Query Processing, Approximation, and Resource Management in a Data Stream Management System*

Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu,
Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, Rohit Varma
Stanford University
http://www-db.stanford.edu/stream

Abstract

This paper describes our ongoing work developing the *Stanford Stream Data Manager* (*STREAM*), a system for executing continuous queries over multiple continuous data streams. The STREAM system supports a declarative query language, and it copes with high data rates and query workloads by providing approximate answers when resources are limited. This paper describes specific contributions made so far and enumerates our next steps in developing a general-purpose Data Stream Management System.

1 Introduction

At Stanford we are building a *Data Stream Management System (DSMS)* that we call *STREAM*. The new challenges in building a DSMS instead of a traditional DBMS arise from two fundamental differences:

1. In addition to managing traditional stored data such as relations, a DSMS must handle multiple continuous, unbounded, possibly rapid and time-varying *data streams*.
2. Due to the continuous nature of the data, a DSMS typically supports long-running *continuous queries*, which are expected to produce answers in a continuous and timely fashion.

Our goal is to build and evaluate a general-purpose DSMS that supports a declarative query language and can cope with high data rates and large numbers of continuous queries. In addition to the obvious need for multi-query optimization and sophisticated scheduling to achieve high performance, we are targeting environments where data rates and query load may exceed available resources. In these cases our system is designed to provide *approximate answers* to continuous queries. Managing the interaction between resource availability and approximation is an important focus of our project. We are developing both static techniques and techniques for adapting as run-time conditions change.

This paper presents a snapshot of our language design, algorithms, system design, and system implementation efforts as of summer 2002. Clearly we are not presenting a finished prototype in any sense, e.g., our query language is designed but only a subset is implemented, and our approximation techniques have been identified but are not exploited fully by our resource allocation algorithms. However, there are a number of concrete contributions to report on at this point:

- An extension of SQL suitable for a general-purpose DSMS with a precisely-defined semantics (Section 2)
- Structure of query plans, accounting for plan sharing and approximation techniques (Section 3)
- A set of techniques for static and dynamic approximation to cope with limited resources (Section 4)
- An algorithm for allocating resources to queries (in a limited environment) that maximizes query result precision (Section 5.1)
- An algorithm for exploiting constraints on data streams to reduce memory overhead in query plan operators (Section 5.2)
- A near-optimal scheduling algorithm for reducing inter-operator queue sizes (Section 5.3)
- A software architecture designed for extensibility and for easy experimentation with DSMS query processing techniques (Section 6)

Some current limitations are:

- Our DSMS is centralized and based on the relational model. We believe that distributed query processing will be essential for many data stream applications, and we are designing our query processor with a migration to distributed processing in mind. We may eventually extend our system to handle XML data streams, but distribution has higher priority.
- We have done no significant work so far in query plan generation. Our system supports a subset of our ex-

* This work was supported by NSF Grant IIS-0118173, a Rambus Corporation Stanford Graduate Fellowship (Babcock), a Microsoft Graduate Fellowship (Datar), an NSF Graduate Fellowship (Olston), and grants (Motwani) from Microsoft, Veritas, and the Okawa Foundation.

tended query language with naive translation to a single plan. It also supports direct input of plans, including plan component sharing across multiple queries.

Due to space limitations this paper does not include a section dedicated to related work. We refer the reader to our recent survey paper [BBD+02], which provides extensive coverage of related work. We do make some comparisons to other work throughout this paper, particularly the *Aurora* project [CCC+02], which appears to be the closest in overall spirit to STREAM. However even these comparisons are narrow in scope and again we refer the reader to [BBD+02].

2 Query Language

The STREAM system allows direct input of query plans, similar to the Aurora approach [CCC+02] and described briefly in Section 6. However, the system also supports a declarative query language using an extended version of SQL. All queries are *continuous*, as opposed to the *one-time* queries supported by a standard DBMS. In this section we focus on the syntax and semantics of continuous queries in our extended SQL.

A DSMS must handle data from both continuous data streams and conventional relations:

- *Streams* have the notion of an arrival order, they are unbounded, and they are append-only. (Updates can be modeled in a stream using keys, but from the query-processor perspective we treat streams as append-only.) In addition to the continuous data streams that arrive at the DSMS, streams result from queries or subqueries that reference one or more streams (possibly with relations), do not perform aggregation over streams, and do not use streams as the target of negation (e.g., through `NOT EXISTS` or `EXCEPT`).
- *Relations* are unordered, and they support updates and deletions as well as insertions. In addition to relations stored by the DSMS, relations result from queries or subqueries that reference relations only, or that include aggregation or negation over streams.

We extend SQL by allowing the `FROM` clause of any query or subquery to contain relations, streams, or both. A stream in the `FROM` clause may be followed by an optional *sliding window specification*, enclosed in brackets, and an optional *sampling clause*.

As introduced in [BBD+02], in our language a window specification consists of an optional *partitioning clause*, a mandatory *window size*, and an optional *filtering predicate*. The partitioning clause partitions the data into several groups, computes a separate window for each group, and then merges the windows into a single result. It is syntactically analogous to a grouping clause, using the keywords `PARTITION BY` in place of `GROUP BY`. As in SQL-99 [UW02], windows are specified using either `ROWS` (e.g., "`ROWS 50 PRECEDING`") or `RANGE` (e.g., "`RANGE 15 MINUTES PRECEDING`"). The filtering predicate is specified using a standard SQL `WHERE` clause.

A sampling clause specifies that a random sample of the data elements from the stream should be used for query processing in place of the entire stream. The syntax of the sampling clause is a sampling rate followed by keyword `SAMPLE`. For example, "`1% SAMPLE`" indicates that, independently, each data element in the stream should be retained with probability 0.01 and discarded with probability 0.99.

