford University Technical Report 2002-52, November 2002. Available at `http://dbpubs.stanford.edu:8090/pub/2002-52`.

[12] P. Bonnet, J. Gehrke, P. Seshadri. Querying the Physical World. In *IEEE Personal Communications*, 7(5):10–15, Oct. 2000.

[13] P. Bonnet, J. Gehrke, P. Seshadri. Towards Sensor Database Systems. In *Proc. 2nd Int. Conf. on Mobile Data Management*, 2001, pages 3–14.

[14] D. Carney, U. Cetinternel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, S. Zdonik. Monitoring streams—A New Class of Data Management Applications. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 215–226.

[15] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, M. Shah. TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res*, 2003, pp. 269–280.

[16] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 203–214.

[17] S. Chandrasekaran, S. Krishnamurthy, S. Madden, A. Deshpande, M. J. Franklin, J. M. Hellerstein, M. Shah. Windows Explained, Windows Expressed. Submitted for publication, 2003. Available at `http://www.cs.berkeley.edu/~sirish/research/streaquel.pdf`.

[18] M. Charikar, K. Chen, M. Farach-Colton. Finding frequent items in data streams. In *Proc. 29th Int. Colloquium on Automata, Languages and Programming*, 2002, pp. 693–703.

[19] M. Charikar, L. O'Callaghan, R. Panigrahy. Better Streaming Algorithms for Clustering Problems. To appear in *Proc. 35th ACM Symp. on Theory of Computing*, June 2003.

[20] J. Chen, D. DeWitt. Dynamic Re-Grouping of Continuous Queries. Manuscript, 2002. Available at `http://www.cs.wisc.edu/niagara/Publications.html`.

[21] J. Chen, D. J. DeWitt, J. F. Naughton. Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries. In *Proc. 18th Int. Conf. on Data Engineering*, 2002, pp. 345–357.

[22] J. Chen, D. DeWitt, F. Tian, Y. Wang. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *Proc. ACM Int. Conf. on Management of Data*, 2000, pp. 379–390.

[23] Y. Chen, G. Dong, J. Han, B. W.Wah, J. Wang. Multi-Dimensional Regression Analysis of Time-Series Data Streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 323–334.

[24] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, S. Zdonik. Scalable Distributed Stream Processing. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res*, 2003.

[25] G. Cormode, M. Datar, P. Indyk, S. Muthukrishnan. Comparing Data Streams Using Hamming Norms (How to Zero In). In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 335–345.

[26] C. Cortes, K. Fisher, D. Pregibon, A. Rogers, F. Smith. Hancock: A Language for Extracting Signatures from Data Streams. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2000, pp. 9–17.

[27] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, O. Spatscheck. GigaScope: High Performance Network Monitoring with an SQL Interface. In *Proc. ACM Int. Conf. on Management of Data*, 2002, p. 623.

[28] M. Datar, A. Gionis, P. Indyk, R. Motwani. Maintaining Stream Statistics over Sliding Windows. In *Proc. 13th SIAM-ACM Symp. on Discrete Algorithms*, 2002, pp. 635–644

[29] D. DeHaan, E. D. Demaine, L. Golab, A. Lopez-Ortiz, J. I. Munro. Towards Identifying Frequent Items in Sliding Windows. University of Waterloo Technical Report CS-2003-06, March 2003. Available at `http://db.uwaterloo.ca/~lgolab/frequent.pdf`.

[30] E. Demaine, A. Lopez-Ortiz, J. I. Munro. Frequency Estimation of Internet Packet Streams with Limited Space. In *Proc. European Symposium on Algorithms*, 2002, pp. 348–360.

[31] A. Deshpande, S. Nath, P. Gibbons, S. Seshan. Cache-and-Query for Wide Area Sensor Databases. To appear in *Proc. ACM Int. Conf. on Management of Data*, June 2003.

[32] A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi. Processing Complex Aggregate Queries over Data Streams. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pp. 61–72.

[33] P. Domingos, G. Hulten. Mining High-Speed Data Streams. In *Proc. 6th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2000, pp. 71–80.

[34] C. Estan, G. Varghese. New Directions in Traffic Measurement and Accounting. In *Proc. ACM SIGCOMM Internet Measurement Workshop*, 2001, pp. 75–80.

[35] C. Faloutsos. Sensor Data Mining: Similarity Search and Pattern Analysis. Tutorial in *28th Int. Conf. on Very Large Data Bases*, 2002.

