Models and Issues in Data Stream Systems *

Brian Babcock Shivnath Babu Mayur Datar Rajeev Motwani Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305
{babcock,shivnath,datar,rajeev,widom}@cs.stanford.edu

Abstract

In this overview paper we motivate the need for and research issues arising from a new model of data processing. In this model, data does not take the form of persistent relations, but rather arrives in multiple, continuous, rapid, time-varying *data streams*. In addition to reviewing past work relevant to data stream systems and current projects in the area, the paper explores topics in stream query languages, new requirements and challenges in query processing, and algorithmic issues.

1 Introduction

Recently a new class of data-intensive applications has become widely recognized: applications in which the data is modeled best not as persistent relations but rather as transient *data streams*. Examples of such applications include financial applications, network monitoring, security, telecommunications data management, web applications, manufacturing, sensor networks, and others. In the data stream model, individual data items may be relational tuples, e.g., network measurements, call records, web page visits, sensor readings, and so on. However, their continuous arrival in multiple, rapid, time-varying, possibly unpredictable and unbounded streams appears to yield some fundamentally new research problems.

In all of the applications cited above, it is not feasible to simply load the arriving data into a traditional database management system (DBMS) and operate on it there. Traditional DBMS's are not designed for rapid and continuous loading of individual data items, and they do not directly support the *continuous queries* [84] that are typical of data stream applications. Furthermore, it is recognized that both *approximation* [13] and *adaptivity* [8] are key ingredients in executing queries and performing other processing (e.g., data analysis and mining) over rapid data streams, while traditional DBMS's focus largely on the opposite goal of precise answers computed by stable query plans.

In this paper we consider fundamental models and issues in developing a general-purpose *Data Stream Management System* (*DSMS*). We are developing such a system at Stanford [82], and we will touch on some of our own work in this paper. However, we also attempt to provide a general overview of the area, along with its related and current work. (Any glaring omissions are, naturally, our own fault.)

We begin in Section 2 by considering the data stream model and queries over streams. In this section we take a simple view: streams are append-only relations with transient tuples, and queries are SQL operating over these logical relations. In later sections we discuss several issues that complicate the model and query language, such as ordering, timestamping, and sliding windows. Section 2 also presents some concrete examples to ground our discussion.

In Section 3 we review recent projects geared specifically towards data stream processing, as well as a plethora of past research in areas related to data streams: active databases, continuous queries, filtering

*Work supported by NSF Grant IIS-0118173. Mayur Datar was also supported by a Microsoft Graduate Fellowship. Rajeev Motwani received partial support from an Okawa Foundation Research Grant.

systems, view management, sequence databases, and others. Although much of this work clearly has applications to data stream processing, we hope to show in this paper that there are many new problems to address in realizing a complete DSMS.

Section 4 delves more deeply into the area of query processing, uncovering a number of important issues, including:

- Queries that require an unbounded amount of memory to evaluate precisely, and approximate query processing techniques to address this problem.
- Sliding window query processing (i.e., considering "recent" portions of the streams only), both as an approximation technique and as an option in the query language since many applications prefer sliding-window queries.
- Batch processing, sampling, and synopsis structures to handle situations where the flow rate of the input streams may overwhelm the query processor.
- The meaning and implementation of blocking operators (e.g., aggregation and sorting) in the presence of unending streams.
- Continuous queries that are registered when portions of the data streams have already "passed by," yet the queries wish to reference stream history.

Section 5 then outlines some details of a query language and an architecture for a DSMS query processor designed specifically to address the issues above.

In Section 6 we review algorithmic results in data stream processing. Our focus is primarily on sketching techniques and building summary structures (*synopses*). We also touch upon sliding window computations, present some negative results, and discuss a few additional algorithmic issues.

We conclude in Section 7 with some remarks on the evolution of this new field, and a summary of directions for further work.

2 The Data Stream Model

In the data stream model, some or all of the input data that are to be operated on are not available for random access from disk or memory, but rather arrive as one or more *continuous data streams*. Data streams differ from the conventional stored relation model in several ways:

- The data elements in the stream arrive online.
- The system has no control over the order in which data elements arrive to be processed, either within a data stream or across data streams.
- Data streams are potentially unbounded in size.
- Once an element from a data stream has been processed it is discarded or archived — it cannot be retrieved easily unless it is explicitly stored in memory, which typically is small relative to the size of the data streams.

Operating in the data stream model does not preclude the presence of some data in conventional stored relations. Often, data stream queries may perform joins between data streams and stored relational data. For the purposes of this paper, we will assume that if stored relations are used, their contents remain static. Thus, we preclude any potential transaction-processing issues that might arise from the presence of updates to stored relations that occur concurrently with data stream processing.

2.1 Queries

Queries over continuous data streams have much in common with queries in a traditional database management system. However, there are two important distinctions peculiar to the data stream model. The first distinction is between *one-time queries* and *continuous queries* [84]. One-time queries (a class that includes traditional DBMS queries) are queries that are evaluated once over a point-in-time snapshot of the data set, with the answer returned to the user. Continuous queries, on the other hand, are evaluated continuously as data streams continue to arrive. Continuous queries are the more interesting class of data stream queries, and it is to them that we will devote most of our attention. The answer to a continuous query is produced over time, always reflecting the stream data seen so far. Continuous query answers may be stored and updated as new data arrives, or they may be produced as data streams themselves. Sometimes one or the other mode is preferred. For example, aggregation queries may involve frequent changes to answer tuples, dictating the stored approach, while join queries are monotonic and may produce rapid, unbounded answers, dictating the stream approach.

The second distinction is between *predefined queries* and *ad hoc queries*. A predefined query is one that is supplied to the data stream management system before any relevant data has arrived. Predefined queries are generally continuous queries, although scheduled one-time queries can also be predefined. Ad hoc queries, on the other hand, are issued online after the data streams have already begun. Ad hoc queries can be either one-time queries or continuous queries. Ad hoc queries complicate the design of a data stream management system, both because they are not known in advance for the purposes of query optimization, identification of common subexpressions across queries, etc., and more importantly because the correct answer to an ad hoc query may require referencing data elements that have already arrived on the data streams (and potentially have already been discarded). Ad hoc queries are discussed in more detail in Section 4.6.

2.2 Motivating Examples

Examples motivating a data stream system can be found in many application domains including finance, web applications, security, networking, and sensor monitoring.

- *Traderbot* [85] is a web-based financial search engine that evaluates queries over real-time streaming financial data such as stock tickers and news feeds. The Traderbot web site [85] gives some examples of one-time and continuous queries that are commonly posed by its customers.

- Modern security applications often apply sophisticated rules over network packet streams. For example, *iPolicy Networks* [52] provides an integrated security platform providing services such as firewall support and intrusion detection over multi-gigabit network packet streams. Such a platform needs to perform complex stream processing including URL-filtering based on table lookups, and correlation across multiple network traffic flows.

- Large web sites monitor web logs (clickstreams) online to enable applications such as personalization, performance monitoring, and load-balancing. Some web sites served by widely distributed web servers (e.g., Yahoo [95]) may need to coordinate many distributed clickstream analyses, e.g., to track heavily accessed web pages as part of their real-time performance monitoring.

- There are several emerging applications in the area of sensor monitoring [16, 58] where a large number of sensors are distributed in the physical world and generate streams of data that need to be combined, monitored, and analyzed.

The application domain that we use for more detailed examples is *network traffic management*, which involves monitoring network packet header information across a set of routers to obtain information on traffic flow patterns. Based on a description of Babu and Widom [10], we delve into this example in some detail to help illustrate that continuous queries arise naturally in real applications and that conventional DBMS technology does not adequately support such queries.

Consider the network traffic management system of a large network, e.g., the backbone network of an Internet Service Provider (ISP) [30]. Such systems monitor a variety of continuous data streams that may be characterized as unpredictable and arriving at a high rate, including both packet traces and network performance measurements. Typically, current traffic-management tools either rely on a special-purpose system that performs online processing of simple hand-coded continuous queries, or they just log the traffic data and perform periodic offline query processing. Conventional DBMS's are deemed inadequate to provide the kind of online continuous query processing that would be most beneficial in this domain. A data stream system that could provide effective online processing of continuous queries over data streams would allow network operators to install, modify, or remove appropriate monitoring queries to support efficient management of the ISP's network resources.

Consider the following concrete setting. Network packet traces are being collected from a number of links in the network. The focus is on two specific links: a customer link, C, which connects the network of a customer to the ISP's network, and a backbone link, B, which connects two routers within the backbone network of the ISP. Let C and B denote two streams of packet traces corresponding to these two links. We assume, for simplicity, that the traces contain just the five fields of the packet header that are listed below.

src: IP address of packet sender.

dest: IP address of packet destination.

id: Identification number given by sender so that destination can uniquely identify each packet.

len: Length of the packet.

time: Time when packet header was recorded.

Consider first the continuous query Q_1, which computes load on the link B averaged over one-minute intervals, notifying the network operator when the load crosses a specified threshold t. The functions `getminute` and `notifyoperator` have the natural interpretation.

Q_1: SELECT `notifyoperator`(sum(len))
FROM B
GROUP BY `getminute`(time)
HAVING sum(len) $>$ t

While the functionality of such a query may possibly be achieved in a DBMS via the use of triggers, we are likely to prefer the use of special techniques for performance reasons. For example, consider the case where the link B has a very high throughput (e.g., if it were an optical link). In that case, we may choose to compute an approximate answer to Q_1 by employing random sampling on the stream — a task outside the reach of standard trigger mechanisms.