2.1 Examples

Our example queries reference a stream `Requests` of requests to a web proxy server, each with four attributes: `client_id`, `domain`, `URL`, and `reqTime`.

The following query counts the number of requests for pages from the domain `stanford.edu` in the last day.

```
SELECT  COUNT(*)
FROM    Requests S [RANGE 1 DAY PRECEDING]
WHERE   S.domain = 'stanford.edu'
```

The semantics of providing continuous answers to this query are covered in Section 2.4.

The following query counts how many page requests were for pages served by Stanford's CS department web server, considering only each client's 10 most recent page requests from the domain `stanford.edu`. This query makes use of a partitioning clause and also brings out the distinction between predicates applied before determining the sliding window cutoffs and predicates applied after windowing.

```
SELECT  COUNT(*)
FROM    Requests S
          [PARTITION BY S.client_id
           ROWS 10 PRECEDING
           WHERE S.domain = 'stanford.edu']
WHERE   S.URL LIKE 'http://cs.stanford.edu/%'
```

Our final example references a stored relation `Domains` that classifies domains by the primary type of web content they serve. This query counts the number of requests for pages from "commerce" domains out of the last $10,000$ requests for pages from domains that have been classified. A 10% sample of the `Requests` stream is used for the query. Notice that the stream of requests must be joined with the Domains relation (resulting in a stream labeled T) before applying the sliding window.

```
SELECT  COUNT(*)
FROM
```

```
    (SELECT R.class
     FROM Requests S 10% SAMPLE, Domains R
     WHERE S.domain = R.domain) T
    [ROWS 10000 PRECEDING]
WHERE   T.class = 'commerce'
```

2.2 Stream Ordering and Timestamps

The sliding windows in our query language require that streams have an ordering for `ROWS` window specifications, and some type of timestamp for `RANGE` windows. In our current language design we assume that each stream tuple has a timestamp, which also implies stream ordering.

In many cases, the arrival times of stream elements at the DSMS can be used as timestamps for input streams and provide sufficient accuracy. However, sometimes it is preferable to use explicit timestamps provided as part of the data stream (generated by the stream source or perhaps by an application-specific timestamping feature). Our `CREATE STREAM` statement, which is used to register an input stream with the system, allows the optional designation of one attribute as `TIMESTAMP` (e.g., we might so designate attribute `reqTime` in the example schema of Section 2.1). Currently we require this attribute to be of type `DATETIME` and we assume its values correspond to actual clock times. Offering more flexibility for explicit timestamps complicates our semantics considerably, but is planned for future work.

By definition, arrival-based timestamps guarantee that stream tuples arrive in timestamp order. In our semantics we assume that streams with explicit timestamps also arrive in timestamp order, modulo a *scrambling bound* B: if a tuple with explicit timestamp τ arrives on stream S, then no tuple with timestamp greater than $\tau - B$ can arrive later on S. We assume B is global to the DSMS but B could easily be stream-specific, declared along with the `TIMESTAMP` attribute.

Timestamps for streams generated by subqueries are defined in Section 2.4. Note that the related areas of *temporal* and *sequence* query languages [SLR96, Soo91] can capture most aspects of the timestamps and window specifications in our language. Those languages are considerably more expressive than our language, and we feel they are "overkill" in typical data stream environments.

2.3 Inactive and Weighted Queries

Two dynamic properties of queries are controlled through our administrative interface discussed in Section 6. One property is whether the query is *active or inactive*, and the other is the *weight* assigned to the query. When a query is inactive, the system may not maintain the answer to the query as new data arrives. However, because an inactive query may be activated at any time, its presence serves as a hint to the system that may influence decisions about query plans and resource allocation (Sections 3–5).

Queries may be assigned weights indicating their relative importance. These weights are taken into account by the system when it is forced to provide approximate answers due to resource limitations. Given a choice between introducing error into the answers of two queries, the system will attempt to provide more precision for the query with higher weight. Weights might also influence scheduling decisions, although we have not yet explored weighted scheduling. Note that inactive queries may be thought of as queries with negligible weight.

2.4 Formal Semantics

One of our contributions is to provide a precise semantics of continuous queries over multiple data streams with user-specified sliding windows. We also have developed an algebra corresponding to our query language and semantics, although it is not presented here due to space limitations. We assume basic knowledge of SQL semantics and focus on three new aspects in our language: sliding windows, mixing relations and streams, and continuous (as opposed to one-time) answers. The semantics of the `SAMPLE` operator are straightforward and not discussed further.

As will be seen, we specify "conservative" semantics, requiring global timestamps, exact coordination among all components of a query, and never allowing answers on a stream until all earlier answers are guaranteed to have been produced. We believe that our conservative semantics serves as an important theoretical baseline, but it may be difficult to implement efficiently for the full generality of the query language. Many applications may be satisfied with a more relaxed or "best-effort" semantics (particularly in terms of timing), which we hope to formalize as future work.

We begin with some definitions. The *current timestamp* τ for an input stream S, denoted $C(S)$, is defined as the largest τ' such that no tuple with timestamp less than τ' can arrive on S. For a stream S that uses arrival-based timestamps, $C(S)$ at any given time is simply the value of the system clock at that time. For a stream S that uses explicit timestamps, $C(S)$ is $\tau' - B$, where τ' is the largest timestamp of any tuple that has arrived on S and B is the *scrambling bound* discussed in Section 2.2. If S uses explicit timestamps but is empty, then we define $C(S)$ to be the earliest possible timestamp.

The current timestamp $C(Q)$ for a query Q is the smallest $C(S)$ over all streams S referenced by Q. $C(Q)$ is computed globally for a query, so the same value is used in defining the semantics for all subqueries in Q.