[36] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan. An Approximate L1-Difference Algorithm for Massive Data Streams. In *Proc. 40th Symp. on Foundations of Computer Science*, 1999. pp. 501–511.

[37] J. Feigenbaum, S. Kannan, M. Strauss, M. Viswanathan. Testing and Spot Checking Data Streams. . In *Proc. 11th SIAM-ACM Symp. on Discrete Algorithms*, 2000, pp. 165–174.

[38] P. Flajolet, G. N. Martin. Probabilistic Counting. In *24th Annual Symp. on Foundations of Computer Science*, 1983, pp. 76–82, 1983.

[39] J. Fong, M. Strauss. An Approximate Lp-difference Algorithm for Massive Data Streams. In *Journal of Discrete Mathematics and Theoretical Computer Science*, 4(2):301–322, 2001.

[40] M. Garofalakis, J. Gehrke, R. Rastogi. Querying and Mining Data Streams: You Only Get One Look. Tutorial in *ACM Int. Conf. on Management of Data*, 2002.

[41] M. Garofalakis, P. Gibbons. Wavelet Synopses with Error Guarantees. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pp. 476–487.

[42] J. Gehrke, F. Korn, D. Srivastava. On Computing Correlated Aggregates Over Continual Data Streams. In *Proc. ACM Int. Conf. on Management of Data*, 2001, pp. 13–24.

[43] P. Gibbons, S. Tirthapura. Estimating Simple Functions on the Union of Data Streams. In *Proc. ACM Symp. on Parallel Algorithms an Architectures*, 2001, pp. 281–291.

[44] P. Gibbons, S. Tirthapura. Distributed Streams Algorithms for Sliding Windows. In *14th Annual ACM Symp. on Parallel Algorithms and Architectures*, 2002, pp. 63–72.

[45] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. J. Strauss. QuickSAND: Quick Summary and Analysis of Network Data. DIMACS Technical Report 2001-43, Dec. 2001. Available at `http://citeseer.nj.nec.com/gilbert01quicksand.html`.

[46] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. J. Strauss. Surfing Wavelets on Streams: One-Pass Summaries for Approximate Aggregate Queries. In *Proc. 27th Int. Conf. on Very Large Data Bases*, 2001, pp. 79–88.

[47] L. Golab, M. T. Özsu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. University of Waterloo Technical Report CS-2003-01, Feb. 2003. Available at `http://db.uwaterloo.ca/~ddbms/publications/stream/multijoins.pdf`.

[48] J. Greenwald, F. Khanna. Space Efficient On-Line Computation of Quantile Summaries. In *Proc. ACM Int. Conf. on Management of Data*, 2001, pp. 58–66.

[49] S. Guha, P. Indyk, S. Muthukrishnan, M. Strauss. Histogramming Data Streams with Fast Per-Item Processing. In *Proc. 29th Int. Colloquium on Automata, Languages and Programming*, 2002, pp. 681–692.

[50] S. Guha, N. Koudas. Approximating a Data Stream for Querying and Estimation: Algorithms and Performance Evaluation. In *Proc. 18th Int. Conf. on Data Engineering*, 2002, pp. 567–576.

[51] S. Guha, N. Koudas, K. Shim. Data-Streams and Histograms. In *Proc. 33rd Annual ACM Symp. on Theory of Computing*, 2001, pp. 471–475

[52] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan. Clustering Data Streams. In *Proc. IEEE Symp. on Foundations of Computer Science*, pp. 359–366.

[53] P. Haas, J. Hellerstein. Ripple Joins for Online Aggregation. In *Proc. ACM Int. Conf. on Management of Data*, 1999, pp. 287-298.

[54] M. A. Hammad, M. J. Franklin, W. G. Aref, A. K. Elmagarmid. Scheduling for shared window joins over data streams. Submitted for publication, Feb. 2003.

[55] J. M. Hellerstein, P. Haas, H. Wang. Online Aggregation. In *Proc. ACM Int. Conf. on Management of Data*, 1997, pp. 171–182.

[56] J. M. Hellerstein, W. Hong, S. Madden, K. Stanek. Beyond Average: Toward Sophisticated Sensing with Queries. To appear in *Proc. 2nd Int. Workshop on Information Processing in Sensor Networks*, Apr. 2003.

