Query Processing, Resource Management, and Approximation 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 thousands of continuous queries. In addition to the obvious need for multi-query optimization, judicious resource allocation, 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 autumn 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)
- An algorithm for exploiting constraints on data streams to reduce memory overhead during query processing (Section 4.1)
- A near-optimal scheduling algorithm for reducing inter-operator queue sizes (Section 4.2)
- A set of techniques for static and dynamic approximation to cope with limited resources (Section 5)
- An algorithm for allocating resources to queries (in a limited environment) that maximizes query result precision (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 mi-

This work was supported by NSF Grants IIS-0118173 and IIS-9817799, by Stanford Graduate Fellowships from Chambers, 3Com and Rambus, by a Microsoft graduate fellowship, and by grants from Microsoft, Veritas, and the Okawa Foundation.

Proceedings of the 2003 CIDR Conference

gration to distributed processing in mind. We may eventually extend our system to handle XML data streams, but extending to distributed processing has higher priority.

We have done no significant work so far in query plan generation. Our system supports a subset of our extended 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 [3], which provides extensive coverage of related work. We do make some comparisons to other work throughout this paper, particularly the *Aurora* project [5], 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 [3].

2 Query Language

The STREAM system allows direct input of query plans, similar to the Aurora approach [5] 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, so we call our language *CQL* (pronounced "sequel"), for *Continuous Query Language*. In this section we focus on the syntax and semantics of continuous queries in CQL.

Queries in a DSMS should handle data from both continuous data streams and conventional relations. For now assume a global, discrete, ordered time domain. Time-related issues are discussed briefly in Section 2.3.

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.) A stream can be thought of as a set (multiset to be precise) of pairs , indicating that a tuple arrives on the stream at time . In addition to the continuous source data streams that arrive at the DSMS, streams may result from queries or subqueries as specified in Section 2.2.

Relations are unordered, and they support updates and deletions as well as insertions (all of which are timestamped). In addition to relations stored by the DSMS, relations may result from queries or subqueries as specified in Section 2.2.

Syntactically, CQL extends 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*. CQL also contains two new operators—`Istream` and `Dstream`—whose function is discussed in Section 2.2.

As introduced in [3], in CQL 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 [19], 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 the keyword `Sample` parameterized by percentage sampling rate. For example, "`Sample(2)`" indicates that, independently, each data element in the stream should be retained with probability and discarded with probability .

2.1 Examples

Our example queries reference a stream `Requests` of requests to a web proxy server. Each request tuple has three attributes: `client_id`, `domain`, and `URL`.

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 (and the next two examples) are covered in Section 2.2.

The following query counts how many page requests were for pages served by Stanford's CS department web server, considering only each client's 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 extracts a 10% sample of requests to sites that are primarily of type "commerce," and from those it streams the URLs of requests where the client_id is in the range [1..1000]. Notice that

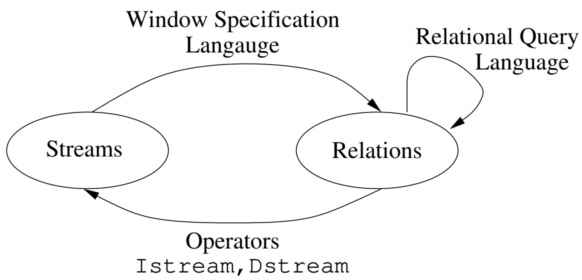

Figure 1: Mappings among streams and relations.

a subquery is used to produce an intermediate stream from which the 10% sample is taken.

```
Select T.URL
From
 (Select client_id, URL
  From Requests S, Domains R
  Where S.domain = R.domain
  And R.type = 'commerce') T Sample(10)
Where T.client_id Between 1 And 1000
```

2.2 Formal Semantics

One of our contributions is to provide a precise semantics for continuous queries over streams and relations. In specifying our semantics we had several goals in mind:

1. We should exploit well-understood relational semantics to the extent possible.
2. Since transformations are crucial for query optimization, we should not inhibit standard relational transformations and we should enable new transformations relevant to streams.
3. Easy queries should be easy to write, and simple queries should do what one expects.