The *relevant data* for a query Q at time τ consists of all stream tuples with timestamps less than or equal to τ,

plus the contents of all relations at the time when $C(Q)$ first became greater than or equal to τ. The *active set* for a query or subquery Q at time τ consists of all tuples that remain after the window specifications in Q are applied to the relevant data of Q at τ. The semantics of `PARTITION BY` are straightforward. When a window size is expressed as a `RANGE`, the offset is computed from time τ. When a window size is expressed using `ROWS`, the offset is computed from the last tuple in the stream with timestamp less than or equal to τ (i.e., from the end of the *relevant* portion of the stream). Define $A(Q, \tau)$ to be the answer that results from evaluating Q using standard relational semantics over the active set of Q at time τ.

The semantics for continuous queries are slightly different depending on whether the query produces a relation (queries involving only relations, or involving aggregation or negation over streams) or produces a stream (queries referencing streams without aggregation or negation):

- When a continuous query Q produces a relation, its answer at any instant is equal to $A(Q, C(Q))$, i.e., the result of the query considering the most recent consistent snapshot of Q's data.
- When Q produces a stream, that stream consists of $\bigcup_{\tau \leq C(Q)} A(Q, \tau)$, where a tuple appears in the result of the union the maximum number of times it appears in any branch of the union. Effectively, under this semantics tuples produced on the result stream can be computed as input streams arrive using the relational data at the time of arrival, and additional result tuples may be produced if relations are updated. If a tuple t appears in the result stream only once, then its timestamp in the answer to Q is the smallest τ such that $t \in A(Q, \tau)$, i.e., the time at which t first appears in the result stream. The generalization to multiple instances of t is straightforward.

This semantics is applied to subqueries (which also may produce streams or relations) analogously, bearing in mind that the value of $C(Q)$ is global to an entire query.

3 Query Plans

This section describes the basic query processing architecture of the STREAM system. Queries are registered with the system and execute continuously as new data arrives. For now let us assume that a separate query plan is used for each continuous query, although sharing of plan components is very important and will be discussed in Section 3.2. We also assume that queries are registered before their input streams begin producing data, although clearly we must address the issue of adding queries over existing (perhaps partially discarded or archived) data streams.

It is worth a short digression to highlight a basic difference between our approach and that of Aurora [CCC+02]. Aurora uses one "mega" query plan performing all computation of interest to all users. Adding a query consists of directly augmenting portions of the current mega-plan, and conversely for deleting a query. In STREAM, queries are independent units that logically generate separate plans, although plans may be combined by the system and ultimately could result in an Aurora-like mega-plan.

To date we have implemented only a subset of the language presented in Section 2, primarily omitting support for aggregation, certain subqueries, certain constructs in window specifications, and of course many esoteric features of standard SQL. A number of important implementation issues have not yet been dealt with, such as monitoring the system clock for time-based windows, encoding relations as streams during query processing, and strict output semantics (Section 2.4). Nevertheless, our basic query processing architecture is in place and functional. In this section we highlight its features but do not go into detail about individual query operators since many of them are analogous to a traditional DBMS.

A query plan in our system runs continuously and is composed of three different types of components:

- Query *operators*, similar to a traditional DBMS. Each operator reads a stream of tuples from a set of input queues, processes the tuples based on its semantics, and writes its output tuples into a single output queue.
- Inter-operator *queues*, also similar to the approach taken by some traditional DBMS's. Queues connect different operators and define the paths along which tuples flow as they are being processed.
- *Synopses*, used to maintain state associated with operators and discussed in more detail next.

A synopsis summarizes the tuples seen so far at some intermediate operator in a running query plan, as needed for future evaluation of that operator. For example, for full precision a join operator must remember all the tuples it has seen so far on each of its input streams, so it maintains one synopsis for each (similar to a *symmetric hash join* [WA91]). On the other hand, simple filter operators, such as selection and duplicate-preserving projection, do not require a synopsis since they need not maintain state.

For many queries, synopsis sizes grow without bound if full precision is expected in the query result [ABB+02]. Thus, an important feature to support is synopses that use some kind of summarization technique to limit their size [GGR02], e.g., *fixed-size hash tables*, *sliding windows*, *reservoir samples*, *quantile estimates*,

and *histograms*. Of course limited-size synopses may produce approximate operator results, further discussed in Sections 4 and 5.

Although operators and synopses are closely coupled in query plans, we have carefully separated their implementation and provide generic interfaces for both. This approach allows us to couple any operator type with any synopsis type, and it also paves the way for operator and synopsis sharing. The generic methods of the `Operator` class are:

- `create`, with parameters specifying the input queues, output queue, and initial memory allocation.
- `changeMem`, with a parameter indicating a dynamic decrease or increase in allocated memory.
- `run`, with a parameter indicating how much work the operator should perform before returning control to the scheduler (see Section 5.3).

The generic methods of the `Synopsis` class are:

- `create`, with a parameter specifying an initial memory allocation.
- `changeMem`, with a parameter indicating a dynamic decrease or increase in allocated memory.
- `insert` and `delete`, with a parameter indicating the data element to be inserted into or deleted from the synopsis.
- `query`, whose parameters and behavior depend on the synopsis type. For example, in a hash-table synopsis this method might look for matching tuples with a particular key value, while for a sliding-window synopsis this method might support a full window scan.

So far in our system we have focused on sliding-window synopses, which keep a summary of the last w tuples of some intermediate stream. Sliding-window synopses are used for approximation (Section 4), in which case w is determined by the tuple size and memory allocation M. They also are used to provide precise results for the `ROWS`-based window specifications in our query language (Section 2), in which case the memory requirement M is determined by the tuple size and w.