[57] M. Henzinger, P. Raghavan, S. Rajagopalan. Computing on Data Streams. In *DIMACS series in Discrete Mathematics and Theoretical Computer Science*, vol. 50, pp. 107–118, 1999.

[58] G. Hulten, L. Spencer, P. Domingos. Mining Time-Changing Data Streams. In *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining*, 2001, pp. 97–106.

[59] P. Indyk. Stable distributions, pseudorandom generators, embeddings and data stream computations. In *Proc. Annual IEEE Symp. on Foundations of Computer Science*, 2000, pp. 189–197.

[60] H. V. Jagadish, I. S. Mumick, A. Silberschatz. View Maintenance Issues for the Chronicle Data Model. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 1995, pp. 113–124.

[61] N. Kabra, D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *Proc. ACM Int. Conf. on Management of Data*, 1998, pp. 106–117.

[62] J. Kang, J. Naughton, S. Viglas. Evaluating Window Joins over Unbounded Streams. To appear in *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[63] F. Korn, S. Muthukrishnan, D. Srivastava. Reverse Nearest Neighbor Aggregates over Data Streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 814–825.

[64] I. Lazaridis, S. Mehrotra. Capturing Sensor-Generated Time Series with Quality Guarantees. To appear in *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[65] A. Lerner, D. Shasha. AQuery: Query Language for Ordered Data, Optimization Techniques, and Experiments. TR2003-836, Courant Institute of Mathematical Sciences, New York University, March 2003. Available at `http://csdocs.cs.nyu.edu/Dienst/Repository/2.0/Body/ncstrl.nyu_cs%2fTR2003-836/pdf`.

[66] L. Liu, C. Pu, W. Tang. Continual Queries for Internet-Scale Event-Driven Information Delivery. In *IEEE Trans. Knowledge and Data Eng.*, 11(4): 610–628, 1999.

[67] B. Livezey, R. R. Muntz. ASPEN: A Stream Processing Environment. In *Proc. PARLE (2)*, 1989, pp. 374–388.

[68] S. Madden, M. J. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. 18th Int. Conf. on Data Engineering*, 2002, pp. 555–566.

[69] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proc. 5th Annual Symp. on Operating Systems Design and Implementation*, 2002.

[70] S. Madden, M. J. Franklin, J. M. Hellerstein, W. Hong. The Design of an Acquisitional Query Processor For Sensor Networks. To appear in *Proc. ACM Int. Conf. on Management of Data*, June 2003.

[71] S. Madden, M. Shah, J. Hellerstein, V. Raman. Continuously Adaptive Continuous Queries Over Streams. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pp. 49–60.

[72] S. Madden, R. Szewczyk, M. J. Franklin, D. Culler. Supporting Aggregate Queries Over Ad-Hoc Wireless Sensor Networks. In *Proc. 4th IEEE Workshop on Mobile Computing Systems and Applications*, 2002, pp. 49–58.

[73] G. S. Manku, R. Motwani. Approximate Frequency Counts over Data Streams. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 346–357.

[74] G.S. Manku, S. Rajagopalan, B.G. Lindsay. Random Sampling Techniques for Space Efficient Online Computation of Order Statistics of Large Datasets. In *Proc. ACM Int. Conf. on Management of Data*, 1999, pp. 251–262.

[75] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, R. Varma. Query Processing, Approximation, and Resource Management in a Data Stream Management System. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res*, 2003, pp. 245–256.

[76] C. Olston, J. Jiang, J. Widom. Adaptive Filters for Continuous Queries over Distributed Data Streams. To appear in *Proc. ACM Int. Conf. on Management of Data*, June 2003.

[77] D.S. Parker. Stream Data Analysis in Prolog. In *L. Sterling, ed., The Practice of Prolog*, Cambridge, MA: MIT Press, 1990.

[78] R. Ramakrishnan, D. Donjerkovic, A. Ranganathan, K. S. Beyer. SRQL: Sorted Relational Query Language. In *Proc. 10th Int. Conf. on Scientific and Statistical Database Management*, 1998, pp. 84–97.

[79] V. Raman, A. Deshpande, J. Hellerstein. Using State Modules for Adaptive Query Processing. To appear in *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[80] R. Sadri, C. Zaniolo, A. M. Zarkesh, J. Adibi. Optimization of Sequence Queries in Database Systems. In *Proc. ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, 2001, pp. 71–81.