Our approach is based on mappings between streams and relations, as illustrated in Figure 1. We explain each arc in Figure 1, then specify our query semantics. Recall again that we assume a global discrete time domain, further discussed in Section 2.3.

A stream is mapped to a relation by applying a window specification. In general any windowing language may be used, but CQL supports the `Rows`, `Range`, `Partition By`, and `Where` constructs described earlier. A window specification is applied to a stream up to a specific time , and the result is a finite set of tuples which is treated as a relation. A window specification may be applied to a source stream, or to a stream produced by a subquery. The semantics of the `Sample` operator, which is applied before windowing, are straightforward and not discussed again in this section.

Relations are mapped to streams by applying special operators:

> `Istream` (for "insert stream") applied to relation contains a stream element whenever tuple is in at time but not in at time .
>
> Analogously, `Dstream` (for "delete stream") applied to relation contains a stream element whenever tuple is in at time but not in at time .

Although these operators can be specified explicitly in queries in order to turn relations into streams for windowing, or to produce streamed rather than relational query results, the most common case is an implicit `Istream` operator as part of CQL's default behavior, discussed below.

The last mapping in Figure 1 is from relations to relations via a relational query language. In general any relational query language may be used, but CQL relies on SQL as its relational basis.

Using the mappings in Figure 1, CQL queries can freely mix relations and streams. However, whenever a stream is used it is mapped immediately to a relation by an explicit or implicit window specification. We can now state the semantics of continuous queries over streams and relations:

> The result of a query at time is obtained by taking all relations at time , all streams up to time converted to relations by their window specifications, and applying conventional relational semantics. If the outermost operator is `Istream` or `Dstream` then the query result is converted to a stream, otherwise it remains as a relation.

Let us briefly consider the three example queries in Section 2.1. The first two queries are similar and relatively straightforward: At any time , a window is evaluated on the `Requests` stream up to time , a predicate is applied, and the result contains the number of remaining tuples. By default, the result is a singleton relation, however if we add an outermost `Istream` operator the result instead streams a new value each time the count changes. The third query is somewhat more complex, relying on two CQL defaults: First, when no window specification is provided the default window is "`[Range Unbounded Preceding]`." Second, whenever the outermost `From` list of a non-aggregation query contains one or more streams, the query result has an `Istream` operator applied by default. We leave it to the reader to verify that the transformations between streams and relations in this query do indeed produce the desired result.

Space limitations prohibit detailed elaboration of our semantics, but we briefly discuss the three goals set out at the beginning of this subsection. First, clearly we rely heavily on existing relational semantics. Second, relational transformations continue to hold and our language supports a number of useful stream-based transformations. For instance, in our third example query, if relation `Domains` is static then we can transform the (default) `Unbounded` window on the `Requests` stream into a

Now window. Finally, although the mapping-based semantics may appear complex, it is our belief that the defaults in our language do make easy queries easy to write, and queries do behave as they appear.

As the most basic of test cases for our semantic goals, query "`Select * From S`" should produce stream , and it does. The default window for is `Unbounded` (but can be transformed to `Now` if contains a key). Each time a stream element arrives on , is logically added to the relational result of the query, and thus is generated in the default `Istream` query result with timestamp .

2.3 Stream Ordering and Timestamps

Our language semantics—or more accurately the ability to implement our semantics—makes a number of implicit assumptions about time:

- So that we can evaluate row-based (`Rows`) and time-based (`Range`) sliding windows, all stream elements arrive in order, timestamped according to a global clock.
- So that we can coordinate streams with relation states, all relation updates are timestamped according to the same global clock as streams.
- So that we can generate query results, the global clock provides periodic "heartbeats" that tell us when no further stream elements or relation updates will occur with a timestamp lower than the heartbeat value.