3.1 Example

Figure 3.1 illustrates plans for two queries, Q_1 and Q_2. Together the plans contain three operators O_1–O_3, four synopses s_1–s_4 (two per join operator), and four queues q_1–q_4. Query Q_1 is a selection over a join of two streams R and S. Query Q_2 is a join of three streams, R, S, and T. The two plans share a subplan joining streams R and S by sharing its output queue q_3. Plan and queue sharing is discussed in Section 3.2. Execution of query operators

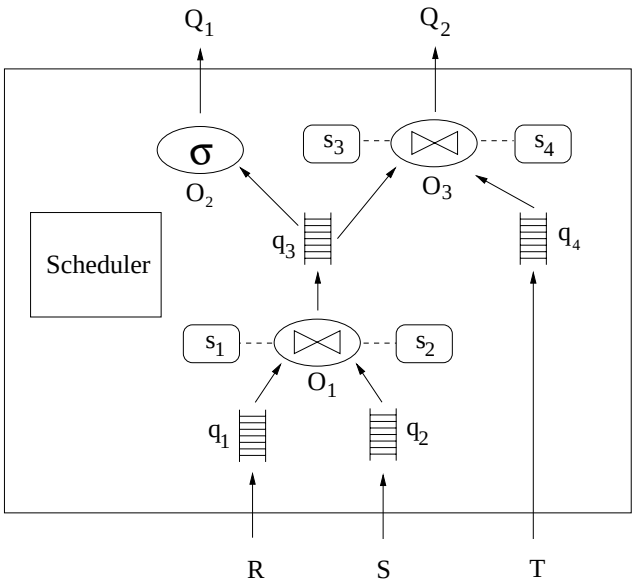

Figure 1: Plans for queries Q_1,Q_2 over streams R,S,T.

is controlled by a global *scheduler*. When an operator O is scheduled, control passes to O for a period currently determined by number of tuples processed, although we may later incorporate timeslice-based scheduling. Section 5.3 considers different scheduling algorithms and their impact on resource utilization.

3.2 Resource Sharing in Query Plans

As illustrated in Figure 3.1, when continuous queries contain common subexpressions we can share resources and computation within their query plans, similar to multi-query optimization and processing in a traditional DBMS [Sel88]. We have not yet focused on resource sharing in our work—we have established a query plan architecture that enables sharing, and we can combine plans that have exact matching subexpressions. However, several important topics are yet to be addressed:

- For now we are considering resource sharing and approximation separately. That is, we do not introduce sharing that intrinsically introduces approximate query results, such as merging subexpressions with different window sizes, sampling rates, or filters. Doing so may be a very effective technique when resources are limited, but we have not yet explored it in sufficient depth to report here.
- Our techniques so far are based on exact common subexpressions. Detecting and exploiting subexpression containment is a topic of future work that poses some novel challenges due to window specifications, timestamps and ordering, and sampling in our query language.

The implementation of a shared queue (e.g., q_3 in Figure 3.1) maintains a pointer to the first unread tuple for each operator that reads from the queue, and it discards tuples once they have been read by all parent operators. Currently multiple queries accessing the same incoming base data stream S "share" S as a common subexpression, although we may decide ultimately that input data streams should be treated separately from common subexpressions.

The number of tuples in a shared queue at any time depends on the rate at which tuples are added to the queue, and the rate at which the slowest parent operator consumes the tuples. If two queries with a common subexpression produce parent operators with very different consumption rates, then it may be preferable not to use a shared subplan. As an example, consider a queue q output from a join operator J, and suppose J is very unselective so it produces nearly the cross-product of its inputs. If J's parent P_1 in one query is a "heavy consumer," then our scheduling algorithm (Section 5.3) is likely to schedule J frequently in order to produce tuples for P_1 to consume. If J's parent P_2 in another query is a "light consumer," then the scheduler will schedule J less frequently so tuples don't proliferate in q. In this situation it may not be beneficial for P_1 and P_2 to share a common subplan rooted in J.

We have shown formally that although subplan sharing may be suboptimal in the case of common subexpressions with joins, for common subexpressions without joins sharing always is preferable. Details are beyond the scope of this paper.

When several operators read from the same queue, and when more than one of those operators builds some kind of synopsis, then it may be beneficial to introduce *synopsis sharing* in addition to *subplan sharing*. A number of interesting issues arise, most of which we have not yet addressed:

- Which operator is responsible for managing the shared synopsis (e.g., allocating memory, inserting tuples)?
- If the synopses required by the different operators are not of identical types or sizes, is there a theory of "synopsis subsumption" (and synopsis overlap) that we can rely on?
- If the synopses are identical, how do we cope with the different rates at which operators may "consume" data in the synopses?

Clearly we have much work to do in the area of resource sharing. Note again that the issue of automatic resource sharing is less crucial in a system like Aurora, where resource sharing is primarily programmed by users when they augment the current mega-plan.

4 Approximations

It is our supposition that the combination of:

- multiple unbounded and possibly rapid incoming data streams,
- multiple complex continuous queries with timeliness requirements, and
- finite computation and memory resources

yields an environment where eventually the system will not be able to provide continuous and timely exact answers to all registered queries. Our goal is to build a system that, under these circumstances, degrades gracefully to *approximate* query answers. Furthermore, the system should maximize the precision of query answers based on the available resources. In this section we discuss approximation techniques, and in Section 5 we discuss the close relationship between approximation and resource management.