The second query Q_2 isolates flows in the backbone link and determines the amount of traffic generated by each flow. A flow is defined here as a sequence of packets grouped in time, and sent from a specific source to a specific destination.

```
Q2: SELECT    flowid, src, dest, sum(len) AS flowlen
    FROM      (SELECT    src, dest, len, time
               FROM      B
               ORDER BY  time )
    GROUP BY  src, dest, getflowid(src, dest, time)
              AS flowid
```

Here `getflowid` is a user-defined function which takes the source IP address, the destination IP address, and the timestamp of a packet, and returns the identifier of the flow to which the packet belongs. We assume that the data in the view (or table expression) in the FROM clause is passed to the `getflowid` function in the order defined by the ORDER BY clause.

Observe that handling Q_2 over stream B is particularly challenging due to the presence of GROUP BY and ORDER BY clauses, which lead to "blocking" operators in a query execution plan.

Consider now the task of determining the fraction of the backbone link's traffic that can be attributed to the customer network. This query, Q_3, is an example of the kind of ad hoc continuous queries that may be registered during periods of congestion to determine whether the customer network is the likely cause.

```
Q3: (SELECT    count (*)
     FROM      C, B
     WHERE     C.src = B.src and C.dest = B.dest
               and C.id = B.id) /
    (SELECT count (*) FROM B)
```

Observe that Q_3 joins streams C and B on their keys to obtain a count of the number of common packets. Since joining two streams could potentially require unbounded intermediate storage (for example if there is no bound on the delay between a packet showing up on the two links), the user may prefer to compute an approximate answer. One approximation technique would be to maintain bounded-memory *synopses* of the two streams (see Section 6); alternatively, one could exploit aspects of the application semantics to bound the required storage (e.g., we may know that joining tuples are very likely to occur within a bounded time window).

Our final example, Q_4, is a continuous query for monitoring the source-destination pairs in the top 5 percent in terms of backbone traffic. For ease of exposition, we employ the WITH construct from SQL-99 [87].

```
Q4: WITH Load AS
      (SELECT     src, dest, sum(len) AS traffic
       FROM       B
       GROUP BY   src, dest)
    SELECT       src, dest, traffic
    FROM         Load AS L1
    WHERE        (SELECT     count(*)
                  FROM       Load AS L2
                  WHERE      L2.traffic < L1.traffic) >
                 (SELECT     0.95×count(*) FROM Load)
    ORDER BY     traffic
```

3 Review of Data Stream Projects

We now provide an overview of several past and current projects related to data stream management. We will revisit some of these projects in later sections when we discuss the issues that we are facing in building a general-purpose data stream management system at Stanford.

Continuous queries were used in the *Tapestry* system [84] for content-based filtering over an append-only database of email and bulletin board messages. A restricted subset of SQL was used as the query language in order to provide guarantees about efficient evaluation and append-only query results. The *Alert* system [74] 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*. The *XFilter* content-based filtering system [6] performs efficient filtering of XML documents based on user profiles expressed as continuous queries in the *XPath* language [94]. *Xyleme* [67] is a similar content-based filtering system that enables very high throughput with a restricted query language. The *Tribeca* stream database manager [83] provides restricted querying capability over network packet streams. The *Tangram* stream query processing system [68, 69] uses stream processing techniques to analyze large quantities of stored data.

The *OpenCQ* [57] and *NiagaraCQ* [24] systems 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 NiagaraCQ project, Shanmugasundaram et al. [79] discuss the problem of supporting blocking operators in query plans over data streams, and Viglas and Naughton [89] propose *rate-based optimization* for queries over data streams, a new optimization methodology that is based on stream-arrival and data-processing rates.

The *Chronicle data model* [55] introduced append-only ordered sequences of tuples (*chronicles*), a form of data streams. They defined a restricted view definition language and algebra (*chronicle algebra*) that operates over chronicles together with traditional relations. The focus of the work was to ensure that views defined in chronicle algebra could be maintained incrementally without storing any of the chronicles. An algebra and a declarative query language for querying ordered relations (*sequences*) was proposed by Seshadri, Livny, and Ramakrishnan [76, 77, 78]. In many applications, continuous queries need to refer to the sequencing aspect of streams, particularly in the form of sliding windows over streams. Related work in this category also includes work on temporal [80] and time-series databases [31], where the ordering of tuples implied by time can be used in querying, indexing, and query optimization.

The body of work on materialized views relates to continuous queries, since materialized views are effectively queries that need to be reevaluated or incrementally updated whenever the base data changes. Of particular importance is work on *self-maintenance* [15, 45, 71]—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* [36]—determining when certain base data can be discarded without compromising the ability to maintain a view. Nevertheless, several differences exist between materialized views and continuous queries in the data stream context: continuous queries may stream rather than store their results, they may deal with append-only input data, they may provide approximate rather than exact answers, and their processing strategy may adapt as characteristics of the data streams change.

The *Telegraph* project [8, 47, 58, 59] shares some target applications and basic technical ideas with a DSMS. Telegraph uses an *adaptive* query engine (based on the *Eddy* concept [8]) to process queries efficiently in volatile and unpredictable environments (e.g., autonomous data sources over the Internet, or sensor networks). Madden and Franklin [58] focus on query execution strategies over data streams generated by sensors, and Madden et al. [59] discuss adaptive processing techniques for multiple continuous queries. The *Tukwila* system [53] also supports adaptive query processing, in order to perform dynamic data integration over autonomous data sources.

The *Aurora* project [16] is building a new data processing system targeted exclusively towards stream monitoring applications. The core of the Aurora system consists of a large network of triggers. Each trigger is a data-flow graph with each node being one among seven built-in operators (or *boxes* in Aurora's terminology). For each stream monitoring application using the Aurora system, an *application administrator* creates and adds one or more triggers into Aurora's trigger network. Aurora performs both compile-time optimization (e.g., reordering operators, shared state for common subexpressions) and run-time optimization of the trigger network. As part of run-time optimization, Aurora detects resource overload and performs *load shedding* based on application-specific measures of quality of service.

4 Queries over Data Streams

Query processing in the data stream model of computation comes with its own unique challenges. In this section, we will outline what we consider to be the most interesting of these challenges, and describe several alternative approaches for resolving them. The issues raised in this section will frame the discussion in the rest of the paper.

4.1 Unbounded Memory Requirements

Since data streams are potentially unbounded in size, the amount of storage required to compute an exact answer to a data stream query may also grow without bound. While external memory algorithms [91] for handling data sets larger than main memory have been studied, such algorithms are not well suited to data stream applications since they do not support continuous queries and are typically too slow for real-time response. The continuous data stream model is most applicable to problems where timely query responses are important and there are large volumes of data that are being continually produced at a high rate over time. New data is constantly arriving even as the old data is being processed; the amount of computation time per data element must be low, or else the latency of the computation will be too high and the algorithm will not be able to keep pace with the data stream. For this reason, we are interested in algorithms that are able to confine themselves to main memory without accessing disk.

Arasu et al. [7] took some initial steps towards distinguishing between queries that can be answered exactly using a given bounded amount of memory and queries that must be approximated unless disk accesses are allowed. They consider a limited class of queries and, for that class, provide a complete characterization of the queries that require a potentially unbounded amount of memory (proportional to the size of the input data streams) to answer. Their result shows that without knowing the size of the input data streams, it is impossible to place a limit on the memory requirements for most common queries involving joins, unless the domains of the attributes involved in the query are restricted (either based on known characteristics of the data or through the imposition of query predicates). The basic intuition is that without domain restrictions an unbounded number of attribute values must be remembered, because they might turn out to join with tuples that arrive in the future. Extending these results to full generality remains an open research problem.

4.2 Approximate Query Answering

As described in the previous section, when we are limited to a bounded amount of memory it is not always possible to produce exact answers for data stream queries; however, high-quality approximate answers are often acceptable in lieu of exact answers. Approximation algorithms for problems defined over data streams has been a fruitful research area in the algorithms community in recent years, as discussed in detail in Section 6. This work has led to some general techniques for data reduction and synopsis construction, including: sketches [5, 35], random sampling [1, 2, 22], histograms [51, 70], and wavelets [17, 92]. Based on these summarization techniques, we have seen some work on approximate query answering. For example,

recent work [27, 37] develops histogram-based techniques to provide approximate answers for *correlated aggregate queries* over data streams, and Gilbert et al. [40] present a general approach for building small-space summaries over data streams to provide approximate answers for many classes of aggregate queries. However, research problems abound in the area of approximate query answering, with or without streams. Even the basic notion of approximations remains to be investigated in detail for queries involving more than simple aggregation. In the next two subsections, we will touch upon several approaches to approximation, some of which are peculiar to the data stream model of computation.

4.3 Sliding Windows

One technique for producing an approximate answer to a data stream query is to evaluate the query not over the entire past history of the data streams, but rather only over sliding windows of recent data from the streams. For example, only data from the last week could be considered in producing query answers, with data older than one week being discarded.

Imposing sliding windows on data streams is a natural method for approximation that has several attractive properties. It is well-defined and easily understood: the semantics of the approximation are clear, so that users of the system can be confident that they understand what is given up in producing the approximate answer. It is deterministic, so there is no danger that unfortunate random choices will produce a bad approximation. Most importantly, it emphasizes recent data, which in the majority of real-world applications is more important and relevant than old data: if one is trying in real-time to make sense of network traffic patterns, or phone call or transaction records, or scientific sensor data, then in general insights based on the recent past will be more informative and useful than insights based on stale data. In fact, for many such applications, sliding windows can be thought of not as an approximation technique reluctantly imposed due to the infeasibility of computing over all historical data, but rather as part of the desired query semantics explicitly expressed as part of the user's query. For example, queries Q_3 and Q_4 from Section 2.2, which tracked traffic on the network backbone, would likely be applied not to all traffic over all time, but rather to traffic in the recent past.

