Chain: Operator Scheduling for Memory Minimization in Data Stream Systems

Brian Babcock [*] Shivnath Babu [†] Mayur Datar [‡] Rajeev Motwani [§]

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

ABSTRACT

In many applications involving continuous data streams, data arrival is bursty and data rate fluctuates over time. Systems that seek to give rapid or real-time query responses in such an environment must be prepared to deal gracefully with bursts in data arrival without compromising system performance. We discuss one strategy for processing bursty streams — *adaptive, load-aware scheduling* of query operators to minimize resource consumption during times of peak load. We show that the choice of an operator scheduling strategy can have significant impact on the run-time system memory usage. We then present *Chain scheduling*, an operator scheduling strategy for data stream systems that is near-optimal in minimizing run-time memory usage for any collection of single-stream queries involving selections, projections, and foreign-key joins with stored relations. Chain scheduling also performs well for queries with sliding-window joins over multiple streams, and multiple queries of the above types. A thorough experimental evaluation is provided where we demonstrate the potential benefits of Chain scheduling, compare it with competing scheduling strategies, and validate our analytical conclusions.

1. INTRODUCTION

In a growing number of information processing applications, data takes the form of *continuous data streams* rather than traditional stored databases. These applications share two distinguishing characteristics that limit the applicability of standard relational database technology: (1) the volume of data is extremely high, and (2) on the basis of the data, decisions are arrived at and acted upon in close to real time. The combination of these two factors make traditional solutions, where effectively data is loaded into static databases for offline querying, impractical for many applications. Motivating applications include networking (traffic engineering, network monitoring, intrusion detection), telecommunications (fraud detection, data mining), financial services (arbitrage, financial monitoring), e-commerce (clickstream analysis, personalization), and sensor networks.

These applications have spawned a considerable and growing body of research into data stream processing [3], ranging from algorithms for data streams to full-fledged data stream systems such as Aurora [5], Hancock [7], Niagara [21], STREAM [26], Tangram [22, 23], Tapestry [28], Telegraph [10], Tribeca [27], and others. For the most part, research in data stream systems has hitherto focused on devising novel system architectures, defining query languages, designing space-efficient algorithms, and so on. Important components of systems research that have received less attention to date are run-time resource allocation and optimization. In this paper we focus on one aspect of run-time resource allocation, namely *operator scheduling*.

There are some features unique to data stream systems, as opposed to traditional relational DBMSs, which make the run-time resource allocation problem different and arguably more critical: First, data stream systems are typically characterized by the presence of multiple *long-running continuous* queries. Second, most data streams are quite irregular in their rate of arrival, exhibiting considerable burstiness and variation of data arrival rates over time. This phenomenon has been extensively studied in the networking context [9, 16, 34]. Data network traffic is widely considered to be *self-similar* and to exhibit *long-range dependence*. Similar findings have been reported for web-page access patterns and e-mail messages [15]. Consequently, conditions in which data stream queries are *executed* are frequently quite different from the conditions for which the query plans were *generated*. Therefore, *adaptivity* becomes critical to a data stream system as compared to a traditional DBMS.

Various approaches to adaptive query processing are possible given that the data may exhibit different types of variability. For example, a system could modify the structure of query plans, or dynamically reallocate memory among query operators in response to changing conditions, as suggested in [19], or take a holistic approach to adaptivity and do away with fixed query plans altogether, as in the Eddies architecture [2, 18]. While these approaches focus primarily on adapting to changing characteristics of the *data itself* (e.g., changing selectivities), we focus on adaptivity towards

[*]Supported in part by a Rambus Corporation Stanford Graduate Fellowship and NSF Grant IIS-0118173.

[†]Supported in part by NSF Grant IIS-0118173.

[‡]Supported in part by Siebel Scholarship and NSF Grant IIS-0118173.

[§]Supported in part by NSF Grant IIS-0118173, an Okawa Foundation Research Grant, an SNRC grant, and grants from Microsoft and Veritas.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD 2003, June 9-12, 2003, San Diego, CA.
Copyright 2003 ACM 1-58113-634-X/03/06 ...$5.00.

changing *arrival* characteristics of the data, similar to [1, 29, 31]. As mentioned earlier, most data streams exhibit considerable burstiness and arrival-rate variation. It is crucial for any stream system to adapt gracefully to such variations in data arrival, making sure that we do not run out of critical resources such as main memory during the bursts. The focus of this paper is to design techniques for such adaptivity.

When processing high-volume, bursty data streams, the natural way to cope with temporary bursts of unusually high rates of data arrival is to buffer the backlog of unprocessed tuples and work through them during periods of light load. However, it is important for the stream system to minimize the memory required for backlog buffering. Otherwise, total memory usage can exceed the available physical memory during periods of heavy load, causing the system to page to disk, which often causes sudden and dramatic decrease in overall system throughput. As we will show in Section 2, the operator scheduling strategy used by the data stream system has a significant impact on the total amount of memory required for backlog buffering. The question we address in this paper is how to most efficiently schedule the execution of query operators to keep the total memory required for backlog buffering at a minimum, assuming query plans and operator memory allocation are fixed.

1.1 Road Map

The rest of this paper is organized as follows. We begin in Section 2 by briefly describing our model for the processing of stream queries and illustrating through an example that the run-time memory requirement can be greatly affected by the operator scheduling strategy, thereby motivating the need for an intelligent scheduling strategy. In Section 3 we formalize the problem of operator scheduling that we consider. Our analytical results on the performance of proposed scheduling strategies are provided in Section 4. In particular, we present *Chain scheduling*, a near-optimal scheduling strategy for the case when all queries involve a single stream, possibly joining with stored relations via foreign-key joins. When joins with stored relations are not foreign-key joins, we present a weaker guarantee. In Section 4.2 we introduce various other scheduling strategies and discuss (qualitatively) how they compare with Chain scheduling. In Section 4.3 we extend Chain scheduling to queries involving windowed-joins over multiple streams. In Section 5 we describe our experimental comparison of the performance of different scheduling strategies. We describe related work in Section 6 before ending with our conclusions in Section 7.

2. STREAM QUERY PROCESSING: MODEL AND ASSUMPTIONS

Data stream systems are characterized by the presence of multiple continuous queries [5, 6, 18, 19]. Query execution can be conceptualized as a *data flow diagram* (as in [5]), which is a directed acyclic graph (DAG) of nodes and edges, where the nodes are pipelined operators (aka *boxes*) that process tuples and edges (aka *arrows*) represent composition of operators. An edge from node A to node B indicates that the output of node A is an input to node B. The edge (A, B) also represents an *input queue* that buffers the output of operator A before it is input to the operator B. Input data streams are represented as "leaf" nodes that have no input edges and query outputs are represented as "root" nodes that have no output edges.