4.1 Static Approximation

In static approximation, queries are modified when they are submitted to the system so that they use fewer resources at execution time. The advantages of static approximation over dynamic approximation (discussed in Section 4.2) are:

1. Assuming the statically optimized query is executed precisely by the system, the user is guaranteed certain query behavior. A user might even participate in the process of static approximation, guiding or approving the system's query modifications.
2. Adaptive approximation techniques and continuous monitoring of system activity are not required—the query is modified once, before it begins execution.

The two static approximation techniques we consider are *window reduction* and *sampling rate reduction*.

4.1.1 Window Reduction

Our query language includes a windowing clause for specifying sliding windows on streams or on subqueries producing streams (Section 2). By decreasing the size of a window, or introducing a window where none was specified originally, both memory and computation requirements can be reduced.[1] In fact, several proposals for stream query languages automatically introduce windows in all joins, sometimes referred to as *band joins*, in order to bound the resource requirement, e.g., [CCC+02, CF02, HF+00, MSHR02, VN02].

[1] Throughout the paper we refer to the resource required for state (synopses and queues) in query plans as "memory." Disk also could be used, although in that case we might want to treat I/O as a separate resource given its different performance characteristics, as in Aurora [CCC+02].

Suppose W is an operator that incorporates a window specification, most commonly a windowed join. Reducing W's window size not only affects the resources used by W, but it can have a ripple effect that propagates up the operator tree—in general a smaller window results in fewer tuples to be processed by the remainder of the query plan. However, there are at least two cases where we need to be careful:

- If W is a duplicate-elimination operator, then shrinking W's window can actually increase its output rate.
- If W is part of the right-hand subtree of a negation construct (e.g., `NOT EXISTS` or `EXCEPT`), then reducing the size of W's output may have the effect of increasing output further up the query plan.

Fortunately, these "bad" cases can be detected statically at query modification time, so the system can avoid introducing or shrinking windows in these situations.

4.1.2 Sampling Rate Reduction

Analogous to shrinking window sizes, we can reduce the sampling rate when a `SAMPLE` clause (Section 2) is applied to a stream or to a subquery producing a stream. We can also introduce `SAMPLE` clauses where not present in the original query. Although changing the sampling rate at an operator O will not reduce the resource requirements of O, it will reduce the output rate. We can also take an existing sample operator and push it down the query plan. However, we must be careful to ensure that we don't introduce unbiased sampling when we do so, especially in the presence of joins as discussed in [CMN99].

4.2 Dynamic Approximation

In our second and more challenging approach, *dynamic approximation*, queries are unchanged, but the system may not always provide precise query answers. Dynamic approximation has some important advantages over static approximation:

- The level of approximation can vary with fluctuations in data rates and distributions, query workload, and resource availability. In "times of plenty," when loads are low and resources are high, queries can be answered precisely, with approximation occurring only when absolutely necessary.
- Approximation can occur at the plan operator level, and decisions can be made based on the global set of (possibly shared) query plans running in the system.

Of course a significant challenge from the usability perspective is conveying to users or applications at any given time what kind of approximation is being performed on their queries, and some applications simply may not want to cope with variable and unpredictable accuracy. We are considering augmenting our query language so users can specify tolerable imprecision (e.g., ranges of acceptable window sizes, or ranges of sampling rates), which offers a middle ground between static and dynamic approximation.

The three dynamic approximation techniques we consider are *synopsis compression*, which is roughly analogous to window reduction in Section 4.1.1, *sampling*, which is analogous to *sampling rate reduction* in Section 4.1.2, and *load shedding*.

4.2.1 Synopsis Compression

One technique for reducing the memory overhead of a query plan is to reduce synopsis sizes at one or more operators. Incorporating a sliding window into a synopsis where no window is being used, or shrinking the existing window, typically shrinks the synopsis. Doing so is analogous to introducing windows or statically reducing window sizes through query modification (Section 4.1.1). Note that if plan sharing is in place then modifying a single window dynamically may affect multiple queries, and if sophisticated synopsis-sharing algorithms are being used then different queries may be affected in different ways.

There are other methods for reducing synopsis size, including maintaining a sample of the intended synopsis content (which is not always equivalent to inserting a sample operator into the query plan), using *histograms* [TGIK02] or compressed *wavelets* [GG02] when the synopsis is used for aggregation or even for a join [CGRS02], and using *Bloom filters* [Blo70] for duplicate elimination, set difference, or set intersection.

All of these techniques share the property that memory use is flexible, and it can be traded against precision statically or on-the-fly. Some of the techniques provide error guarantees, e.g., [GG02], however we have not solved the general problem of conveying accuracy to users dynamically.

4.2.2 Sampling and Load Shedding

The two primary consumers of memory in our query plans are synopses and queues (recall Section 3). In the previous subsection we discussed approximation techniques that reduce synopsis sizes (which may as a side-effect reduce queue sizes). In this section we mention approximation techniques that reduce queue sizes (which may as a side-effect reduce synopsis sizes).

One technique is to introduce one or more *sample* operators into the query plan, or to reduce the sampling rate at existing operators. This approach is the dynamic analogue of introducing sampling or statically reducing a sampling rate through query modification (Section 4.1.1), although again we note that when plan shar-

ing is in place one sampling rate may affect multiple queries.

We can also simply drop tuples from queues when they grow too large, a technique sometimes referred to as *load shedding* [CCC+02]. Load shedding at queues, which typically drops chunks of tuples at a time, differs from sampling at operators, which eliminates tuples probabilistically. Both are effective techniques for reducing queue sizes. While sampling may be more "unbiased," load shedding may be easier to implement and to make decisions about dynamically.