If we use a centralized system clock and the system timestamps stream elements and relation updates as they arrive, then these assumptions are satisfied. We can also handle less strict notions of time, including application-defined time. Full details are beyond the scope of this paper, but out-of-order streams can be ordered by buffering and waiting for heartbeats, and the absence of a heartbeat can be compensated for with timeout mechanisms, in the worst case resulting in some query result imprecision.

Note that the related areas of *temporal* and *sequence* query languages [15, 16] 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.

2.4 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.

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 [5]. 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 an initial implementation in place for a fairly substantial subset of the language presented in Section 2, omitting primarily certain subqueries and many esoteric features of standard SQL. In this section we highlight the features of our basic query processing architecture but do not go into detail about individual query operators. Most of our operators are stream- or window-based analogs to operators found in 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* [21]). 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 [1]. Thus, an important feature to support is synopses that use some kind of summarization technique to limit their size [10], 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 Section 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 4.2).

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-

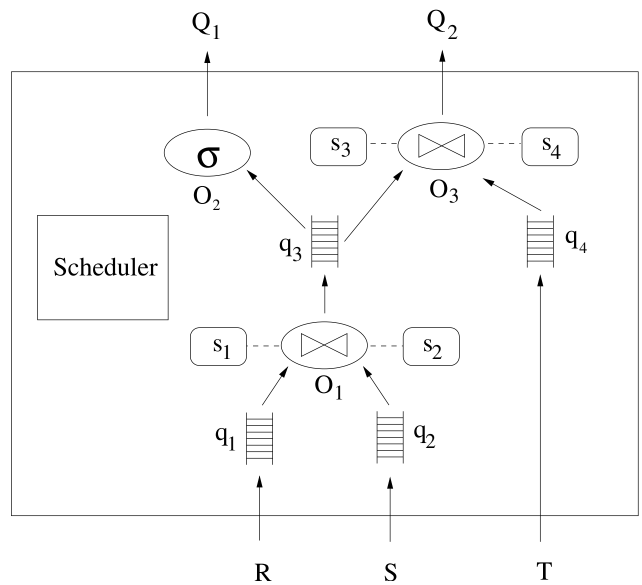

Figure 2: Plans for queries , over streams , , .

window synopses, which keep a summary of the last ( ) tuples of some intermediate stream. One use of sliding-window synopses is to implement the window specifications in our query language (Section 2). For `RANGE` window specifications we cannot bound the synopsis size, but for `ROWS` window specifications the memory requirement is determined by the tuple size and number of rows . Sliding-window synopses also are used for approximation (Section 5), in which case typically is determined by the tuple size and memory allocation .

3.1 Example

Figure 3.1 illustrates plans for two queries, and . Together the plans contain three operators – , four synopses – (two per join operator), and four queues – . Query is a selection over a join of two streams and . Query is a join of three streams, , , and . The two plans share a subplan joining streams and by sharing its output queue . Plan and queue sharing is discussed in Section 3.2. Execution of query operators is controlled by a global *scheduler*. When an operator is scheduled, control passes to for a period currently determined by number of tuples processed, although we may later incorporate timeslice-based scheduling. Section 4.2 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 [14]. 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., 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 "share" 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 output from a join operator , and suppose is very unselective so it produces nearly the cross-product of its inputs. If 's parent in one query is a "heavy consumer," then our scheduling algorithm (Section 4.2) is likely to schedule frequently in order to produce tuples for to consume. If 's parent in another query is a "light consumer," then the scheduler will schedule less frequently so tuples don't proliferate in . In this situation it may not be beneficial for and to share a common subplan rooted in .

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 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 are focusing initially 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.

In this section we discuss two techniques we have developed that can reduce memory overhead dramatically during query execution. Neither of these techniques compromises the precision of query results.