There are a variety of research issues in the use of sliding windows over data streams. To begin with, as we will discuss in Section 5.1, there is the fundamental issue of how we define timestamps over the streams to facilitate the use of windows. Extending SQL or relational algebra to incorporate explicit window specifications is nontrivial and we also touch upon this topic in Section 5.1. The implementation of sliding window queries and their impact on query optimization is a largely untouched area. In the case where the sliding window is large enough so that the entire contents of the window cannot be buffered in memory, there are also theoretical challenges in designing algorithms that can give approximate answers using only the available memory. Some recent results in this vein can be found in [9, 26].

While existing work on sequence and temporal databases has addressed many of the issues involved in time-sensitive queries (a class that includes sliding window queries) in a relational database context [76, 77, 78, 80], differences in the data stream computation model pose new challenges. Research in temporal databases [80] is concerned primarily with maintaining a full history of each data value over time, while in a data stream system we are concerned primarily with processing new data elements on-the-fly. Sequence databases [76, 77, 78] attempt to produce query plans that allow for *stream access*, meaning that a single scan of the input data is sufficient to evaluate the plan and the amount of memory required for plan evaluation is a constant, independent of the data. This model assumes that the database system has control over which sequence to process tuples from next, e.g., when merging multiple sequences, which we cannot assume in a data stream system.

4.4 Batch Processing, Sampling, and Synopses

Another class of techniques for producing approximate answers is to give up on processing every data element as it arrives, resorting to some sort of sampling or batch processing technique to speed up query execution. We describe a general framework for these techniques. Suppose that a data stream query is answered using a data structure that can be maintained incrementally. The most general description of such a data structure is that it supports two operations, `update(tuple)` and `computeAnswer()`. The `update` operation is invoked to update the data structure as each new data element arrives, and the `computeAnswer` method produces new or updated results to the query. When processing continuous queries, the best scenario is that both operations are fast relative to the arrival rate of elements in the data streams. In this case, no special techniques are necessary to keep up with the data stream and produce timely answers: as each data element arrives, it is used to update the data structure, and then new results are computed from the data structure, all in less than the average inter-arrival time of the data elements. If one or both of the data structure operations are slow, however, then producing an exact answer that is continually up to date is not possible. We consider the two possible bottlenecks and approaches for dealing with them.

Batch Processing

The first scenario is that the `update` operation is fast but the `computeAnswer` operation is slow. In this case, the natural solution is to process the data in batches. Rather than producing a continually up-to-date answer, the data elements are buffered as they arrive, and the answer to the query is computed periodically as time permits. The query answer may be considered approximate in the sense that it is not timely, i.e., it represents the exact answer at a point in the recent past rather than the exact answer at the present moment. This approach of approximation through batch processing is attractive because it does not cause any uncertainty about the accuracy of the answer, sacrificing timeliness instead. It is also a good approach when data streams are bursty. An algorithm that cannot keep up with the peak data stream rate may be able to handle the average stream rate quite comfortably by buffering the streams when their rate is high and catching up during the slow periods. This is the approach used in the XJoin algorithm [88].

Sampling

In the second scenario, `computeAnswer` may be fast, but the `update` operation is slow — it takes longer than the average inter-arrival time of the data elements. It is futile to attempt to make use of all the data when computing an answer, because data arrives faster than it can be processed. Instead, some tuples must be skipped altogether, so that the query is evaluated over a sample of the data stream rather than over the entire data stream. We obtain an approximate answer, but in some cases one can give confidence bounds on the degree of error introduced by the sampling process [48]. Unfortunately, for many situations (including most queries involving joins [20, 22]), sampling-based approaches cannot give reliable approximation guarantees. Designing sampling-based algorithms that can produce approximate answers that are provably close to the exact answer is an important and active area of research.

Synopsis Data Structures

Quite obiously, data structures where both the `update` and the `computeAnswer` operations are fast are most desirable. For classes of data stream queries where no exact data structure with the desired properties exists, one can often design an approximate data structure that maintains a small *synopsis* or *sketch* of the data rather than an exact representation, and therefore is able to keep computation per data element to a minimum. Performing data reduction through synopsis data structures as an alternative to batch processing

or sampling is a fruitful research area with particular relevance to the data stream computation model. Synopsis data structures are discussed in more detail in Section 6.

4.5 Blocking Operators

A *blocking query operator* is a query operator that is unable to produce the first tuple of its output until it has seen its entire input. Sorting is an example of a blocking operator, as are aggregation operators such as SUM, COUNT, MIN, MAX, and AVG. If one thinks about evaluating continuous stream queries using a traditional tree of query operators, where data streams enter at the leaves and final query answers are produced at the root, then the incorporation of blocking operators into the query tree poses problems. Since continuous data streams may be infinite, a blocking operator that has a data stream as one of its inputs will never see its entire input, and therefore it will never be able to produce any output. Clearly, blocking operators are not very suitable to the data stream computation model, but aggregate queries are extremely common, and sorted data is easier to work with and can often be processed more efficiently than unsorted data. Doing away with blocking operators altogether would be problematic, but dealing with them effectively is one of the more challenging aspects of data stream computation.

Blocking operators that are the root of a tree of query operators are more tractable than blocking operators that are interior nodes in the tree, producing intermediate results that are fed to other operators for further processing (for example, the "sort" phase of a sort-merge join, or an aggregate used in a subquery). When we have a blocking aggregation operator at the root of a query tree, if the operator produces a single value or a small number of values, then updates to the answer can be streamed out as they are produced. When the answer is larger, however, such as when the query answer is a relation that is to be produced in sorted order, it is more practical to maintain a data structure with the up-to-date answer, since continually retransmitting the entire answer would be cumbersome. Neither of these two approaches works well for blocking operators that produce intermediate results, however. The central problem is that the results produced by blocking operators may continue to change over time until all the data has been seen, so operators that are consuming those results cannot make reliable decisions based on the results at an intermediate stage of query execution.

One approach to handling blocking operators as interior nodes in a query tree is to replace them with non-blocking analogs that perform approximately the same task. An example of this approach is the *juggle* operator [72], which is a non-blocking version of sort: it aims to locally reorder a data stream so that tuples that come earlier in the desired sort order are produced before tuples that come later in the sort order, although some tuples may be delivered out of order. An interesting open problem is how to extend this work to other types of blocking operators, as well as to quantify the error that is introduced by approximating blocking operators with non-blocking ones.

Tucker et al. [86] have proposed a different approach to blocking operators. They suggest augmenting data streams with assertions about what can and cannot appear in the remainder of the data stream. These assertions, which are called *punctuations*, are interleaved with the data elements in the streams. An example of the type of punctuation one might see in a stream with an attribute called `daynumber` is "for all future tuples, `daynumber` ≥ 10." Upon seeing this punctuation, an aggregation operator that was grouping by daynumber could stream out its answers for all `daynumbers` less than 10. Similarly, a join operator could discard all its saved state relating to previously-seen tuples in the joining stream with `daynumber` < 10, reducing its memory consumption.

An interesting open problem is to formalize the relationship between punctuation and the memory requirements of a query — e.g., a query that might otherwise require unbounded memory could be proved to be answerable in bounded memory if guarantees about the presence of appropriate punctuation are provided. Closely related is the idea of schema-level assertions (constraints) on data streams, which also may help with blocking operators and other aspects of data stream processing. For example, we may know that

`daynumbers` are clustered or strictly increasing, or when joining two stream we may know that a kind of "referential integrity" exists in the arrival of join attribute values. In both cases we may use these constraints to "unblock" operators or reduce memory requirements.

4.6 Queries Referencing Past Data

In the data stream model of computation, once a data element has been streamed by, it cannot be revisited. This limitation means that ad hoc queries that are issued after some data has already been discarded may be impossible to answer accurately. One simple solution to this problem is to stipulate that ad hoc queries are only allowed to reference future data: they are evaluated as though the data streams began at the point when the query was issued, and any past stream elements are ignored (for the purposes of that query). While this solution may not appear very satisfying, it may turn out to be perfectly acceptable for many applications.

A more ambitious approach to handling ad hoc queries that reference past data is to maintain summaries of data streams (in the form of general-purpose synopses or aggregates) that can be used to give approximate answers to future ad hoc queries. Taking this approach requires making a decision in advance about the best way to use memory resources to give good approximate answers to a broad range of possible future queries. The problem is similar in some ways to problems in physical database design such as selection of indexes and materialized views [23]. However, there is an important difference: in a traditional database system, when an index or view is lacking, it is possible to go to the underlying relation, albeit at an increased cost. In the data stream model of computation, if the appropriate summary structure is not present, then no further recourse is available.

Even if ad hoc queries are declared only to pertain to future data, there are still research issues involved in how best to process them. In data stream applications, where most queries are long-lived continuous queries rather than ephemeral one-time queries, the gains that can be achieved by multi-query optimization can be significantly greater than what is possible in traditional database systems. The presence of ad hoc queries transforms the problem of finding the best query plan for a set of queries from an offline problem to an online problem. Ad hoc queries also raise the issue of adaptivity in query plans. The Eddy query execution framework [8] introduces the notion of flexible query plans that adapt to changes in data arrival rates or other data characteristics over time. Extending this idea to adapt the joint plan for a set of continuous queries as new queries are added and old ones are removed remains an open research area.