5 Resource Management

Effective resource management is a key component of a data stream management system, and it is a specific focus of our project. There are a number of relevant resources in a DSMS: memory, computation, I/O if disk is used, and network bandwidth in a distributed DSMS. We focus primarily on memory consumed by query plan synopses and queues, although some of our techniques can be applied readily to other resources. Furthermore, in many cases reducing memory overhead has a natural side-effect of reducing other resource requirements as well.

We motivated the need for sophisticated memory management in Section 4, where we saw that when resources are limited we can reduce memory overhead in a variety of ways that all result in approximate query answers. When conditions such as data rates and query load change, the availability and best use of resources change also. Our overall goal is to maximize query precision by making the best use of available resources, and ultimately to have the capability of doing so dynamically and adaptively. Solving the overall problem (which further includes *inactive* and *weighted* queries as discussed in Section 2.3) involves a huge number of variables, and certainly is intractable in the general case. To date we have developed:

1. An algorithm for allocating memory to a query plan statically, maximizing result precision under a relatively simple precision model. This work is described in Section 5.1.
2. An algorithm for incorporating known constraints on input data streams to reduce synopsis sizes without compromising precision. This work is described in Section 5.2.
3. An algorithm for operator scheduling that minimizes queue sizes. This work is described in Section 5.3.

In comparison with other systems for processing queries over data streams, both the *Telegraph* [HF+00] and *Niagara* [CDTW00] projects do consider resource management (largely dynamic in the case of Telegraph and static in the case of Niagara), but not in the context of providing approximate query answers when available resources are insufficient. An important contribution was made in Aurora [CCC+02] with the introduction of "QoS graphs" that capture tradeoffs among precision, response time, resource usage, and usefulness to the application. However, in Aurora approximation currently appears to occur solely through *drop-boxes* that perform load shedding as described in Section 4.2.2.

5.1 Static Resource Allocation

Our work so far in static resource allocation addresses a restricted scenario but provides a solid basis for more general algorithms. Consider one query, and assume the query plan is provided or the system has already selected a "best" query plan. Plans are expressed using the operators of relational algebra (including set difference, which as usual introduces some challenges). We use a simple model of precision that measures the accuracy of a query result as its average rate of *false positives* and *false negatives*.

We give a brief overview of our approach and algorithm. Let us assume that each operator in a query plan has a known function from resources to precision, typically based on one or more of the approximation methods that reduce synopsis sizes discussed in Section 4. Further suppose that we know how to compute precision for a plan from precision for its constituent operators—we will discuss this computation shortly. Finally, assume we have fixed total resources. (Resources can be of any type as long as they can be expressed and allocated numerically.) Then our goal of allocating resources to operators in order to maximize overall query precision can be expressed as a nonlinear optimization problem, which we currently solve using a packaged numerical iterative improvement solver, although in the long run scalability of the packaged solver may become an issue.

In the language handled by our static resource allocation algorithm, all operators and plans produce a stream of output tuples, although ordering is not relevant for the operators we consider. The precision of a stream—either a result stream or a stream within a query plan—is defined by (*FP*,*FN*), where $FP \in [0,1]$ and $FN \in [0,1]$. *FP* captures the false positive rate: the probability that an output stream tuple is incorrect. *FN* captures the false negative rate: the probability, for each correct output stream tuple, that there is another correct tuple that was missed. (*FP*,*FN*) also can denote the precision of an operator, with the interpretation that the operator produces a result stream with (*FP*,*FN*) precision when given input(s) with (0,0) (exact) precision. In all cases, *FP* and *FN* denote expected (mean) precision values over time.

We assume that all plan operators map allocated re-

sources to precision specifications (*FP*,*FN*). Currently we do not depend on monotonicity—i.e., we do not assume that more resources result in lower values for *FP* and *FN*—although we can expect monotonicity to hold and are investigating whether it may help us in our numerical solver. We have devised (and shown to be correct, both mathematically and empirically) fairly complex formulas that, for each operator type, compute output stream precision (*FP*,*FN*) values from the precision of the input streams and the precision of the operator itself.

We assume the base input streams to a query have exact precision, i.e., (0,0). We apply our formulas bottom-up to the query plan, feeding the result to the numerical solver which produces the optimal resource allocation.

The next steps in this work are to incorporate variance into our precision model, to extend the model to include value-based precision so we can handle operators such as aggregation, and eventually to couple plan generation with resource allocation.

5.2 Exploiting Constraints Over Data Streams

So far we have not discussed exploiting data or arrival characteristics of input streams during query processing. Certainly we must be able to handle arbitrary streams, but when we have additional information about streams, either by gathering statistics over time or through constraint specifications at stream-registration time, we can use this information to reduce resource requirements without compromising query result precision. (An alternate and more dynamic technique is for the streams to contain *punctuations*, which specify run-time constraints that also can be used to reduce resource requirements; see [TMSF].)

Our main contribution to date has been to identify several types of constraints over data streams, and for each constraint type to specify an "adherence parameter" that captures how closely a given stream or set of streams adheres to a constraint of that type. We have developed query plan construction and execution algorithms that take stream constraints into account in order to reduce synopsis sizes at query operators, while still producing precise output streams. Using our algorithm, the closer the streams adhere to the specified constraints at run-time, the smaller the required synopses. We have implemented our algorithm in a stand-alone query processor in order to run experiments, and our next step is to incorporate it into the STREAM prototype.

As a simple example, consider a continuous query that joins a stream `Orders` (hereafter O) with a stream `Fulfillments` (hereafter F) based on `orderID` and `itemID` (orders may be fulfilled in multiple pieces), perhaps to monitor average fulfillment delays. In the general case, answering this query precisely requires synopses of unbounded size [ABB+02]. However, if we know that all tuples for a given orderID and itemID arrive on O before the corresponding tuples arrive on F, then we need not maintain a join synopsis for the F operand at all. Furthermore, if O tuples arrive clustered by orderID, then we need only save O tuples for a given orderID until the next orderID is seen.