If a single data stream is input to multiple queries, we assume that multiple copies of the stream are created by the system and fed to each of the queries separately. Consequently, we assume that all streams participate in only one query and therefore every incoming tuple is input to only a single query. If instead of making multiple copies, the system chooses to share the input buffer across multiple queries, the optimal strategy may differ; we plan to consider this case in future work. In this paper, we will concentrate on the case of a *fixed query plan* which does not change over time.

As mentioned earlier, we assume that all operators execute in a *streaming* or *pipelined* manner. Operators like select and project naturally fit in this category. A join operator can also be made to work in this manner, using a *symmetric hash join* implementation [35]. While operators like select and project do not need to maintain any run-time state, a symmetric-hash-join operator needs to maintain state that is proportional to the size of the input seen so far, which is unbounded for streams. However, for applications on streams, a relevant notion of join is that of a *sliding-window* join [5, 6, 14, 18, 19], where a tuple from one stream is joined on arrival with only a bounded *window* of tuples in the other stream, and vice versa. Consequently, the state maintained by a sliding-window join operator is bounded. An example of a sliding-window join is a *tuple-based* sliding-window join where the window on each stream is specified as a fixed number of tuples [14, 19]. Clearly, the run-time state stored by a tuple-based sliding-window join operator is of fixed size.

In summary, the operators that we consider act like *filters* which operate on a tuple and produce s tuples, where s is the *selectivity* of the operator. The selectivity s is at most 1 for the select and project operators, but it may be greater than 1 for a join. For the rest of this paper, the reader should keep in mind that when we refer to the selectivity s of an operator, we are referring to the above notion of viewing the operator as a *filter* that (on an average) produces s tuples on processing 1 tuple. We assume that the run-time state stored by each operator is fixed in size and thus the variable portion of the memory requirement is derived from the sizes of the input queues to operators. The memory for all input queues is obtained from a common system memory pool. The aim of operator scheduling is to minimize the total memory requirement of all queues; in particular, to keep the memory requirement below a certain threshold.

An important strategy for adapting to higher rates of data arrival is *load shedding,* i.e., dropping input tuples based on some criterion such as *quality-of-service* [5]. Load shedding may become necessary when the input rate consistently exceeds the maximum processing rate of the system so that backlogs in the input queues will build up and eventually we will run out of memory. There are various strategies for load shedding (see [5]), but that is not the focus of this paper. These strategies are orthogonal to those that we discuss in this paper and could be used in conjunction. Instead, we focus on the following issue: Even though the average arrival rate is within computational limits, there may be bursts of high load leading to high memory usage to buffer the backlog of unprocessed tuples. How do we efficiently schedule operators in order to keep the peak memory usage at a minimum? Our assumption that the arrival rate is within computational limits implies that it is eventually possible to clear the backlog of unprocessed tuples, when the bursts of high arrival rate have receded.

Every tuple that enters the system must pass through a unique path of operators, referred to as an *operator path*. (Recall that we do not share tuples among query plans.) If the arrival rate of the tuples is uniform, then there is not much to be done. The following simple scheduling strategy will have the minimum memory requirement: Whenever a tuple enters the system, schedule it through all the operators in its operator path. If any operator in the path is a join that produces multiple tuples, then schedule each resulting tuple in turn through the remaining portion of the operator path. Henceforth, this strategy will be referred to as the FIFO (*First In,*

First Out) strategy. But note that, as mentioned earlier, such uniformity in arrival is seldom the case and hence we need more sophisticated scheduling strategies guaranteeing that the queue sizes do not exceed the memory threshold. The following example illustrates how a scheduling strategy can fare better than FIFO and make the difference between exceeding memory threshold or not.

EXAMPLE 2.1. *Consider a simple operator path that consists of two operators: O_1 followed by O_2. Assume that O_1 takes unit time to process a tuple and produces 0.2 tuples[1], i.e., its selectivity is 0.2. Further, assume that O_2 takes unit time to operate on 0.2 tuples (alternatively, 5 time units to operate on 1 tuple) and produces 0 tuples, i.e., O_2 outputs the tuple out of the system and hence has selectivity 0. Thus, it takes 2 units of time for any tuple to pass through the operator path.*

We assume that, over time, the average arrival rate of tuples is no more than 1 tuple per 2 units of time. This assumption guarantees that we do not have an unbounded build-up of tuples over time. However, the arrival of tuples could be bursty. Consider the following arrival pattern: A tuple arrives at every time instant from $t = 0$ to $t = 6$, then no tuples arrive from time $t = 7$ through $t = 13$. Consider the following two scheduling strategies:

- **FIFO scheduling:** *Tuples are processed in the order that they arrive. A tuple is passed through both operators in two consecutive units of time, during which time no other tuple is processed.*
- **Greedy scheduling:** *At any time instant, if there is a tuple that is buffered before O_1, then it is operated on using 1 time unit; otherwise, if tuples are buffered before O_2, then 0.2 tuples are processed using 1 time unit.*

The following table shows the total size of input queues for the two strategies:

Time	*Greedy scheduling*	*FIFO scheduling*
0	*1*	*1*
1	*1.2*	*1.2*
2	*1.4*	*2.0*
3	*1.6*	*2.2*
4	*1.8*	*3.0*
5	*2.0*	*3.2*
6	*2.2*	*4.0*

After time $t = 6$, input queue sizes for both strategies decline until they reach 0 after time $t = 13$. Observe that Greedy scheduling has smaller maximum memory requirement than FIFO scheduling. In fact, if the memory threshold is set to 3, then FIFO scheduling becomes infeasible, while Greedy scheduling does not. □

This example illustrates the need for an intelligent scheduling strategy in order to execute queries using limited amount of memory. The aim of this paper is to design scheduling strategies that will form the core of a *resource manager* in a data stream system.

Here are some desirable properties of any such scheduling strategy:

- The strategy should have provable guarantees on its performance in terms of metrics such as resource utilization, response times, and latency.
- Because it will be executed every few time steps, the strategy should be efficient to execute.
- The strategy should not be too sensitive to inaccuracies in estimates of parameters such as queue sizes, operator selectivities, and operator execution times.

The algorithms and heuristics developed in this paper do not achieve all the stated goals, although we feel that they represent an important first step. Our algorithms have some shortcomings; for example, they do not incorporate application-level quality-of-service requirements, so some important tuples that are waiting to be processed may be delayed for an unacceptably long time before being scheduled. Our future work will be directed towards removing such shortcomings.

[1]A tuple does not refer to an individual tuple, but rather to a fixed unit of memory, such as a page, that contains multiple tuples and for which we can assume that selectivity assumptions hold on the average. More details are provided in Section 3.