1. An algorithm for incorporating known constraints on input data streams to reduce synopsis sizes. This work is described in Section 4.1.

2. An algorithm for operator scheduling that minimizes queue sizes. This work is described in Section 4.2.

In Section 5 we discuss approximation techniques, and the important interaction between resource allocation and approximation.

4.1 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,

---

Disk also could be used for synopses and queues, although in that case we might want to treat I/O as a separate resource given its different performance characteristics, as in Aurora [5].

either by gathering statistics over time or through constraint specifications at stream-registration time, we can use this information to reduce resource requirements without sacrificing 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 [18].)

We have identified several types of constraints over data streams, and for each constraint type we specify an "adherence parameter" that captures how closely a given stream or pair 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 standalone 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) with a stream `Fulfillments` (hereafter) 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 [1]. However, if we know that all tuples for a given orderID and itemID arrive on before the corresponding tuples arrive on , then we need not maintain a join synopsis for the operand at all. Furthermore, if tuples arrive clustered by orderID, then we need only save 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 to be clustered by orderID within a tolerance parameter : no more than 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 before the corresponding tuple arrives on , but we may be able to bound the time delay with a constant . These constants are the "adherence parameters" discussed earlier, and it should be clear that the smaller the value of , the smaller the necessary synopses.

The constraints considered in our work are *many-one join* and *referential integrity* constraints between two streams, and *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. Details are beyond the scope of this paper.

4.2 Scheduling

Query plans are executed via a *global scheduler*, which calls the `run` methods of query plan operators (Section 3) in order to make progress in 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 focused on minimizing peak total queue size during query processing, in keeping with our general project goal of coping with limited resources.

Consider the following very simple example. Suppose we have a query plan with two unary operators: operates on input queue , writing its results to queue which is the input to operator . Suppose takes one time unit to operate on a batch of tuples from , and it has 20% selectivity, i.e., it introduces tuples into when consuming tuples from . (Time units and batches of input tuples simplify exposition; their actual values are not relevant to the overall reasoning in our example.) Operator takes one time unit to operate on tuples, and let us assume that its output is not queued by the system since it is the final result of the query. Suppose that over the long-term the average arrival rate of tuples at is no more than tuples per time unit, 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 5.2.2.) However, tuple arrivals may be bursty.

Here are two possible scheduling strategies:

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

Suppose we have the following arrival pattern: tuples arrive at every time instant from to , then no tuples arrive from time through . On average, tuples arrive per unit of time, but with an initial burst. The following table shows the total size of queues and under the two scheduling strategies during the burst, where each table entry is a multiplier for .

Time	1	2	3	4	5	6	7
Strat. 1	1	1.2	2	2.2	3	3.2	4
Strat. 2	1	1.2	1.4	1.6	1.8	2.0	2.2

After time , queue sizes for both strategies decline until they reach 0 when time . In this example, both strategies finish at , and Strategy 2 is clearly preferable in terms of run-time memory overhead during the burst.

We have designed a scheduling policy that provably has near-optimal maximum total queue size and is based roughly 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). However, this *per-operator greedy* approach may underutilize a high-priority operator if the operators feeding it are low priority. Therefore, we consider *chains* of operators within a plan when making scheduling decisions. Details of our scheduling algorithm and the proof of its near-optimality are fairly involved and not presented due to space limitations.

Scheduling chains of operators also is being considered in Aurora's *train scheduling* algorithm [5], although for entirely different reasons. 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 peak queue sizes as our only scheduling objective. Also, we have been assuming a single thread of control shared among operators, while Aurora considers multiple threads for different operators. *Eddies* [2, 13] uses a scheduling criterion similar to our per-operator greedy approach, but for routing individual tuples through operator queues rather than scheduling the execution of operators. Furthermore, like Aurora the goal of Eddies is to maximize throughput, unlike our goal of minimizing total queue size.