In practice, constraints may not be adhered to by data streams strictly, even if they "usually" hold. For example, we may expect tuples on stream O to be clustered by orderID within a tolerance parameter k: no more than k tuples with a different orderID appear between two tuples with same orderID. Similarly, due to network delays a tuple for a given orderID and itemID may arrive on F before the corresponding tuple arrives on O, but we may be able to bound the time delay with a constant k. These constants are the "adherence parameters" discussed earlier, and it should be clear that the smaller the value of k, the smaller the necessary synopses.

The constraints considered in our work are *many-one join* and *referential integrity* constraints between two streams, and *unique-value*, *clustered-arrival*, and *ordered-arrival* constraints on individual streams. Our algorithm accepts select-project-join queries over streams with arbitrary constraints, and it produces a query plan that exploits constraints to reduce synopsis sizes without compromising precision. The details are extensive and beyond the scope of this paper.

5.3 Scheduling

Query plans are executed via a *global scheduler*, whose job it is to call the `run` methods of query plan operators (Section 3) in order to make progress moving tuples through query plans and producing query results. Our initial scheduler uses a simple round-robin scheme, and a single granularity for the `run` operator expressed as the maximum number of tuples to be consumed from the operator's input queue before relinquishing control. This simple scheduler gives us a functioning system but clearly is far from optimal for most sets of query plans.

There are many possible objectives for the scheduler, including stream-based variations of response time, throughput, and (weighted) fairness among queries. For our first cut at a more "intelligent" scheduler, we have decided to focus on minimizing intermediate queue sizes, in keeping with our general project goal of coping with limited resources. Furthermore, the granularity of scheduling we consider is a "time unit," during which some operators may be able to consume more input tuples than others. We have not considered parallelism in our scheduling algorithms.

Consider the following very simple example. Suppose we have a query plan with two unary operators: O_1 operates on input queue q_1, writing its results to queue q_2

which is input to operator O_2. Suppose O_1 takes one time unit to operate on a batch of n tuples from q_1, and it has 20% selectivity, i.e., it produces $n/5$ tuples in q_2 when it consumes n tuples from q_1. (Time units and batches of n input tuples simplify exposition; their actual values are not relevant to the overall reasoning in our example.) Operator O_2 takes one time unit to operate on $n/5$ tuples, and it produces no tuples on its output queue. Let us assume that over time the average arrival rate of tuples on q_1 is no more than n tuples per two time units, so all tuples can be processed and queues will not grow without bound. (If queues do grow without bound, eventually some form of load shedding must occur, as discussed in Section 4.2.2). However, tuple arrivals may be bursty.

Here are two possible scheduling strategies:

1. Tuples are processed to completion in the order they arrive on q_1. Each batch of n tuples in q_1 is processed by O_1 and then O_2 based on arrival time, consuming two time units overall.
2. If there is a batch of n tuples in q_1, then O_1 operates on them using one time unit, producing $n/5$ new tuples in q_2. Otherwise, if there are any tuples in q_2 then up to $n/5$ of these tuples are operated on by O_2, consuming one time unit.

Suppose we have the following arrival pattern: $2n$ tuples arrive on q_1 at time $\tau = 0$, followed by no tuples at time $\tau = 1$ and n tuples each at times $\tau = 2$ and $\tau = 3$. The following table shows the total size of queues q_1 and q_2 under the two scheduling strategies, where each table entry is a multiplier for n.

Time τ	0	1	2	3	4	5	6	7	8
Strat. 1	2	1.2	2	2.2	2	1.2	1	.2	0
Strat. 2	2	1.2	1.4	1.6	.8	.6	.4	.2	0

In this example, both strategies finish at the 8th time step, and Strategy 2 is clearly preferable in terms of memory overhead.

We have a designed a scheduling policy that provably has close-to-optimal queue size overhead, and is based on the general property observed in our example: greedily schedule the operator that "consumes" the largest number of tuples per time unit and is the most selective (i.e., "produces" the fewest tuples). Two additional considerations are reflected in the algorithm:

- We favor operators with full batches of tuples in their input queues over higher-priority (i.e., more selective or more consuming) operators with underfull input queues, so that operators can make full use of their timeslices and tuples continue to move through the query plan.
- A high-priority operator may be underutilized if the operators feeding it are low priority, so we also consider *chains* of operators within a plan when we make scheduling decisions. However, we do not schedule chains as a unit, a strategy taken by Aurora's *train scheduling* algorithm [CCC+02]. Aurora's objective is to improve throughput by reducing context-switching between operators, batching the processing of tuples through operators, and reducing I/O overhead since their inter-operator queues may be written to disk. So far we have considered minimizing memory-based queue sizes as our only scheduling objective.

Details of our scheduling algorithm and the proof of its near-optimality are fairly involved and not presented due to space limitations.

Our algorithm achieves queue size minimization, but we may pay in increased time to initial results. In our example above, although both strategies finish processing tuples at the same time and for simplicity the plan produces an empty answer, it should be clear that Strategy 1 generally has the potential to produce initial results more quickly than Strategy 2. Important next steps are to incorporate response time and (weighted) fairness across queries into our scheduling algorithm, as well as introducing flexible timeslices and taking the cost of context-switching into account.