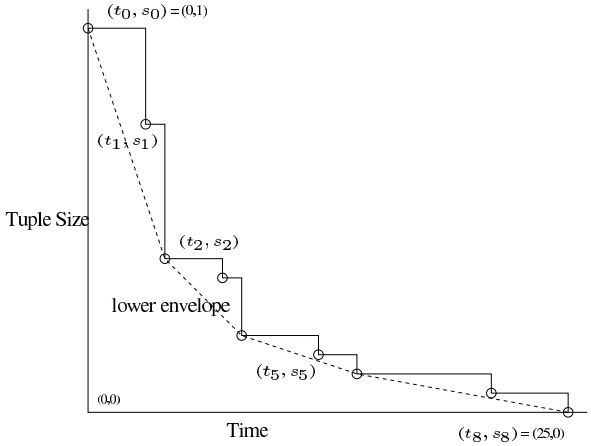
3. OPERATOR SCHEDULING AND MEMORY REQUIREMENTS

As mentioned earlier, query execution can be captured by a data flow diagram, where every tuple passes through a unique *operator path*. Thus queries can be represented as rooted trees. Every operator is a filter that operates on a tuple and produces s tuples, where s is the operator selectivity. Obviously, the selectivity assumption does not hold at the granularity of a single tuple but is merely a convenient abstraction to capture the average behavior of the operator. For example, we assume that a select operator with selectivity 0.5 will select about 500 tuples of every 1000 tuples that it processes. Henceforth, a tuple should not be thought of as an individual tuple, but should be viewed as an convenient abstraction of a memory unit, such as a page, that contains multiple tuples. Over adequately large memory units, we can assume that if an operator with selectivity s operates on inputs that require one unit of memory, its output will require s units of memory.

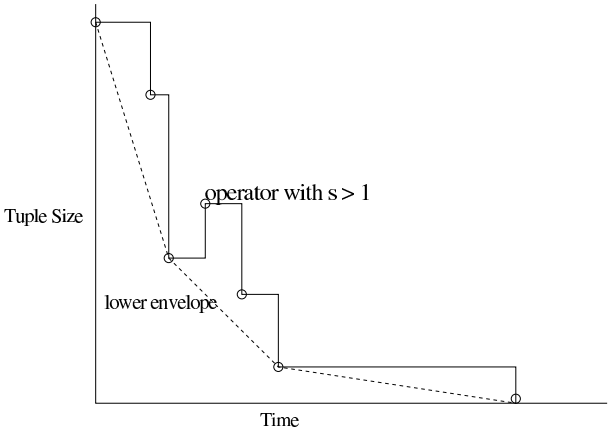
The FIFO scheduling strategy (from Example 2.1) maintains the entire backlog of unprocessed tuples at the beginning of each operator path. Thus the sizes of intermediate input queues will be small in case of FIFO, albeit at the cost of large queues at the beginning of each operator path. In fact, the Greedy strategy performed better than FIFO in Example 2.1 precisely because it chose to maintain most of its backlog at the input queue between the first operator and the second. Since the first operator had a low selectivity, it was beneficial to buffer two fractional tuples (each of size 0.2) at this intermediate queue, rather than buffer a single tuple of size 1 at the beginning of the operator path. This suggests that it is important to consider the different sizes of a tuple as it progresses through its operator path. We capture this using the notion of a *progress chart*, illustrated in Figure 1.

The horizontal axis of the progress chart represents time and the vertical axis represents tuple size. The chart contains $m + 1$ *operator points* $(t_0, s_0), (t_1, s_1), \ldots, (t_m, s_m)$, marked by circles, where $0 = t_0 < t_1 < t_2 < \ldots < t_m$ are positive integers. The points represent an operator path consisting of m operators, where the ith $(1 \leq i \leq m)$ operator takes $t_i - t_{i-1}$ units of time to process a tuple of size s_{i-1}, at the end of which it produces a tuple of size s_i. The selectivity of operator i is s_i/s_{i-1}. (We assume that operators can be preempted in the middle of their execution and resumed later; a time unit represents the smallest duration an operator must be continuously run before preemption.)

Adjacent operator points are connected by a solid line called the *progress line* representing the progress of a tuple along the operator path. We imagine tuples as moving along this progress line. A tuple τ enters the system with size $s_0 = 1$. After being processed by the ith query operator, the tuple has received t_i total processor



(a) All operators have selectivity $s \leq 1$



(b) One operator has selectivity $s > 1$

Figure 1: Progress charts

time and its size has been reduced to s_i. At this point, we say that the tuple has made *progress* $p(\tau) = t_i$. A good way to interpret the progress chart is the following — the ith horizontal segment represents the execution of the ith operator, and the following vertical segment represents the drop in tuple size due to operator i's selectivity. For every operator path, the last operator has selectivity $s_m = 0$. This is because it will eject the tuples it produces out of the system and they no longer need to be buffered. If all operator selectivities are less than or equal to 1, the progress chart is non-increasing as shown in Figure 1(a). However, for query plans that include a join operator with selectivity greater than 1, the progress chart looks like the one shown in Figure 1(b).

We assume that the selectivities and per-tuple processing times are known for each operator. We use these to construct the progress chart as explained above. Selectivities and processing times could be learned during query execution by gathering statistics over a period of time. If we expect these values to change over time, we could use the following strategy as in [2]: divide time into fixed windows and collect statistics independently in each window; use the statistics from the ith window to compute the progress chart for the $(i+1)$st window.

Consider an operator path with m operators represented by the operator points $(t_0, s_0), \ldots, (t_m, s_m)$. For any point (t, s) along the progress line, where $t_{i-1} \leq t < t_i$ for some $1 \leq i \leq m$, the *derivative* of the point with respect to the jth operator point (t_j, s_j) is given by $d(t, s, j) = \frac{-(s_j - s)}{t_j - t}$, for $m \geq j \geq i$. The derivative is undefined for $j < i$. The derivative is nothing but the negative slope of the line connecting the point (t, s) to an operator point to its right. The *steepest* derivative at the point (t, s) for $t_{i-1} \leq t < t_i$ is denoted $D(t, s) = \max_{m \geq j \geq i} d(t, s, j)$, and the operator point for which the maximum is achieved[2] is defined as the *Steepest Descent Operator Point*, $SDOP(t, s) = (t_b, s_b)$ where $b = \min\{j | m \geq j \geq i$ and $d(t, s, j) = D(t, s)\}$.

Consider the following subsequence of operator points: Start with the point $x_0 = (t_0, s_0)$. Let $x_1 = SDOP(x_0)$, let $x_2 = SDOP(x_1)$, and so on, until we finally reach the point $x_k = (t_m, s_m)$. If we connect this sequence of points $x_0, x_1, \ldots, x_k$ by straight line segments, we obtain the *lower envelope* for the progress chart. In Figures 1(a) and 1(b), the lower envelope is represented by a dashed line. Observe that the lower envelope is convex.

We make the following simple observation regarding the lower envelope.