While our algorithm achieves queue-size minimization, it may incur increased time to initial results. In our example above, although both strategies finish processing tuples at the same time, Strategy 1 generally has the potential to produce initial results earlier than Strategy 2. An important next step is to incorporate response time and (weighted) fairness across queries into our scheduling algorithm.

5 Approximations

The previous section described two complementary techniques for reducing the memory overhead (synopses and queue sizes respectively) required during query processing. Even exploiting those techniques, 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. In this section we present a number of approximation techniques, and we discuss the close relationship between resource management and approximation.

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 2.4) involves a huge number of variables, and certainly is intractable in the general case.

In the remainder of this section we propose some *static approximation* (compile-time) techniques in Section 5.1 and some *dynamic approximation* (run-time, adaptive) techniques in Section 5.2. In Section 5.3 we present our first algorithm—largely theoretical at this point but a good first step—for allocating memory in order to maximize result precision.

In comparison with other systems for processing queries over data streams, both the *Telegraph* [12] and *Niagara* [9] 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 [5] 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 5.2.2.

5.1 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 5.2) are:

1. Assuming the statically optimized query is executed precisely by the system, the user is guaranteed cer-

tain 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*.

5.1.1 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. 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., [5, 7, 12, 13, 20].

Suppose is an operator that incorporates a window specification, most commonly a windowed join. Reducing 's window size not only affects the resources used by , 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 is a duplicate-elimination operator, then shrinking 's window can actually increase its output rate.
- If is part of the right-hand subtree of a negation construct (e.g., `NOT EXISTS` or `EXCEPT`), then reducing the size of '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.

5.1.2 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 will not reduce the resource requirements of , 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 biased sampling when we do so, especially in the presence of joins as discussed in [8].

5.2 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 5.1.1, *sampling*, which is analogous to *sampling rate reduction* in Section 5.1.2, and *load shedding*.

5.2.1 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 5.1.1). 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* [17] or compressed *wavelets* [11] when the synopsis is used for aggregation or even for a join [6], and using *Bloom filters* [4] 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., [11], however we have not solved the general problem of conveying accuracy to users dynamically.

5.2.2 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 5.1.1), although again we note that when plan sharing is in place one sampling rate may affect multiple queries.

We can also simply drop tuples from queues when the queues grow too large, a technique sometimes referred to as *load shedding* [5]. Load shedding at queues differs from sampling at operators since load shedding may drop chunks of tuples at a time, instead of eliminating tuples probabilistically. Both are effective techniques for reducing queue sizes. While sampling may be more "unbiased," dropping chunks of tuples may be easier to implement and to make decisions about dynamically.

5.3 Resource Allocation to Maximize Precision

In this subsection we present our preliminary work on allocating resources to query plans with the objective of maximizing result precision. The work is applicable whenever resource limitations are expected to force approximate query results. We address a restricted scenario for now, but we believe our approach 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 earlier in this section. (We have encoded a number of realistic functions from memory to precision for several relational operators, but details are beyond the scope of this paper.) 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 for which we use a packaged solver, although we are optimistic about finding a more efficient formulation.

In the language handled by our 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* and *FN* . *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 resources 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.

5.4 Resource Management and Approximation: 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 5.1 and 5.3), 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 5.2). 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 can be controlled by updating the value of attribute `Memory` in '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 is stored in attribute `Count` of '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 [22]). 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`, which also contains links to papers expanding on several of the topics covered in this paper, including the query language, exploiting constraints, operator scheduling, and resource allocation.

We are grateful to Aris Gionis, Jon McAlister, Liadan O'Callaghan, Qi Sun, and Jeff Ullman for their participation in the STREAM project, and to Bruce Lindsay for his excellent suggestion about heartbeats.

References

[1] 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.

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

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

[13] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, Madison, Wisconsin, May 2002.

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

[15] 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.

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

[17] 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.

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

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

[20] 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.

[21] 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.

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