5.4 Resource Management: Summary and Discussion

Recall that our overall goal is to manage resources carefully, and to perform approximation in the face of resource limitations in a flexible, usable, and principled manner. We want solutions that perform static approximation based on predictable resource availability (Sections 4.1 and 5.1), and we want alternate solutions that perform dynamic approximation and resource allocation to maximize the use of available resources and adapt to changes in data rates and query loads (Section 4.2). Although we have solved some pieces of the problem in limited environments, many important challenges lie ahead; for example:

- We need a means of monitoring synopsis and queue sizes and determining when dynamic reduction measures (e.g., window size reduction, load shedding) should kick in.
- Even if we have a good algorithm for initial allocation of memory to synopses and queues, we need a reallocation algorithm to handle the inevitable changes in data rates and distributions.
- The ability to add, delete, activate, and deactivate queries at any time forces all resource allocation

schemes, including static ones, to provide a means of making incremental changes.

It is clear to us that no system will provide a completely general and optimal solution to the problems posed here, particularly in the dynamic case. However, we will continue to chip away at important pieces of the problem, with (we hope) the end result being a cohesive system that achieves good performance and usable, understandable functionality.

6 Implementation and Interfaces

Since we are developing the STREAM prototype from scratch we have the opportunity to create an extensible and flexible software architecture, and to provide useful interfaces for system developers and "power users" to visualize and influence system behavior. Here we cover three features of our design: our generic entities, our encoding of query plans, and the system interface. Collectively, these features form the start of a comprehensive "workbench" we envision for programming and interacting with the DSMS.

6.1 Entities and Control Tables

In the implementation of our system, operators, queues, and synopses all are subclasses of a generic `Entity` class. Each entity has a table of attribute-values pairs called its *Control Table* (CT for short), and each entity exports an interface to query and update its CT. The CT serves two purposes in our system so far. First, some CT attributes are used to dynamically control the behavior of an entity. For example, the amount of memory used by a synopsis S can be controlled by updating the value of attribute `Memory` in S's control table. Second, some CT attributes are used to collect statistics about entity behavior. For example, the number of tuples that have passed through a queue q is stored in attribute `Count` of q's control table. These statistics are available for resource management and for user-level system monitoring. It is a simple matter to add new attributes to a CT as needs arise, offering convenient extensibility.

6.2 Query Plans

We want to be able to create, view, understand, and manually edit query plans in order to explore various aspects of query optimization. Our query plans are implemented as networks of *entities* as described in the previous section, stored in main memory. A graphical interface is provided for creating and viewing plans, and for adjusting attributes of operators, queues, and synopses. The interface was very easy to implement based on our generic CT structure, since the same code could be used for most query plan elements.

Query plans may be viewed and edited even as queries are running. Currently we do not support viewing of data moving through query plans, although we certainly are planning this feature for the future. Since continuous queries in a DSMS should be persistent, main-memory plan structures are mirrored in XML files, which were easy to design again based on CT attribute-value pairs. Plans are loaded at system startup, and any modifications to plans during system execution are reflected in the corresponding XML. Of course users are free to create and edit XML plans offline.

6.3 Programmatic and Human Interfaces

Rather than creating a traditional application programming interface (API), we provide a web interface to the DSMS through direct HTTP (and we are planning to expose the system as a *web service* through SOAP [WSD01]). Remote applications can be written in any language and on any platform. They can register queries, they can request and update CT attribute values, and they can receive the results of a query as a streaming HTTP response in XML. For human users, we have developed a web-based GUI exposing the same functionality.

7 Conclusion and Acknowledgments

A system realizing the techniques described in this paper is being developed at Stanford; please visit `http://www.db.stanford.edu/stream`. We are grateful to Aris Gionis, Jon McAlister, Liadan O'Callaghan, Qi Sun, and Jeff Ullman for their participation in the STREAM project.

References

[ABB+02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 221–232, Madison, Wisconsin, May 2002.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–16, Madison, Wisconsin, May 2002.

[Blo70] B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker,

N. Tatbul, and S. Zdonik. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, Hong Kong, China, August 2002.

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

[CF02] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, Hong Kong, China, August 2002.

[CGRS02] K. Chakrabarti, M.N. Garofalakis, R. Rastogi, and K. Shim. Approximate query processing using wavelets. In *Proc. 26th Intl. Conf. on Very Large Data Bases*, pages 111–122, Cairo, Egypt, August 2002.

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

[GG02] M.N. Garofalakis and P.B. Gibbons. Wavelet synopses with error guarantees. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 476–487, Madison, Wisconsin, May 2002.

[GGR02] M.N. Garofalakis, J. Gehrke, and R. Rastogi. Querying and mining data streams: You only get one look (*tutorial*). In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, page 635, Madison, Wisconsin, May 2002.

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

[MSHR02] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Dynamic multidimensional histograms. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, Madison, Wisconsin, May 2002.

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

[SLR96] P. Seshadri, M. Livny, and R. Ramakrishnan. The design and implementation of a sequence database system. In *Proc. 22nd Intl. Conf. on Very Large Data Bases*, pages 99–110, Bombay, India, September 1996.

[Soo91] M.D. Soo. Bibliography on temporal databases. *SIGMOD Record*, 20(1):14–24, March 1991.

[TGIK02] N. Thaper, S. Guha, P. Indyk, and N. Koudas. Dynamic multidimensional histograms. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 428–439, Madison, Wisconsin, May 2002.

[TMSF] P. Tucker, D. Maier, T. Sheard, and L. Fegaras. Punctuated data streams. http://www.cse.ogi.edu/~ptucker/PStream.

[UW02] J.D. Ullman and J. Widom. *A First Course in Database Systems (Second Edition)*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[VN02] S.D. Viglas and J.F. Naughton. Rate-based query optimization for streaming information sources. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 37–48, Madison, Wisconsin, May 2002.

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

[WSD01] Web Services Description Language (WSDL) 1.1, March 2001. http://www.w3.org/TR/wsdl.