PROPOSITION 3.1. *Let $(t_0, s_0), (t_1, s_1), \ldots, (t_k, s_k)$ denote the sequence of points on the lower envelope for a progress chart. The magnitude of the slopes for the segments joining (t_{i-1}, s_{i-1}) and (t_i, s_i), for $1 \leq i \leq k$, must be non-increasing with i.*

We give only a brief sketch of the proof. Suppose there exists an index i such that the magnitude of slope for the segment joining (t_i, s_i) to (t_{i+1}, s_{i+1}) is strictly greater than that of the segment joining (t_{i-1}, s_{i-1}) to (t_i, s_i). Then, the slope of the segment joining (t_{i-1}, s_{i-1}) to (t_{i+1}, s_{i+1}) is strictly greater than that for the segment joining (t_{i-1}, s_{i-1}) to (t_i, s_i). In that case, by definition, (t_i, s_i) is not the SDOP for the point (t_{i-1}, s_{i-1}), and hence does not belong to the lower envelope.

4. SCHEDULING STRATEGIES AND PERFORMANCE BOUNDS

We are now ready to present our proposed scheduling strategy and derive its performance bounds. We begin by defining a framework for specifying a strategy. Ideally, one can view a scheduling strategy as being invoked at the end of every unit of time; the smallest duration for which operators should be run without preemption. On each invocation, a strategy must select an operator from among those with nonempty input queues and schedule it for the next time unit. In reality, we need not invoke the strategy after every time unit. It turns out that in most cases it is only required to do so periodically or when certain events occur, e.g., an operator that is currently scheduled to run finishes processing all tuples in its input queue, or a new block of tuples arrives on an input stream. All scheduling strategies considered in this paper choose the next operator to schedule based on statically-assigned priorities, i.e., scheduling and operator execution do not change operator priorities. Thus, the scheduling strategy itself causes little overhead since priorities need not be recomputed whenever operators are scheduled or stopped. Under this model, we will describe strate-

[2]To break ties, when there are multiple such points, we use the one with the smallest index.

gies that assign priorities to different operators across *all* queries and provide guarantees where possible.

As we will see momentarily, the priority that our scheduling strategy assigns to each operator is completely determined by the progress chart that the operator belongs to. We need to ensure that our estimates for selectivities and per-tuple processing times, based on which the progress charts are computed, are not very outdated. Therefore, we periodically recompute these progress charts, based on the statistics that are gathered over the most recent window of time during query execution. The task of recomputing the progress charts from these statistics is straightforward and incurs little overhead.

The queries that we consider can be categorized into the following two types.

1. **Single-Stream Queries:** These queries typically involve selections and projections over a single stream, may involve joins with one or more static stored relations, and possibly end with a grouping and aggregation. This is a fairly common class of queries in data stream applications. Single-stream queries are discussed in Section 4.1.

2. **Multi-Stream Queries:** A distinguishing feature of this class of queries is that they involve at least one join between two streams. Such queries are typically used to correlate data across two or more streams (e.g., a query that joins network packet streams from two routers to find packets that passed through both routers). As indicated earlier, we assume that all joins over streams are tuple-based sliding-window joins. Multi-stream queries are discussed in Section 4.3.

4.1 Single-Stream Queries

We present an operator scheduling algorithm that we call *Chain scheduling*. The name "chain scheduling" comes from the fact that our algorithm groups query operators into operator chains corresponding to segments in the lower envelope of the query progress chart.

Given an operator path and its progress chart P, let P' denote the *lower envelope simulation* of P, defined as the progress chart whose progress line consists of the lower envelope of P. Consider a tuple τ and a progress line segment l_i joining (t_i, s_i) to (t_{i+1}, s_{i+1}) in P'. We say that tuple τ *lies on* l_i if $t_i \leq p(\tau) < t_{i+1}$. (Recall from Section 3 that $p(\tau)$ denotes the progress made by τ along the operator path.) Moreover, we say that τ is *at the beginning of* l_i if $p(\tau) = t_i$ and that it is *in the middle of* l_i if $t_i < p(\tau) < t_{i+1}$.

Consider a data stream system with n distinct operator paths represented by the progress charts $\mathcal{P} = \{P_1, \ldots, P_n\}$ with lower envelope simulations $\mathcal{P}' = \{P'_1, \ldots, P'_n\}$. The Chain scheduling strategy (henceforth simply Chain, for brevity) for such a system proceeds as follow:

> **Chain:** At any time instant, consider all tuples that are currently in the system. Of these, schedule for a single time unit the tuple that lies on the segment with the steepest slope in its lower envelope simulation. If there are multiple such tuples, select the tuple which has the earliest arrival time.

The way we describe our strategy, it may appear that it is "tuple-based", i.e., we make decisions at the level of each tuple. That is not the case — Chain statically assigns priorities to operators (not tuples) equaling the slope of the lower-envelope segment to which the operator belongs. At any time instant, of all the operators with tuples in their input queues, the one with the highest priority is chosen to be executed for the next time unit.

Using the special structure of the lower envelope, we show that Chain is an optimal strategy for the collection of progress charts $\mathcal{P}'$. To this end, define a *clairvoyant* strategy as one which has full knowledge of the progress charts and *the tuple arrivals in the future*. Clearly, no strategy can actually have knowledge of the future, but the notion of clairvoyance provides a useful benchmark against which we can compare the performance of any valid scheduling strategy. We will compare Chain to clairvoyant strategies.

THEOREM 4.1. *Let C denote the Chain scheduling strategy and A denote any clairvoyant scheduling strategy. Consider two otherwise identical systems, one using C and the other using A, processing identical sets of tuples with identical arrival times over operator paths having progress charts $\mathcal{P}'$. At every moment in time, the system using A will require at least as much memory as the system using C, implying the Chain strategy is optimal over any collection of lower envelopes.*

PROOF SKETCH: Since tuple arrival times are the same for each system, differences in memory requirements at time t will be due to the number of tuples that each system has been able to process by that time. Let $C(t)$ and $A(t)$ denote the number of tuples that have been "consumed" through processing (the total reduction in size over all tuples) by the two strategies at any time instant t. We wish to show that for all t, $C(t) \geq A(t)$.

Let $d_1 > d_2 > \cdots > d_l > 0$ denote the distinct slopes of the segments in the lower envelopes, arranged in a decreasing order. The slope of a segment is the fraction of a tuple *consumed* (reduction in the size of the tuple) when a tuple moves for a unit time along the segment. At any instant t, let t_i^C (respectively, t_i^A) denote the number of time units moved along all segments with slope d_i by the strategy C (respectively, strategy A). Since strategy C prefers to move along the segment with the steepest slope, and since by Proposition 3.1 the slopes are nonincreasing for any lower envelope, it follows that $t_1^C + t_2^C + \cdots + t_i^C \geq t_1^A + t_2^A + \cdots + t_i^A$ for $1 \leq i \leq l$.