5 Proposal for a DSMS

At Stanford we have begun the design and prototype implementation of a comprehensive DSMS called *STREAM* (for STanford StREam DatA Manager) [82]. As discussed in earlier sections, in a DSMS traditional one-time queries are replaced or augmented with continuous queries, and techniques such as sliding windows, synopsis structures, approximate answers, and adaptive query processing become fundamental features of the system. Other aspects of a complete DBMS also need to be reconsidered, including query languages, storage and buffer management, user and application interfaces, and transaction support. In this section we will focus primarily on the query language and query processing components of a DSMS and only touch upon other issues based on our initial experiences.

5.1 Query Language for a DSMS

Any general-purpose data management system must have a flexible and intuitive method by which the users of the system can express their queries. In the STREAM project, we have chosen to use a modified version of SQL as the query interface to the system (although we are also providing a means to submit query plans directly). SQL is a well-known language with a large user population. It is also a declarative language

that gives the system flexibility in selecting the optimal evaluation procedure to produce the desired answer. Other methods for receiving queries from users are possible; for example, the Aurora system described in [16] uses a graphical "boxes and arrows" interface for specifying data flow through the system. This interface is intuitive and gives the user more control over the exact series of steps by which the query answer is obtained than is provided by a declarative query language.

The main modification that we have made to standard SQL, in addition to allowing the FROM clause to refer to streams as well as relations, is to extend the expressiveness of the query language for sliding windows. It is possible to formulate sliding window queries in SQL by referring to timestamps explicitly, but it is often quite awkward. SQL-99 [14, 81] introduces analytical functions that partially address the shortcomings of SQL for expressing sliding window queries by allowing the specification of moving averages and other aggregation operations over sliding windows. However, the SQL-99 syntax is not sufficiently expressive for data stream queries since it cannot be applied to non-aggregation operations such as joins.

The notion of sliding windows requires at least an ordering on data stream elements. In many cases, the arrival order of the elements suffices as an "implicit timestamp" attached to each data element; however, sometimes it is preferable to use "explicit timestamps" provided as part of the data stream. Formally we say (following [16]) that a data stream S consists of a set of (tuple, timestamp) pairs: $S = \{(s_1, i_1), (s_2, i_2), \ldots, (s_n, i_n)\}$. The timestamp attribute could be a traditional timestamp or it could be a sequence number — all that is required is that it comes from a totally ordered domain with a distance metric. The ordering induced by the timestamps is used when selecting the data elements making up a sliding window.

We extend SQL by allowing an optional *window specification* to be provided, enclosed in brackets, after a stream (or subquery producing a stream) that is supplied in a query's FROM clause. A window specification consists of:

1. an optional partitioning clause, which partitions the data into several groups and maintains a separate window for each group,

2. a window size, either in "physical" units (i.e., the number of data elements in the window) or in "logical" units (i.e., the range of time covered by a window, such as 30 days), and

3. an optional filtering predicate.

As in SQL-99, physical windows are specified using the `ROWS` keyword (e.g., `ROWS 50 PRECEDING`), while logical windows are specified via the `RANGE` keyword (e.g., `RANGE 15 MINUTES PRECEDING`). In lieu of a formal grammar, we present several examples to illustrate our language extension.

The underlying source of data for our examples will be a stream of telephone call records, each with four attributes: `customer_id`, `type`, `minutes`, and `timestamp`. The `timestamp` attribute is the ordering attribute for the records. Suppose a user wanted to compute the average call length, but considering only the ten most recent long-distance calls placed by each customer. The query can be formulated as follows:

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

where the expression in braces defines a sliding window on the stream of calls.

Contrast the previous query to a similar one that computes the average call length considering only long-distance calls that are among the last 10 calls of all types placed by each customer:

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

The distinction between filtering predicates applied before calculating the sliding window cutoffs and predicates applied after windowing motivates our inclusion of an optional WHERE clause within the window specification.

Here is a slightly more complicated example returning the average length of the last 1000 telephone calls placed by "Gold" customers:

```
SELECT  AVG(V.minutes)
FROM    (SELECT S.minutes
        FROM Calls S, Customers T
        WHERE S.customer_id = T.customer_id
        AND T.tier = 'Gold')
        V [ROWS 1000 PRECEDING]
```

Notice that in this example, the stream of calls must be joined to the Customers relation before applying the sliding window.

5.2 Timestamps in Streams

In the previous section, sliding windows are defined with respect to a timestamp or sequence number attribute representing a tuple's arrival time. This approach is unambiguous for tuples that come from a single stream, but it is less clear what is meant when attempting to apply sliding windows to composite tuples that are derived from tuples from multiple underlying streams (e.g., windows on the output of a join operator). What should the timestamp of a tuple in the join result be when the timestamps of the tuples that were joined to form the result tuple are different? Timestamp issues also arise when a set of distributed streams make up a single logical stream, as in the web monitoring application described in Section 2.2, or in truly distributed streams such as sensor networks when comparing timestamps across stream elements may be relevant.

In the previous section we introduced *implicit* timestamps, in which the system adds a special field to each incoming tuple, and *explicit* timestamps, in which a data attribute is designated as the timestamp. Explicit timestamps are used when each tuple corresponds to a real-world event at a particular time that is of importance to the meaning of the tuple. Implicit timestamps are used when the data source does not already include timestamp information, or when the exact moment in time associated with a tuple is not important, but general considerations such as "recent" or "old" may be important. The distinction between implicit and explicit timestamps is similar to that between *transaction* and *valid* time in the temporal database literature [80].

Explicit timestamps have the drawback that tuples may not arrive in the same order as their timestamps — tuples with later timestamps may come before tuples with earlier timestamps. This lack of guaranteed ordering makes it difficult to perform sliding window computations that are defined in relation to explicit timestamps, or any other processing based on order. However, as long as an input stream is "almost-sorted" by timestamp, except for local perturbations, then out-of-order tuples can easily be corrected with little buffering. It seems reasonable to assume that even when explicit timestamps are used, tuples will be delivered in roughly increasing timestamp order.

Let us now look at how to assign appropriate timestamps to tuples output by binary operators, using join as an example. There are several possible approaches that could be taken; we discuss two. The first approach, which fits better with implicit timestamps, is to provide no guarantees about the output order of

tuples from a join operator, but to simply assume that tuples that arrive earlier are likely to pass through the join earlier; exact ordering may depend on implementation and scheduling vagaries. Each tuple that is produced by a join operator is assigned an implicit timestamp that is set to the time that it was produced by the join operator. This "best-effort" approach has the advantage that it maximizes implementation flexibility; it has the disadvantage that it makes it impossible to impose precisely-defined, deterministic sliding-window semantics on the results of subqueries.

The second approach, which fits with either explicit or implicit timestamps, is to have the user specify as part of the query what timestamp is to be assigned to tuples resulting from the join of multiple streams. One simple policy is that the order in which the streams are listed in the FROM clause of the query represents a prioritization of the streams. The timestamp for a tuple output by a join should be the timestamp of the joining tuple from the input stream listed first in the FROM clause. This approach can result in multiple tuples with the same timestamp; for the purpose of ordering the results, ties can be broken using the timestamp of the other input stream. For example, if the query is

```
SELECT  *
FROM    S1 [ROWS 1000 PRECEDING],
        S2 [ROWS 100 PRECEDING]
WHERE   S1.A = S2.B
```

then the output tuples would first be sorted by the timestamp of S1, and then ties would be broken according to the timestamp of S2.

The second, stricter approach to assigning timestamps to the results of binary operators can have a drawback from an implementation point of view. If it is desirable for the output from a join to be sorted by timestamp, the join operator will need to buffer output tuples until it can be certain that future input tuples will not disrupt the ordering of output tuples. For example, if S1's timestamp has priority over S2's and a recent S1 tuple joins with an S2 tuple, it is possible that a future S2 tuple will join with an older S1 tuple that still falls within the current window. In that case, the join tuple that was produced second belongs before the join tuple that was produced first. In a query tree consisting of multiple joins, the extra latency introduced for this reason could propagate up the tree in an additive fashion. If the inputs to the join operator did not have sliding windows at all, then the join operator could never confidently produce outputs in sorted order.

As mentioned earlier, sliding windows have two distinct purposes: sometimes they are an important part of the query semantics, and other times they are an approximation scheme to improve query efficiency and reduce data volumes to a manageable size. When the sliding window serves mostly to increase query processing efficiency, then the best-effort approach, which allows wide latitude over the ordering of tuples, is usually acceptable. On the other hand, when the ordering of tuples plays a significant role in the meaning of the query, such as for query-defined sliding windows, then the stricter approach may be preferred, even at the cost of less efficient implementation. A general-purpose data stream processing system should support both types of sliding windows, and the query language should allow users to specify one or the other.

In our system, we add an extra keyword, `RECENT`, that replaces `PRECEDING` when a "best-effort" ordering may be used. For example, the clause `ROWS 10 PRECEDING` specifies a window consisting of the previous 10 tuples, strictly sorted by timestamp order. By comparison, `ROWS 10 RECENT` also specifies a sliding window consisting of 10 records, but the DSMS is allowed to use its own ordering to produce the sliding window, rather than being constrained to follow the timestamp ordering. The `RECENT` keyword is only used with "physical" window sizes specified as a number of records; "logical" windows such as `RANGE 3 DAYS PRECEDING` must use the `PRECEDING` keyword.