[81] P. Seshadri, M. Livny, R. Ramakrishnan. Sequence Query Processing. In *Proc. ACM Int. Conf. on Management of Data*, 1994, pp. 430–441.

[82] P. Seshadri, M. Livny, R. Ramakrishnan. SEQ: A Model for sequence Databases. In *Proc. 11th Int. Conf. on Data Engineering*, 1995, pp. 232–239.

[83] P. Seshadri, M. Livny, R. Ramakrishnan. The Design and Implementation of a Sequence Database System. In *Proc. 22nd Int. Conf. on Very Large Data Bases*, 1996, pp. 99–110.

[84] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, M. J. Franklin. Flux: An Adaptive Partitioning Operator for Continuous Query Systems. To appear in *Proc. 19th Int. Conf. on Data Engineering*, 2003.

[85] Stream Query Repository. `www-db.stanford.edu/stream/sqr`.

[86] B. Subramanian, T. W. Leung, S. L. Vandenberg, S. B. Zdonik. The AQUA Approach to Querying Lists and Trees in Object-Oriented Databases. In *Proc. 11th Int. Conf. on Data Engineering*, 1995, pp. 80–89.

[87] M. Sullivan, A. Heybey. Tribeca: A System for Managing Large Databases of Network Traffic. In *Proc. USENIX Annual Technical Conf.* , 1998.

[88] N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, M. Stonebraker. Load Shedding in a Data Stream Manager. Technical Report, Brown University, Feb. 2003. Available at `http://www.cs.brown.edu/~tatbul/papers/tatbul_tr.pdf`.

[89] D. Terry, D. Goldberg, D. Nichols, B. Oki. Continuous Queries over Append-Only Databases. In *Proc. ACM Int. Conf. on Management of Data*, 1992, pp. 321–330.

[90] Traderbot. `www.traderbot.com`.

[91] P. Tucker, D. Maier, T. Sheard, L. Fegaras. Punctuating Continuous Data Streams. Technical Report, Oregon University, 2001. Available at `http://www.cs.brown.edu/courses/cs227/papers/TM02-punctuating.pdf`.

[92] P. Tucker, D. Maier, T. Sheard, L. Fegaras. Enhancing relational operators for querying over punctuated data streams. Manuscript, 2002. Available at `http://www.cse.ogi.edu/dot/niagara/pstream/punctuating.pdf`.

[93] P. Tucker, T. Tufte, V. Papadimos, D. Maier. NEXMark—a Benchmark for Querying Data Streams. Manuscript, 2002. Available at `http://www.cse.ogi.edu/dot/niagara/pstream/nexmark.pdf`.

[94] T. Urhan, M. J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. In *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[95] T. Urhan, M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. In *Proc. 27th Int. Conf. on Very Large Data Bases*, 2001, pp. 501–510.

[96] T. Urhan, M. J. Franklin, L. Amsaleg. Cost-based Query Scrambling for Initial Delays. In *Proc. ACM Int. Conf. on Management of Data*, 1998, pp. 130–141.

[97] S. Viglas and J. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proc. ACM Int. Conf. on Management of Data*, 2002, pp. 37–48.

[98] S. Viglas, J. Naughton, J. Burger. Maximizing the Output Rate of Multi-Join Queries over Streaming Information Sources. Available at `http://www.cs.wisc.edu/niagara/papers/mjoin.pdf`.

[99] H. Wang, C. Zaniolo. ATLaS: A Native Extension of SQL for Data Mining and Stream Computations. UCLA CS Technical Report, 2002. Available at `http://citeseer.nj.nec.com/551711.html`.

[100] A. Wilschut, P. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. 1st Int. Conf. Parallel and Distributed Information Systems*, 1991, pp. 68–77.

[101] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. In *SIGMOD Record*, 31(3):9–18, Sep. 2002.

[102] Y. Yao and J. Gehrke. Query Processing for Sensor Networks. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res*, 2003, pp. 233–244.

[103] B.-K. Yi, N. Sidiropoulos, T. Johnson, H. V. Jagadish, C. Faloutsos, A. Biliris. Online Data Mining for Co-Evolving Time Sequences. In *Proc. 16th Int. Conf. on Data Engineering*, 2000, pp. 13–22.

[104] Y. Zhu, D. Shasha. StatStream: Statistical Monitoring of Thousands of Data Streams in Real Time. In *Proc. 28th Int. Conf. on Very Large Data Bases*, 2002, pp. 358–369.