The number of tuples that are consumed by strategy C is given by $C(t) = \sum_{1 \leq i \leq l} t_i^C d_i$, while the number consumed by A is given by $A(t) = \sum_{1 \leq i \leq l} t_i^A d_i$. Since $d_1 > d_2 > \cdots > d_l > 0$ and $t_1^C + t_2^C + \cdots + t_i^C \geq t_1^A + t_2^A + \cdots + t_i^A$ for $1 \leq i \leq l$, it follows that $C(t) \geq A(t)$. ■

We now consider the performance of Chain on general progress charts, beginning with the following observation:

PROPOSITION 4.2. *For all segments with a particular slope, Chain guarantees that there is at most one tuple that lies in the middle of one of these segments. The remaining such tuples must be at the beginning of their respective segments. Consequently, this strategy maintains the arrival order of the tuples.*

To see that Proposition 4.2 is true, recall that amongst all the tuples lying on the steepest-descent segment(s), Chain prefers to keep moving the tuple with the earliest timestamp, so it will keep moving this tuple until it has cleared the segment and moved to the next segment.

When Chain is implemented over a general progress chart P, it "pretends" that tuples move along the lower envelopes, although in reality the tuples move along the actual progress chart P. However, we show that this does not matter too much — the memory requirements of Chain on P are not much more than its memory

requirements on the lower envelope simulation P'. Consider a segment joining (t_i, s_i) to (t_{i+1}, s_{i+1}) on any of the lower envelopes of P. Let δ_i denote the maximum (over this segment) of the difference between the tuple-size coordinates (vertical axis) of P and its lower envelope P' for the same value of the time coordinate (horizontal axis).

LEMMA 4.3. *Let $C(t)$ denote the number of tuples that are consumed by the Chain strategy moving along the lower envelopes. Let $AC(t)$ denote the number of tuples that are* actually *consumed by Chain when tuples move along the actual progress charts. At any time instant t, $AC(t) \geq C(t) - \sum_i \delta_i$ where the sum $\sum_i \delta_i$ is taken over all segments corresponding to all lower envelopes.*

PROOF SKETCH: Consider a tuple making the same moves along the time axis for the two functions corresponding to the actual progress chart and the lower envelope. The size of the tuple is same when it is at the beginning of any segment. The size differs only if the tuple is in the middle of any segment. Moreover, for any segment i, the maximum difference is equal to δ_i, as per the definition of δ_i. Proposition 4.2 guarantees that the Chain strategy will have at most one tuple in the middle of any segment. Putting all this together, we obtain that $AC(t) \geq C(t) - \sum_i \delta_i$. ■

Another simple observation that we make about progress charts follows from the fact that the lower envelope always lies beneath the actual progress chart.

PROPOSITION 4.4. *For any progress made by a tuple along the progress chart P, if we use the lower envelope simulation of P to measure the memory requirement instead of the actual progress chart P, then we will always underestimate the memory requirement.*

It follows from Proposition 4.4 that the memory requirement for *any* strategy on progress chart P will be greater than the memory requirement of Chain on the lower envelope simulation of P, since we proved earlier (Theorem 4.1) that the Chain strategy is optimal over any collection of lower envelopes.

We can combine the preceding observations to prove a statement about the near-optimality of Chain on a general progress chart P. Since the performance of the Chain strategy over the lower envelope is a lower bound on the optimum memory usage (even for clairvoyant strategies), we obtain that the Chain strategy applied to the actual progress chart is optimal to within an additive factor of $\sum_i \delta_i$. For the important case where all operator selectivities are at most 1, which corresponds to queries where there are no non-foreign-key joins to stored relations, we can give a tight bound on $\sum_i \delta_i$ to obtain the following result:

THEOREM 4.5. *If the selectivities of all operators are at most 1 and the total number of queries is n, then $AC(t) \geq C(t) - n$.*

PROOF SKETCH: When the selectivities of all operators are at most 1, the progress chart is a non-increasing step function as shown in Figure 1(a). Then, for a segment joining (t_i, s_i) to (t_{i+1}, s_{i+1}), it must be the case that $\delta_i = s_i - s_{i+1}$. Consequently, the sum of δ_j over all segments j that belong to the same lower envelope (same query) equals 1. As a result, the sum $\sum_i \delta_i$ over all segments in all lower envelopes is bounded from above by the number of queries n. Combining this with Lemma 4.3 implies that $AC(t) \geq C(t) - n$. ■

Thus, when all selectivities are at most 1, Chain differs from optimal by at most one unit of memory per operator path. We emphasize that the guarantee is much stronger than merely saying that the maximum (over time) memory usage is not much more than the maximum memory usage of an optimal strategy. In fact, we are guaranteed that the Chain strategy will be off by at most one unit of memory (per query) as compared to *any clairvoyant* strategy (with an unfair knowledge of future tuple arrivals), at *all instants of time* and not just when we compare the maximum memory usage. This is a fairly strong worst-case bound on the performance of Chain. It is quite surprising that, even without the knowledge of future arrivals, Chain is able to maintain near-optimality at all time instants.

Our analysis of Chain is tight in that there exist cases where it will suffer from being worse than the optimal strategy by an additive factor of n.

4.2 Comparison With Other Strategies

Before proceeding to the case of queries joining multiple streams, we present other natural scheduling strategies against which we will compare Chain scheduling:

1. **Round-Robin:** The standard *Round-Robin* strategy cycles over the list of active operators and schedules the first operator that is ready to execute. On being scheduled, an operator runs until a fixed time quantum expires or an input queue to the operator becomes empty. In contrast to Chain (and other priority-based scheduling strategies) Round-Robin has the desirable property of avoiding starvation (i.e., no operator with tuples in its input queue goes unscheduled for an unbounded amount of time). With Chain, especially during bursts, ready-to-execute operators in low-priority chains may have to wait for a while before they are scheduled. However, the simplicity and starvation avoidance of Round-Robin come at the cost of lack of any adaptivity to bursts.

2. **FIFO:** The *FIFO* strategy (Example 2.1) processes input tuples in the order of arrival, with each tuple being processed to completion before the next tuple is considered. In general, FIFO is a good strategy to minimize the overall response time of tuples in the query result. Like Round-Robin, FIFO ignores selectivity and processing time of operators and shows no adaptivity to bursts.

3. **Greedy:** In the *Greedy* strategy, each operator is treated separately (as opposed to considering chains of operators) and has a static priority $(1 - s')/t'$, where s' is the selectivity and t' is the per-tuple processing time of that operator. This ratio captures the fraction of tuples eliminated by the operator in unit time. The problem with this strategy is that it does not take into account the position of the operator *vis-a-vis* other operators in the operator path. For instance, suppose a fast, highly selective operator H follows a few less selective operators. Although operator H will get high priority, the ones preceding it will not. As a result, at most time instants H will not be ready to be scheduled as its input queues will be empty. This demonstrates the need to prioritize earlier operators in an inductive manner, a notion that is captured by the lower envelope in Chain.