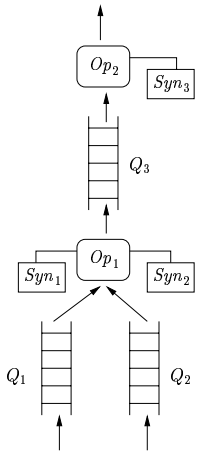

Figure 1: A portion of a query plan in our DSMS.

5.3 Query Processing Architecture of a DSMS

In this section, we describe the query processing architecture of our DSMS. So far we have focused on continuous queries only. When a query is registered, a *query execution plan* is produced that begins executing and continues indefinitely. We have not yet addressed ad hoc queries registered after relevant streams have begun (Section 4.6).

Query execution plans in our system consist of *operators* connected by *queues*. Operators can maintain intermediate state in *synopsis* data structures. A portion of an example query plan is shown in Figure 1, with one binary operator (Op_1) and one unary operator (Op_2). The two operators are connected by a queue Q_3, and operator Op_1 has two input queues, Q_1 and Q_2. Also shown in Figure 1 are two synopsis structures used by operator Op_1, Syn_1 and Syn_2, one per input. For example, Op_1 could be a *sliding window join operator*, which maintains a sliding window synopsis for each join input (Section 4.3). The system memory is partitioned dynamically among the synopses and queues in query plans, along with the buffers used for handling streams coming over the network and a cache for disk-resident data. Note that both Aurora [16] and Eddies [8] use a single globally-shared queue for inter-operator data flow instead of separate queues between operators as in Figure 1.

Operators in our system are scheduled for execution by a central *scheduler*. During execution, an operator reads data from its input queues, updates the synopsis structures that it maintains, and writes results to its output queues. (Our operators thus adhere to the `update` and `computeAnswer` model discussed in Section 4.4.) The period of execution of an operator is determined dynamically by the scheduler and the operator returns control back to the scheduler once its period expires. We are experimenting with different policies for scheduling operators and for determining the period of execution. The period of execution may be based on time, or it may be based on other quantities, such as the number of tuples consumed or produced. Both Aurora and Eddies have chosen to perform fine-grained scheduling where, in each step, the scheduler chooses a tuple from the global queue and passes it to an operator for processing, an approach that our scheduler could choose if appropriate.

We expect continuous queries and the data streams on which they operate to be long-running. During the lifetime of a continuous query parameters such as stream data characteristics, stream flow rates, and the number of concurrently running queries may vary considerably. To handle these fluctuations, all of our operators are *adaptive*. So far we have focused primarily on adaptivity to available memory, although other factors could be considered, including using disk to increase temporary storage at the expense of latency.

Our approach to memory adaptivity is basically one of trading accuracy for memory. Specifically, each operator maximizes the accuracy of its output based on the size of its available memory, and handles dynamic changes in the size of its available memory gracefully, since at run-time memory may be taken away from one operator and given to another. As a simple example, a sliding window join operator as discussed above may be used as an approximation to a join over the entire history of input streams. If so, the larger the windows (stored in available memory), the better the approximation. Other examples include duplicate elimination using limited-size hash tables, and sampling using *reservoirs* [90]. The Aurora system [16] also proposes adaptivity and approximations, and uses load-shedding techniques based on application-specified measures of quality of service for graceful degradation in the face of system overload.

Our fundamental approach of trading accuracy for memory brings up some interesting problems:

- We first need to understand how different query operators can produce approximate answers under limited memory, and how approximate results behave when operators are composed in query plans.
- Given a query plan as a tree of operators and a certain amount of memory, how can the DSMS allocate memory to the operators to maximize the accuracy of the answer to the query (i.e., minimize approximation)?
- Under changing conditions, how can the DSMS reallocate memory among operators?
- Suppose we are given a query rather than a query plan. How does the query optimizer efficiently find the plan that, with the best memory allocation, minimizes approximation? Should plans be modified when conditions change?
- Even further, since synopses could be shared among query plans [75], how do we optimally consider a set of queries, which may be weighted by importance?

In addition to memory management, we are faced the problem of scheduling multiple query plans in a DSMS. The scheduler needs to provide *rate synchronization* within operators (such as stream joins) and also across pipelined operators in query plans [8, 89]. Time-varying arrival rates of data streams and time-varying output rates of operators add to the complexity of scheduling. Scheduling decisions also need to take into account memory allocation across operators, including management of buffers for incoming streams, availability of synopses on disk as opposed to in memory, and the performance requirements of individual queries.

Aside from the query processing architecture, user and application interfaces need to be reinvestigated in a DSMS given the dynamic environment in which it operates. Systems such as Aurora [16] and Hancock [25] completely eliminate declarative querying and provide only procedural mechanisms for querying. In contrast, we will provide a declarative language for continuous queries, similar to SQL but extended with operators such as those discussed in Section 5.1, as well as a mechanism for directly submitting plans in the query algebra that underlies our language.

We are developing a comprehensive DSMS interface that allows users and administrators to visually monitor the execution of continuous queries, including memory usage and approximation behavior. We will also provide a way for administrators to adjust system parameters as queries are running, including memory allocation and scheduling policies.

6 Algorithmic Issues

The algorithms community has been fairly active of late in the area of data streams, typically motivated by problems in databases and networking. The model of computation underlying the algorithmic work is

similar to that in Section 2 and can be formally stated as follows: *A* data stream algorithm *takes as input a sequence of data items $x_1, \ldots, x_N, \ldots$ called the* data stream*, where the sequence is scanned only once in the increasing order of the indexes. The algorithm is required to maintain the value of a function f on the prefix of the stream seen so far.*

The main complexity measure is the space used by the algorithm, although the time required to process each stream element is also relevant. In some cases, the algorithm maintains a data structure which can be used to compute the value of the function f on demand, and then the time required to process each such query also becomes of interest. Henzinger et al. [49] defined a similar model but also allowed the algorithm to make multiple passes over the stream data, making the number of passes itself a complexity measure. We will restrict our attention to algorithms which are allowed only one pass.

We will measure space and time in terms of the parameter N which denotes the number of stream elements seen so far. The primary performance goal is to ensure that the space required by a stream algorithm is "small." Ideally, one would want the memory bound to be independent of N (which is unbounded). However, for most interesting problems it is easy to prove a space lower bound that precludes this possibility, thereby forcing us to settle for bounds that are merely sublinear in N. A problem is considered to be "well-solved" if one can devise an algorithm which requires at most $O(poly(\log N))$ space and $O(poly(\log N))$ processing time per data element or query[1]. We will see that in some cases it is impossible to achieve such an algorithm, even if one is willing to settle for approximations.

The rest of this section summarizes the state of the art for data stream algorithms, at least as relevant to databases. We will focus primarily on the problems of creating summary structures (synopses) for a data stream, such as histograms, wavelet representation, clustering, and decision trees; in addition, we will also touch upon known lower bounds for space and time requirements of data stream algorithms. Most of these summary structures have been considered for traditional databases [13]. The challenge is to adapt some of these techniques to the data stream model.

6.1 Random Samples

Random samples can be used as a summary structure in many scenarios where a small sample is expected to capture the essential characteristics of the data set [65]. It is perhaps the easiest form of summarization in a DSMS and other synopses can be built from a sample itself. In fact, the join synopsis in the AQUA system [2] is nothing but a uniform sample of the base relation. Recently stratified sampling has been proposed as an alternative to uniform sampling to reduce error due to the variance in data and also to reduce error for group-by queries [1, 19]. To actually compute a random sample over a data stream is relatively easy. The reservoir sampling algorithm of Vitter [90] makes one pass over the data set and is well suited for the data stream model. There is also an extension by Chaudhuri, Motwani and Narasayya [22] to the case of weighted sampling.

6.2 Sketching Techniques

In their seminal paper, Alon, Matias and Szegedy [5] introduced the notion of *randomized sketching* which has been widely used ever since. Sketching involves building a summary of a data stream using a small amount of memory, using which it is possible to estimate the answer to certain queries (typically, "distance" queries) over the data set.

Let $\boldsymbol{S} = (x_1, \ldots, x_N)$ be a sequence of elements where each x_i belongs to the domain $D = \{1, \ldots, d\}$. Let the multiplicity $m_i = |\{j | x_j = i\}|$ denote the number of occurrences of value i in the sequence $\boldsymbol{S}$. For $k \geq 0$, the *kth frequency moment* F_k of $\boldsymbol{S}$ is defined as $F_k = \sum_{i=1}^{d} m_i^k$; further, we define $F_\infty^* = \max_i m_i$.

[1] We use $poly$ to denote a polynomial function.

The frequency moments capture the statistics of the distribution of values in $\boldsymbol{S}$ — for instance, F_0 is the number of distinct values in the sequence, F_1 is the length of the sequence, F_2 is the self-join size (also called Gini's index of homogeneity), and F_∞ is the most frequent item's multiplicity. It is not very difficult to see that an exact computation of these moments requires linear space and so we focus our attention on approximations.

The problem of efficiently estimating the number of distinct values (F_0) has received particular attention in the database literature, particularly in the context of using single pass or random sampling algorithms [18, 46]. A sketching technique to compute F_0 was presented earlier by Flajolet and Martin [35]; however, this had the drawback of requiring explicit families of hash functions with very strong independence properties. This requirement was relaxed by Alon, Matias, and Szegedy [5] who presented a sketching technique to estimate F_0 within a constant factor[2]. Their technique uses linear hash functions and requires only $O(\log d)$ memory. The key contribution of Alon et al. [5] was a sketching technique to estimate F_2 that uses only $O(\log d + \log N)$ space and provides arbitrarily small approximation factors. This technique has found many applications in the database literature, including join size estimation [4], estimating L_1 norm of vectors [33], and processing complex aggregate queries over multiple streams [27, 37]. It remains an open problem to come up with techniques to maintain correlated aggregates [37] that have *provable* guarantees.