We conclude this subsection with some discussion points:

Pushing Down Selections. Query optimizers try to order operators so that more selective operators precede those that are less selective, making it likely that query plans have very selective operators early on. It is precisely on this type of query plan that Chain performs best compared to strategies such as FIFO and Round-Robin. The FIFO strategy, which does not exploit the low selectivity of operators at the beginning of a query plan, will accumulate

a large backlog of unprocessed tuples at the beginning of each operator path during bursty periods, as illustrated in Example 2.1. The Round-Robin scheduling strategy will have a similar problem since it treats all ready operators equally. Interestingly, Greedy will mirror Chain if the operators in the plan are in decreasing order of priority, when each operator will form a chain on its own. Still, non-commutativity of operators will sometimes result in query plans that favor Chain over Greedy. For example, tuple-based sliding-window joins like the ones we consider in this paper do not commute with most other operators including selections. Pushing a selection down below a tuple-based sliding-window join will change the result of the join by filtering out some tuples before they reach the join, slowing the rate at which tuples expire from the sliding window.

Starvation and Response Times. As mentioned earlier, the Chain strategy may suffer from starvation and poor response times, especially during bursts. As ongoing work, we are considering how to adapt our strategy to take into account additional objectives that we do not currently consider, such as the objectives of minimizing query-result latency and avoiding starvation. We suspect that a randomized strategy combining Chain with FIFO might do well for a metric that takes into account both memory and tuple latency.

Scheduling Overhead. Clearly, the scheduling overhead is negligible for simple strategies like Round-Robin and FIFO. In Chain, scheduling decisions need to be made only when an operator finishes processing all tuples in its input queue, or when a new block of tuples arrives in an input stream. In either case, the scheduling decision involves picking an operator from the highest-priority chain that contains a ready operator. Underlying progress charts and chain priorities need to be recomputed only when operator selectivities or tuple-processing times change; otherwise, the operator chains and their priority order is fixed. Recomputing progress charts from statistics and chains, and their priorities, from these progress charts takes very little time. Thus, Chain also incurs negligible overhead; Greedy behaves similarly.

Context Switching Overhead. The context switching overhead incurred by a scheduling strategy depends on the underlying query execution architecture. We assume that all operators and the scheduler run in a single thread. To get an operator to process its input queues, the scheduler calls a specific procedure defined for that operator. This query execution model is similar to the framework implemented in some recent data stream projects [19, 25]. Context switching from one operator to another is equivalent to making a new procedure call, which has low cost in modern processor architectures. Context switching costs can become significant if different operators are part of separate threads, as in Aurora [5]. Even if context switching costs are significant, we do not expect these costs to hamper the effectiveness of Chain. Compared to other scheduling policies like Round-Robin and FIFO, Chain tends to minimize the number of context switches. Chain will force a context switch only when an operator finishes processing all tuples in its input queue, or when a new block of tuples arrives in an input stream and unblocks a higher-priority operator than the one currently scheduled.

Throughput. All techniques perform the same amount of computation, considering that scheduling costs and context switching costs are negligible. We would expect throughput to be the same over time, provided the main memory threshold is not exceeded. During bursts, FIFO would momentarily produce more result tuples compared to the other strategies.

The experimental results in Section 5 validate the intuitive statements and analytical results presented in this section.

4.3 Multi-Stream Queries

We now extend Chain scheduling to queries with multiple streams that contain at least one sliding-window join between two streams. For presentation, in this particular section a "tuple" refers to a single stream tuple, as opposed to a larger unit of memory such as a page, as is the case in the rest of the paper.

Recall that we assume tuple-based sliding-window joins. We assume that every tuple has a globally unique timestamp (across all streams), and that tuples within a single stream arrive in increasing order of timestamp. For instance, the stream system might be timestamping tuples when they arrive in the system. A procedural description of the result of a tuple-based sliding-window join is as follows: For a join between streams R and S, when a tuple arrives on stream R, it will be joined with the latest w_S tuples in S (where w_S is the size of the sliding window on stream S). Symmetric processing occurs for tuples arriving on stream S. When tuples with timestamps t_1 and t_2 are joined, the timestamp of the result tuple is $max(t_1, t_2)$.

The synchronization inherent in the above semantics restricts freedom in operator scheduling. To guarantee correctness and ensure that join output is produced in sorted timestamp order, we need to synchronize the two inputs to a join by processing them in strict timestamp order across both input streams (similar to a *merge sort* on the timestamp attribute). In other words, when joining streams R and S, no tuple from R with timestamp t will be processed by the join operator until it has processed all tuples in S with timestamp less than t. As a result, a sliding-window join operator will block if any one of its input queues is empty, even if tuples are available in the other input queue.

4.3.1 Extending Chain Scheduling to Joins

In order to extend Chain scheduling, we first need to extend the progress chart model to multi-stream queries. A query with multiple streams is a rooted tree with input streams at the leaves of the tree. We break up the tree into parallel operator paths, one for each input stream, that connect individual leaf nodes (representing input streams) to the root of the tree. The operator paths thus obtained can share common segments. Each operator path is individually broken up into chains for scheduling purposes. For example, consider a query with a sliding-window join ($\bowtie$) between two streams R and S, followed by a selection condition (σ) on the output of the join. Additionally, there is a project operator (π) on the stream R before it joins S. The decomposition of the tree corresponding to this query gives two operator paths $R \rightarrow \pi \rightarrow \bowtie \rightarrow \sigma$ and $S \rightarrow \bowtie \rightarrow \sigma$. The $\bowtie \rightarrow \sigma$ segment is shared between the operator paths. Note that the join operator is part of both operator paths and will therefore be part of two operator chains when these paths are broken up into chains for scheduling. However, as discussed earlier in this section, the join operator always processes tuples in strict timestamp order across its input queues irrespective of the chain, as part of which, it gets scheduled. Furthermore, the sliding-window join operator will block if any one of its input queues is empty.

The per-tuple processing times (t) and selectivities (s) for all operators other than join are defined in a straightforward manner, similar to the case of single-stream queries. We now specify the quantities (t, s) for a join operator between two streams.

A sliding-window join is abstracted using a model similar to

the one described in [14]. Let the average number of tuples in a stream S per unit time (as per the timestamp on tuples) be λ_S. Like average tuple selectivity, λ_S is a convenient abstraction to capture the average behavior of sliding-window join operators. Consider a sliding-window join operator between streams R and S that processes tuples with timestamps belonging to an interval of size t'. During this run, the join operator will process (on an average) $t'(\lambda_R + \lambda_S)$ input tuples and produce $t'(\lambda_R \alpha_{W(S)} + \lambda_S \alpha_{W(R)})$ result tuples, where $\alpha_{W(S)}$ is the (average) selectivity of the semi-join of stream R with the sliding window for S, i.e., the average number of tuples from the sliding window for S that a tuple from R joins with. $\alpha_{W(R)}$ is defined analogously. The system time taken for this run is $t' \times (\lambda_R \times t_R + \lambda_S \times t_S)$, where t_X is the average time taken to process a tuple from stream X. Here t_S includes the time taken to compare the head tuples in the queues for R and S, the time to probe the sliding window for R and produce result tuples, and the time to update the sliding window for S. Given $\lambda_R, \lambda_S, \alpha_{w(R)}, \alpha_{w(S)}, t_R, t_S$, for input streams R and S, the selectivity s for the sliding-window join is given by $\frac{\lambda_R \alpha_{W(S)} + \lambda_S \alpha_{W(R)}}{\lambda_R + \lambda_S}$ and per-tuple processing time t (wall clock time) is given by $\frac{\lambda_R \times t_R + \lambda_S t_S}{\lambda_R + \lambda_S}$. It is easy to inductively derive $\lambda_{S'}$ for any stream S' that is the result of an intermediate operator in the query plan.