The key idea behind the F_2-sketching technique can be described as follows: *Every element i in the domain D is hashed uniformly at random onto a value $z_i \in \{-1, +1\}$. Define the random variable $X = \sum_i m_i z_i$ and return X^2 as the estimator of F_2.* Observe that the estimator can be computed in a single pass over the data provided we can efficiently compute the z_i values. If the hash functions have four-way independence[3], it is easy to prove that the quantity X^2 has expectation equal to F_2 and variance less than $2F_2^2$. Using standard tricks, we can combine several independent estimators to accurately and with high probability obtain an estimate of F_2. At an intuitive level, we can view this technique as a tug-of-war where elements are randomly assigned to one of the two sides of the rope based on the value i; the square of the displacement of the rope captures the skew F_2 in the data.

Observe that computing the self-join size of a relation is exactly the same as computing F_2 for the values of the join attribute in the relation. Alon et al. [4] extended this technique to estimating the join size between two distinct relations A and B, as follows. Let Y and Z be random variables corresponding to A and B, respectively, similar to the random variable X above; the mapping from domain values i to z_i is the same for both relations. Then, it can be proved that the estimator YZ has expected value equal to $|A \bowtie B|$ and variance less than $2|A \bowtie A||B \bowtie B|$. In order to get small relative error, we can use $O(\frac{|A \bowtie A||B \bowtie B|}{|A \bowtie B|^2})$ independent estimators. Observe that for estimating joins between two relations, the number of estimators depends on the data distribution. In a recent paper, Dobra et al. [27] extended this technique to estimate the size of multi-way joins and for answering complex aggregates queries over them. They also provide techniques to optimally partition the data domain and use estimators on each partition independently, so as to minimize the total memory requirement.

The frequency moment F_2 can also be viewed as the L_2 norm of a vector whose value along the ith dimension is the multiplicity m_i. Thus, the above technique can be used to compute the L_2 norm under a update model for vectors, where each data element (v, i) increments (or decrements) some m_i by a quantity v. On seeing such an update, we update the corresponding sketch by adding vz_i to the sum. The sketching idea can also be extended to compute the L_1 norm of a vector, as follows. Let us assume that each dimension of the underlying vector is an integer, bounded by M. Consider the unary representation of the vector. It has Md bit positions (elements), where d is the dimension of the underlying vector. A 1 in the unary

[2]As discussed in Section 6.7, recently Bar-Yossef et al. [12] and Gibbons and Tirthapura [38] have devised algorithms which, under certain conditions, provide arbitrarily small approximation factors without recourse to perfect hash functions.

[3]Hash functions with four-way independence can be obtained using standard techniques involving the use of parity check matrices of BCH codes [65].

representation denotes that the element corresponding to the bit position is present once; otherwise, it is not present. Then F_2 captures the L_1 norm of the vector. The catch is that, given an element r_i along dimension i, it is required that we can efficiently compute the range sum $\sum_{j=0}^{r_i-1} z_{i,j}$ of the hash values corresponding to the pertinent bit positions that are set to 1. Feigenbaum et al. [33] showed how to construct such a family of *range-summable* ± 1-valued hash functions with limited (four-way) independence. Indyk [50] provided a uniform framework to compute the L_p norm (for $p \in (0, 2]$) using the so-called p-stable distributions, improving upon the previous paper [33] for estimating the L_1 norm, in that it allowed for arbitrary addition and deletion updates in every dimension. The ability to efficiently compute L_1 and L_2 norm of the difference of two vectors is central to some synopsis structures designed for data streams.

6.3 Histograms

Histograms are commonly-used summary structures to succinctly capture the distribution of values in a data set (i.e., a column, or possibly even a collection of columns, in a table). They have been employed for a multitude of tasks such as query size estimation, approximate query answering, and data mining. We consider the summarization of data streams using histograms. There are several different types of histograms that have been proposed in the literature. Some popular definitions are:

- **V-Optimal Histogram:** These approximate the distribution of a set of values $v_1, \ldots, v_n$ by a piecewise-constant function $\widehat{v}(i)$, so as to minimize the sum of squared error $\sum_i (v_i - \widehat{v}(i))^2$.
- **Equi-Width Histograms:** These partition the domain into buckets such that the number of v_i values falling into each bucket is uniform across all buckets. In other words, they maintain quantiles for the underlying data distribution as the bucket boundaries.
- **End-Biased Histograms:** These will maintain exact counts of items that occur with frequency above a threshold, and approximate the other counts by an uniform distribution. Maintaining the count of such frequent items is related to Iceberg queries [32].

We give an overview of recent work on computing such histograms over data streams.

V-Optimal Histograms over Data Streams

Jagadish et al. [54] showed how to compute optimal V-Optimal Histograms for a given data set using dynamic programming. The algorithm uses $O(N)$ space and requires $O(N^2 B)$ time, where N is the size of the data set and B is the number of buckets. This is prohibitive for data streams. Guha, Koudas and Shim [43] adapted this algorithm to *sorted* data streams. Their algorithm constructs an arbitrarily-close V-Optimal Histogram (i.e., with error arbitrarily close to that of the optimal histogram), using $O(B^2 \log N)$ space and $O(B^2 \log N)$ time per data element.

In a recent paper, Gilbert et al. [39], removed the restriction that the data stream be sorted, providing algorithms based on the sketching technique described earlier for computing L_2 norms. The idea is to view each data element as an update to an underlying vector of length N that we are trying to approximate using the best B-bucket histogram. The time to process a data element, the time to reconstruct the histogram, and the size of the sketch are each bounded by $poly(B, \log N, 1/\epsilon)$, where ϵ is the relative error we are willing to tolerate. Their algorithm proceeds by first constructing a *robust* approximation to the underlying "signal." A *robust* approximation is built by repeatedly adding a dyadic interval of constant value[4] which best reduces the approximation error. In order to find such a dyadic interval it is necessary to efficiently compute the

[4] A signal that corresponds to a constant value over the dyadic interval and zero everywhere else.

sketch of the original signal minus the constant dyadic interval[5]. This translates to efficiently computing the range sum of p-stable random variables (used for computing the L_2 sketch, see Indyk [50]) over the dyadic interval. Gilbert et al. [39] show how to construct such efficiently range-summable p-stable random variables. From the robust histogram they cull a histogram of desired accuracy and with B buckets.

Equi-Width Histograms and Quantiles

Equi-width histograms based on histograms are summary structures which characterize data distributions in a manner that is less sensitive to outliers. In traditional databases they are used by optimizers for selectivity estimation. Parallel database systems employ value range data partitioning that requires generation of quantiles or splitters that partition the data into approximately equal parts. Recently, Greenwald and Khanna [41] presented a single-pass deterministic algorithm for efficient computation of quantiles. Their algorithm needs $O(\frac{1}{\epsilon}\log \epsilon N)$ space and guarantees a precision of ϵN. They employ a novel data structure that maintains a sample of the values seen so far (quantiles), along with a range of possible ranks that the samples can take. The error associated with each quantile is the width of this range. They periodically merge quantiles with "similar" errors so long as the error for the combined quantile does not exceed ϵN. This algorithm improves upon the previous set of results by Manku, Rajagopalan, and Lindsay [61, 62] and Chaudhuri, Motwani, and Narasayya [21].

End-Biased Histograms and Iceberg Queries

Many applications maintain simple aggregates (count) over an attribute to find aggregate values above a specified threshold. These queries are referred to as *iceberg queries* [32]. Such iceberg queries arise in many applications, including data mining, data warehousing, information retrieval, market basket analysis, copy detection, and clustering. For example, a search engine might be interested in gathering search terms that account for more than 1% of the queries. Such frequent item summaries are useful for applications such as caching and analyzing trends. Fang et al. [32] gave an efficient algorithm to compute Iceberg queries over disk-resident data. Their algorithm requires multiple passes which is not suited to the streaming model. In a recent paper, Manku and Motwani [60] presented randomized and deterministic algorithms for frequency counting and iceberg queries over data streams. The randomized algorithm uses adaptive sampling and the main idea is that any item which accounts for an ϵ fraction of the items is highly likely to be a part of a uniform sample of size $\frac{1}{\epsilon}$. The deterministic algorithm maintains a sample of the distinct items along with their frequency. Whenever a new item is added, it is given a benefit of doubt by over-estimating its frequency. If we see an item that already exists in the sample, its frequency is incremented. Periodically items with low frequency are deleted. Their algorithms require $O(\frac{1}{\epsilon}\log(\epsilon N))$ space, where N is the length of the data stream, and guarantee that any element is undercounted by at most ϵN. Thus, these algorithms report all items of count greater than ϵN. Moreover, for all items reported, they guarantee that the reported count is less than the actual count, but by no more than ϵN.

6.4 Wavelets

Wavelets are often used as a technique to provide a summary representation of the data. Wavelets coefficients are projections of the given signal (set of data values) onto an orthogonal set of basis vector. The choice of basis vectors determines the type of wavelets. Often Haar wavelets are used in databases for their ease of computation. Wavelet coefficients have the desirable property that the signal reconstructed from the top few wavelet coefficients best approximates the original signal in terms of the L_2 norm.

[5]That is, a sketch for L_2 norm of the difference between the original signal and the dyadic interval with constant value.

Recent papers have demonstrated the efficacy of wavelets for different tasks such as selectivity estimation [63], data cube approximation [93] and computing multi-dimensional aggregates [92]. This body of work indicates that estimates obtained from wavelets were more accurate than those obtained from histograms with the same amount of memory. Chakrabarti et al. [17] propose the use of wavelets for general purpose approximate query processing and demonstrate how to compute joins, aggregations, and selections entirely in the wavelet coefficient domain.

To extend this body of work to data streams, it becomes important to devise techniques for computing wavelets in the streaming model. In a related development, Matias, Vitter, and Wang [64] show how to dynamically maintain the top wavelet coefficients efficiently as the underlying data distribution is updated. There has been recent work in computing the top wavelet coefficients in the data stream model. The technique of Gilbert et al. [39], to approximate the best dyadic interval that most reduces the error, gives rise to an easy greedy algorithm to find the best B-term Haar wavelet representation. This is because the Haar wavelet basis consists of dyadic intervals with constant values. This improves upon a previous result by Gilbert et al. [40]. If the data is presented in a sorted order, there is a simple algorithm that maintains the best B-term Haar wavelet representation using $O(B + \log N)$ space in a deterministic manner [40].

While there has been lot of work on summary structures, it remains an interesting open problem to address the issue of global space allocation between different synopses vying for the same space. It requires that we come up with a global error metric for the synopses, which we minimize given the (main memory) space constraint. Moreover, the allocation will have to be dynamic as the underlying data distribution and query workload changes over time.

6.5 Sliding Windows

As discussed in Section 4, sliding windows prevent *stale* data from influencing analysis and statistics, and also serve as a tool for approximation in face of bounded memory. There has been very little work on extending summarization techniques to sliding windows and it remains a ripe research area. We briefly describe some of the recent work.

Datar et al. [26] showed how to maintain simple statistics over sliding windows, including the sketches used for computing the L_1 or L_2 norm. Their technique requires a multiplicative space overhead of $O(\frac{1}{\epsilon} \log N)$, where N is the length of the sliding window and ϵ is the accuracy parameter. This enables the adaptation of the sketching-based algorithms to the sliding windows model. They also provide space lower bounds for various problems in the sliding windows model. In another paper, Babock, Datar and Motwani [9] adapt the reservoir sampling algorithm to the sliding windows case. In their paper for computing Iceberg queries over data streams, Manku and Motwani [60] also present techniques to adapt their algorithms to the sliding window model. Guha and Koudas [42] have adapted their earlier paper [43], to provide a technique for maintaining V-Optimal Histograms over sorted data streams for the sliding window model; however, they require the buffering of all the elements in the sliding window. The space requirement is linear in the size of the sliding window (N), although update time per data element is amortized to $O((B^3/\epsilon^2) \log^3 N)$, where B is the number of buckets in the histogram and ϵ is the accuracy parameter.

Some open problems for sliding windows are: clustering, maintaining top wavelet coefficients, maintaining statistics like variance, and computing correlated aggregates [37].

6.6 Negative Results

There is an emerging set of negative results on space-time requirements of algorithms that operate in data stream model. Henzinger, Raghavan, and Rajagopalan [49] provided space lower bounds for concrete problems in the data stream model. These lower bounds are derived from results in communication complexity [56]. To understand the connection, observe that the memory used by any one-pass algorithm for a

function f, after seeing a prefix of the data stream, is lower bounded by the one-way communication required by two parties trying to compute f where the first party has access to the same prefix and the second party has access to the suffix of the stream that is yet to arrive. Henzinger et al. use this approach to provide lower bounds for problems such as frequent item counting, approximate median, and some graph problems.

Again based on communication complexity, Alon, Matias and Szegedy [5] provide almost tight lower bounds for computing the frequency moments. In particular they proved a lower bound of $\Omega(N)$ for estimating F_∞, the count of the most frequent item, where N is the domain size. At first glance this lower bound and a similar lower bound in Henzinger et al. [49] may seem to contradict the frequent item-set counting results of Manku and Motwani [60]. But note that the latter paper estimates the count of the most frequent item only if it exceeds ϵN. Such skewed distributions are common in practice, while the lower bounds are proven for pathological distributions where items have near-uniform frequency. This serves as a reminder that while it may be possible to prove strong space lower bounds for stream computations, considerations from applications sometimes enable us to circumvent the negative results.

Saks and Sun [73] provide space lower bounds for distance approximation between two vectors under the L_p norm, for $p > 3$, in the data stream model. Munro and Paterson [66] showed that any algorithm that computes quantiles exactly in p passes requires $\Omega(N^{1/p})$ space. Space lower bounds for maintaining simple statistics like count, sum, min/max, and number of distinct values under the sliding windows model can be found in the work of Datar et al. [26].

A general lower bound technique for sampling-based algorithms was presented by Bar-Yossef et al. [11]. It is useful for deriving space lower bounds for data stream algorithms that resort to oblivious sampling. It remains an interesting open problem to obtain similar general lower bound techniques for other classes of algorithms for the data stream model. We feel that techniques based on communication complexity results [56] will prove useful in this context.

6.7 Miscellaneous

In this section, we give a potpourri of algorithmic results for data streams.

Data Mining

Decision trees are another form of synopsis used for prediction. Domingos et al. [28, 29] have studied the problem of maintaining decision trees over data streams. Clustering is yet another way to summarize data. Consider the k-median formulation for clustering: Given n data points in a metric space, the objective is to choose k representative points, such that the sum of the errors over the n data points is minimized. The "error" for each data point is the distance from that point to the nearest of the k chosen representative points. Guha et al. [44] presented a single-pass algorithm for maintaining approximate k-medians (cluster centers) that uses $O(N^\epsilon)$ space for some $\epsilon < 1$ using $O(poly(\log N))$ amortized time per data element, to compute a constant factor approximation to the k-median problem. Their algorithm uses a divide-and-conquer approach which works as follows: Clustering proceeds hierarchically, where a small number (N^ϵ) of the original data points are clustered into k centers. These k-centers are weighted by the number of points that are closest to them in the local solution. When we get N^ϵ weighted cluster centers by clustering different sets, we cluster them into higher-level cluster centers, and so on.

Multiple Streams

Gibbons and Tirthapura [38] considered the problem of computing simple functions, such as the number of distinct elements, over unions of data stream. This is useful for applications that work in a distributed environment, where it is not feasible to send all the data to a central site for processing. It is important to

note that some of the techniques presented earlier, especially those that are based on *sketching*, are amenable to distributed computation over multiple streams.

Reductions of Streams

In a recent paper, Bar-Yossef, Kumar, and Sivakumar [12] introduce the notion of *reductions* in streaming algorithms. In order for the reductions to be efficient, one needs to employ *list-efficient* streaming algorithms. The idea behind list-efficient streaming algorithms is that instead of being presented one data item at a time, they are implicitly presented with a list of data items in a succinct form. If the algorithm can efficiently process the list in time that is a function of the *succinct representation size*, then it is said to be list-efficient. They develop some list-efficient algorithms and using the reduction paradigm address several interesting problems like computing frequency moments [5] (which includes the special case of counting the number of distinct elements) and counting the number of triangles in a graph presented as a stream. They also prove a space lower bound for the latter problem. To the best of our knowledge, besides this work and that of Henzinger et al. [49], there has been little work on graph problems in the streaming model. Such algorithms will likely be very useful for analyzing large graphical structures such as the web graph.

Property Testing

Feigenbaum et al. [34] introduced the concept of *streaming property testers* and *streaming spot checkers*. These are programs that make one pass over the data and using small space verify if the data satisfies certain property. They show that there are properties that are efficiently testable by a streaming-tester but not by a sampling-tester, and other problems for which the converse is true. They also present an efficient sampling-tester for testing the "groupedness" property of a sequence that use $O(\sqrt{N})$ samples, $O(\sqrt{N})$ space and $O(\sqrt{N}\log N)$ time. A sequence $\sigma_1, \ldots, \sigma_N$ is said to be *grouped* if $\sigma_i = \sigma_j$ and $i < k < j$ imply $\sigma_i = \sigma_k = \sigma_j$, i.e., for each type T, all occurrences of T are in a single contiguous run. Thus, groupedness is a natural relaxation of the sortedness property and is a natural property that one may desire in a massive streaming data set. The work discussed here illustrates that some properties are efficiently testable by sampling algorithms but not streaming algorithms.

Measuring Sortedness

Measuring the "sortedness" of a data stream could be useful in some applications; for example, it is useful in determining the choice of a sort algorithm for the underlying data. Ajtai et al. [3] have studied the problem of estimating the number of inversions (a measure of sortedness) in a permutation to within a factor ϵ, where the permutation is presented in a data stream model. They obtained an algorithm using $O(\log N \log\log N)$ space and $O(\log N)$ time per data element. They also prove an $\Omega(N)$ space lower bound for randomized exact computation, thus showing that approximation is essential.

7 Conclusion and Future Work

We have isolated a number of issues that arise when considering data management, query processing, and algorithmic problems in the new setting of continuous data streams. We proposed some initial solutions, described past and current work related to data streams, and suggested a general architecture for a Data Stream Management System (DSMS). At this point let us take a step back and consider some "meta-questions" with regard to the motivations and need for a new approach.

- Is there more to effective data stream systems than conventional database technology with enhanced support for streaming primitives such as triggers, temporal constructs, and data rate management?