Having specified the (t, s) values for the windowed join operator between two streams, we build the progress chart for each operator path as described in Section 3. Our basic Chain strategy remains unchanged. The only difference is the following: earlier an operator could be blocked (cannot be scheduled) only when input queue is empty. However, in the case of a join, the left input queue might have tuples while the right input queue could be empty. In such a case, the chain corresponding to the left queue might want to schedule the join operator, but it cannot do so because the operator is blocked for input on the right input queue. This chain will not be considered for scheduling until the join operator becomes unblocked.

As before, the scheduling strategy is executed whenever an operator finishes processing all tuples in one of its input queues, or when a new block of tuples arrives in some input stream, and the highest-priority ready chain is scheduled as always.

Unlike the single-stream case, we do not have any analytical results for our adaptation of Chain to the multiple-stream case. However, experimental results suggest that Chain performs extremely well, for both single-stream and multiple-stream queries, compared to the other scheduling strategies that were considered.

5. EXPERIMENTS

In this section we briefly describe our simulation framework and the results of various experiments that we conducted to compare the performance of different scheduling policies: Chain, FIFO, Greedy, and Round-Robin.

We begin with a brief description of our simulation framework. We ignore the cost of context switching and scheduling overhead in our simulations. As discussed in Section 4.2, Chain makes fewer context switches as compared to FIFO, Greedy, or Round-Robin. Therefore, had we included context-switching costs, the relative performance of Chain would have been even better than shown here.

FIFO processes each block of input stream tuples to completion before processing the next block of tuples in strict arrival order. Round-Robin cycles through a list of operators and each ready operator is scheduled for one time quantum. The size of the time quantum does affect the performance of Round-Robin, but it does not change the nature of the results presented here.

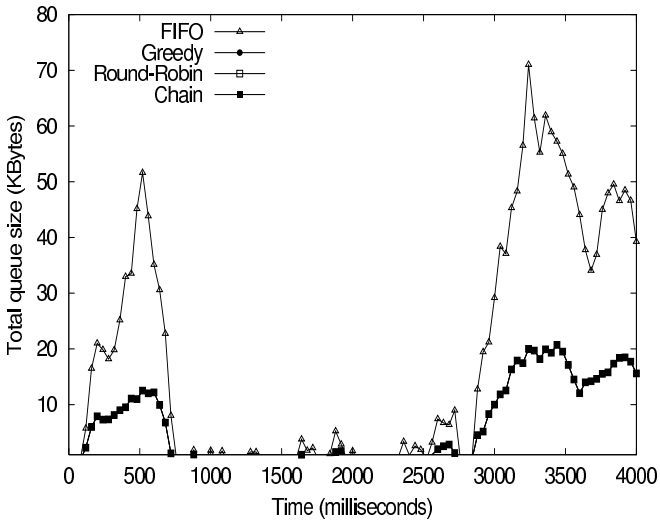


Figure 2: Queue size vs. Time (Single stream, two operators, real data set)

The notion of progress chart captures the average behavior of the query execution in terms of the sizes of memory units as they make progress through their operator paths. The experiments we describe were designed by choosing a particular progress chart to use for both the real and synthetic data sets and then adjusting selection conditions and join predicates to closely reflect the progress chart. Of course, during actual query execution there are short-term deviations from the average behavior captured in the progress chart. In our experiments, we follow the query execution and report the memory usage at various times.

The experiments described here used static estimates for operator selectivities and processing times (derived from a preliminary pass over the data) to build the progress charts. We also performed experiments that allowed adaptivity by dividing time into windows and using the statistics gathered during the ith time window to build progress charts for the $(i+1)$st time window. The results were similar to our other experiments, and we do not report them here.

Next we briefly describe the data sets that were used in the various experiments:

1. **Synthetic Data Set:** The networking community has performed considerable research on how to model bursty traffic to most closely approximate the distributions prevalent in most real data sets. A good reference is the paper by Willinger, Paxson, et al. [33]. Based on their "ON/OFF" model we generate synthetic bursty traffic by flows that begin according to a Poisson process (with mean inter arrival time equal to 1 time unit) and then send packets continuously for some duration chosen from a heavy-tailed distribution. We used the Pareto distribution for packet durations, which has a probability mass function given by $p(x) = \alpha k^\alpha x^{-\alpha-1}$, for $\alpha, k > 0, x \geq k$. We used $k = 1$ and $\alpha = 2.3$ in our experiments. While the arrival times are generated as above, the attribute values are generated uniformly from a numeric domain; this allows us to choose predicates with desired selectivities.

2. **Real Data Set:** The Internet Traffic Archive [11] is a good source of real-world stream data sets. One of their traces, named "DEC-PKT", contains one hour's worth of all wide-

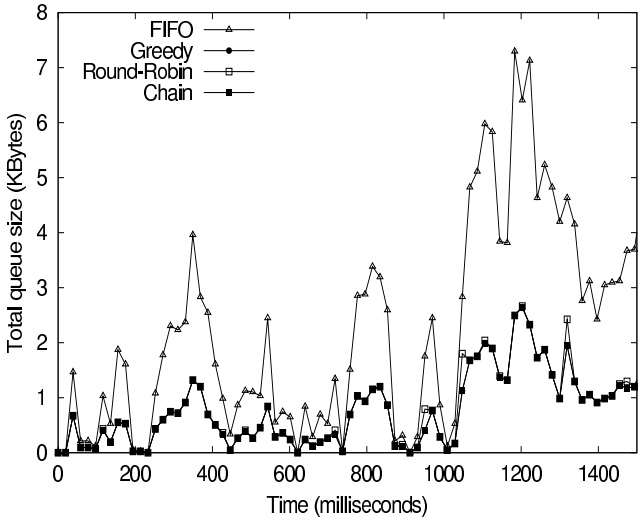


Figure 3: Queue size vs. Time (Single stream, two operators, synthetic data set)

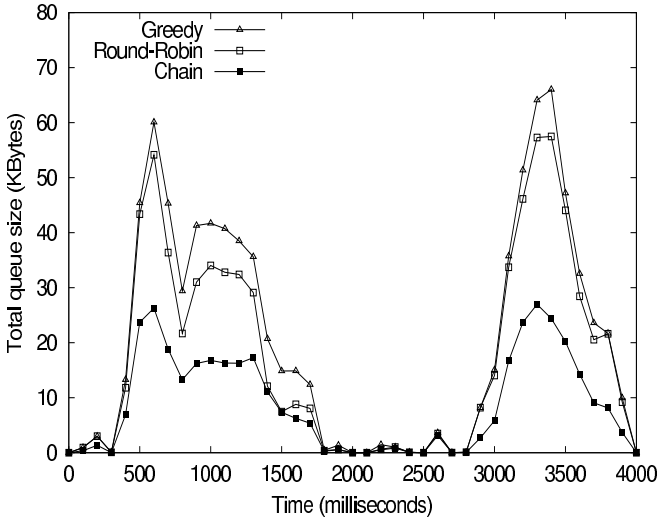


Figure 4: Queue size vs. Time (Single stream, four operators, real data set)

area traffic between Digital Equipment Corporation and the rest of the world. We use this trace as the real-world data set for our experiments. Attributes such as IP addresses and packet sizes were used in selection and join predicates. The exact predicates were chosen to give the desired selectivities for each experiment.