- Is there a need for database researchers to develop fundamental and general-purpose models, algorithms, and systems for data streams? Perhaps it suffices to build ad hoc solutions for each specific application (network management, web monitoring, security, finance, sensors etc.).
- Are there any "killer apps" for data stream systems?

We believe that all three questions can be answered in the affirmative, although of course only time will tell.

Assuming positive answers to the "meta-questions" above, we see several fundamental aspects to the design of data stream systems, some of which we discussed in detail in this paper. One important general question is the interface provided by the DSMS. Our approach at Stanford is to extend SQL to support stream-oriented primitives, providing a purely declarative interface as in traditional database systems, although we also allow direct submission of query plans. In contrast, the Aurora project [16] provides a procedural "boxes and arrows" approach as the primary interface for the application developer.

Other fundamental issues discussed in the paper include timestamping and ordering, support for sliding window queries, and dealing effectively with blocking operators. A major open question, about which we had very little to say, is that of dealing with distributed streams. It does not make sense to redirect high-rate streams to a central location for query processing, so it becomes imperative to push some processing to the points of arrival of the distributed streams, raising a host of issues at every level of a DSMS [58]. Another issue we touched on only briefly in Section 4.5 is that of constraints over streams, and how they can be exploited in query processing. Finally, many systems questions remain open in query optimization, construction of synopses, resource management, approximate query processing, and the development of an appropriate and well-accepted benchmark for data stream systems.

From a purely theoretical perspective, perhaps the most interesting open question is that of defining extensions of relational operators to handle data stream constructs, and to study the resulting "stream algebra" and other properties of these extensions. Such a foundation is surely key to developing a general-purpose well-understood query processor for data streams.

Acknowledgements

We thank all the members of the Stanford STREAM research group for their contributions and feedback.

References

[1] 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.

[2] 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.

[3] M. Ajtai, T. Jayram, R. Kumar, and D. Sivakumar. Counting inversions in a data stream. *manuscript*, 2001.

[4] N. Alon, P. Gibbons, Y. Matias, and M. Szegedy. Tracking join and self-join sizes in limited storage. In *Proc. of the 1999 ACM Symp. on Principles of Database Systems*, pages 10–20, 1999.

[5] 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, 1996.

[6] 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, Sept. 2000.

[7] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, June 2002. Available at http://dbpubs.stanford.edu/pub/2001-49.

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

[9] B. Babcock, M. Datar, and R. Motwani. Sampling from a moving window over streaming data. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 633–634, 2002.

[10] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, Sept. 2001.

[11] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Sampling algorithms: Lower bounds and applications. In *Proc. of the 2001 Annual ACM Symp. on Theory of Computing*, pages 266–275, 2001.

[12] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 623–632, 2002.

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

[14] S. Bellamkonda, T. Borzkaya, B. Ghosh, A. Gupta, J. Haydu, S. Subramanian, and A. Witkowski. Analytic functions in oracle 8i. Available at http://www-db.stanford.edu/dbseminar/Archive /SpringY2000/speakers/agupta/paper.pdf.

[15] 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.

[16] D. Carney, U. Cetinternel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of dbms applications. Technical Report CS-02-01, Department of Computer Science, Brown University, Feb. 2002.

[17] 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, Sept. 2000.

[18] M. Charikar, S. Chaudhuri, R. Motwani, and V. Narasayya. Towards estimation error guarantees for distinct values. In *Proc. of the 2000 ACM Symp. on Principles of Database Systems*, pages 268–279, 2000.

[19] S. Chaudhuri, G. Das, and V. Narasayya. A robust, optimization-based approach for approximate answering of aggregate queries. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 295–306, May 2001.

[20] S. Chaudhuri and R. Motwani. On sampling and relational operators. *Bulletin of the Technical Committee on Data Engineering*, 22:35–40, 1999.

[21] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: How much is enough? In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 436–447, 1998.

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

[23] S. Chaudhuri and V. Narasayya. An efficient cost-driven index selection tool for microsoft sql server. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 146–155, 1997.

[24] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagraCQ: 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.

[25] 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, Aug. 2000.

[26] M. Datar, A. Gionis, P. Indyk, and R. Motwani. Maintaining stream statistics over sliding windows. In *Proc. of the 2002 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 635–644, 2002.

[27] A. Dobra, J. Gehrke, M. Garofalakis, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[28] 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, Aug. 2000.

[29] P. Domingos, G. Hulten, and L. Spencer. Mining time-changing data streams. In *Proc. of the 2001 ACM SIGKDD Intl. Conf. on Knowledge Discovery and Data Mining*, pages 97–106, 2001.

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

[31] 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.

[32] M. Fang, N. Shivakumar, H. Garcia-Molina, R. Motwani, and J. D. Ullman. Computing iceberg queries efficiently. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 299–310, 1998.

[33] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. An approximate l1-difference algorithm for massive data streams. In *Proc. of the 1999 Annual IEEE Symp. on Foundations of Computer Science*, pages 501–511, 1999.

[34] J. Feigenbaum, S. Kannan, M. Strauss, and M. Viswanathan. Testing and spot checking of data streams. In *Proc. of the 2000 Annual ACM-SIAM Symp. on Discrete Algorithms*, pages 165–174, 2000.

[35] P. Flajolet and G. Martin. Probabilistic counting. In *Proc. of the 1983 Annual IEEE Symp. on Foundations of Computer Science*, 1983.

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

[37] 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.

[38] P. Gibbons and S. Tirthapura. Estimating simple functions on the union of data streams. In *Proc. of the 2001 ACM Symp. on Parallel Algorithms and Architectures*, pages 281–291, 2001.

[39] A. Gilbert, S. Guha, P. Indyk, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Fast, small-space algorithms for approximate histogram maintenance. In *Proc. of the 2002 Annual ACM Symp. on Theory of Computing*, 2002.

[40] A. 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*, pages 79–88, 2001.

[41] 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, 2001.

[42] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, 2002.

[43] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 2001 Annual ACM Symp. on Theory of Computing*, pages 471–475, 2001.

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

[45] 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, Mar. 1996.

[46] P. Haas, J. Naughton, P. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of the 1995 Intl. Conf. on Very Large Data Bases*, pages 311–322, Sept. 1995.

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

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

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

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

[51] 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, Sept. 1999.

[52] iPolicy Networks home page. http://www.ipolicynetworks.com.

[53] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. 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.

[54] H. Jagadish, N. Koudas, S. Muthukrishnan, V. Poosala, K. Sevcik, and T. Suel. Optimal histograms with quality guarantees. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 275–286, 1998.

[55] H. Jagadish, I. 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.

[56] E. Kushlevitz and N. Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[57] 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, Aug. 1999.

[58] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, Feb. 2002. (To appear).

[59] S. Madden, J. Hellerstein, M. Shah, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002. (To appear).

[60] G. Manku and R. Motwani. Approximate frequency counts over streaming data. *manuscript*, 2002.

[61] G. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 426–435, June 1998.

[62] G. 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.

[63] Y. Matias, J. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 448–459, June 1998.

[64] Y. Matias, J. 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, Sept. 2000.

[65] R. Motwani and P. Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[66] J. Munro and M. Paterson. Selection and sorting with limited storage. *Theoretical Computer Science*, 12:315–323, 1980.

[67] 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.

[68] D. S. Parker, R. R. Muntz, and H. L. Chau. The Tangram stream query processing system. In *Proc. of the 1989 Intl. Conf. on Data Engineering*, pages 556–563, Feb. 1989.

[69] D. S. Parker, E. Simon, and P. Valduriez. SVP: A model capturing sets, lists, streams, and parallelism. In *Proc. of the 1992 Intl. Conf. on Very Large Data Bases*, pages 115–126, Aug. 1992.

[70] 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.

[71] D. Quass, A. Gupta, I. 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, Dec. 1996.

[72] V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering for interactive data processing. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, 1999.

[73] M. Saks and X. Sun. Space lower bounds for distance approximation in the data stream model. In *Proc. of the 2002 Annual ACM Symp. on Theory of Computing*, 2002.

[74] 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, Sept. 1991.

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

[76] 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.

[77] P. Seshadri, M. Livny, and R. Ramakrishnan. Seq: A model for sequence databases. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pages 232–239, Mar. 1995.

[78] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, pages 99–110, Sept. 1996.

[79] 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.

[80] R. Snodgrass and I. Ahn. A taxonomy of time in databases. In *Proc. of the 1985 ACM SIGMOD Intl. Conf. on Management of Data*, pages 236–245, 1985.

[81] S.-. Standard. On-line analytical processing (sql/olap). Available from http://www.ansi.org/, document #ISO/IEC 9075-2/Amd1:2001.

[82] Stanford Stream Data Management (STREAM) Project. http://www-db.stanford.edu/stream.

[83] 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, Sept. 1996.

[84] D. Terry, D. Goldberg, D. Nichols, and B. 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.

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

[86] P. Tucker, D. Maier, T. Sheard, and 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.

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

[88] T. Urhan and M. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[89] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, June 2002. (To appear).

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

[91] J. Vitter. External memory algorithms and datastructures. In J. Abello, editor, *External Memory Algorithms*, pages 1–18. Dimacs, 1999.

[92] J. 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.

[93] J. Vitter, M. Wang, and B. Iyer. Data cube approximation and histograms via wavelets. In *Proc. of the 1998 Intl. Conf. on Information and Knowledge Management*, Nov. 1998.

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

[95] Yahoo home page. http://www.yahoo.com.