5.1 Single-Stream Queries without Joins

Our first experimental results compare the performance of different scheduling strategies for single-stream queries without joins. We first consider a simple query with two operators. Its progress chart in terms of coordinates (t_i, s_i) of operator points is: $(0, 1)$, $(500, 0.3)$, and $(4000, 0)$, where times are in microseconds. In terms of the terminology in Section 3, a *tuple* in the progress chart contains 100 individual tuples of size 24 bytes each. This query consists of a fast and highly selective operator followed by a slow operator that "consumes" much fewer tuples per unit time. This is similar to Example 2.1.

Figures 2 and 3 show the variation in total queue size over time for the real and synthetic data set, respectively. We observe that Chain and Greedy have almost identical performance for this simple query plan. This is explained by the fact that each operator forms a chain of its own and hence they are expected to behave identically. Round-Robin performs almost as well as Chain on both data sets. In later experiments we will see how as the number of operators increase, the performance of Round-Robin degrades. FIFO performs badly with respect to Chain since it under-utilizes the fast and highly selective first operator during any burst.

The second query that we consider has four operators with selectivities less than 1. Its progress chart in terms of coordinates (t_i, s_i) of operator points is: $(0, 1)$, $(400, 0.9)$, $(2000, 0.88)$, $(2200, 0.1)$, and $(4000, 0)$. The third operator is fast and highly selective and it comes between two operators that have much lower tuple consumption per unit time. This is a typical scenario where Greedy is expected to perform badly as compared to Chain. Indeed, we observe this in Figures 4 and 5, which show the variation in total queue size over time for this query on the real and synthetic data sets, respectively. (For legibility, we have not shown the performance of FIFO and Round-Robin in Figures 4 and 5 respectively, both of which perform nearly as bad as Greedy in either case.) Because Greedy does not schedule the less selective second operator during bursts, the fast and selective third operator remains underutilized, explaining Greedy's bad performance. Because it uses the lower envelope to determine priorities, Chain is able to recognize the usefulness of the third operator although it is sandwiched between two "unproductive" operators. Notice that unlike the previous case Round-Robin does badly compared to Chain in Figure 4.

5.2 Queries having Joins with Stored Relations

Recall from Section 2 that a join with a relation could result in an operator with selectivity strictly greater than one. The real data set that we worked with did not include stored relations, so we report experimental results over synthetic data only. The progress chart used here in terms of coordinates (t_i, s_i) of operator points is: $(0, 1)$, $(400, 0.9)$, $(1300, 2.0)$, $(1500, 0.2)$, and $(4000, 0)$, where the second operator is a join with a stored relation. Figure 6 shows the performance of different scheduling strategies for the bursty synthetic data. (For legibility, we have not shown the performance of Round-Robin in Figure 6, which performs as badly as Greedy. Also, we have not connected the points corresponding to FIFO by line segments.) Since FIFO and Round-Robin do not take operator selectivities into account, their performance remains more or less similar to what we observed in the previous experiments (Section 5.1). Because of the low priority of the join operator, during bursts Greedy does not utilize the fast and selective operator which follows the join. On the other hand, the first three operators comprise a single chain in Chain, so the fast and selective third operator does get used during bursts, leading to vastly improved performance over Greedy (Figure 6).

5.3 Queries with Sliding-Window Joins Between Streams

We study the performance of the different strategies for a query over two streams R and S that are joined by a sliding-window join. Both semijoins in the sliding-window join have an average selectivity of 2. The output of the windowed join passes through two selection conditions σ_2 and σ_3. Furthermore, before joining with S, stream R passes through a selection condition σ_1. The selection conditions σ_1 and σ_3 are not very selective, while σ_2 (the selection

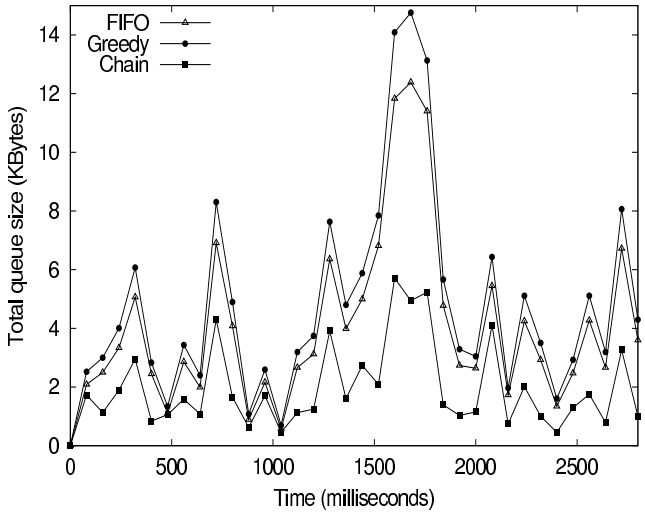


Figure 5: Queue size vs. Time (Single stream, four operators, synthetic data set)

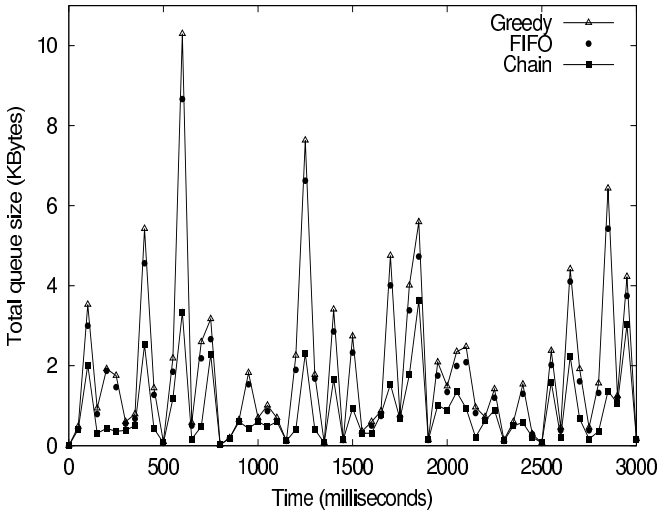


Figure 6: Queue Size vs. Time (single stream, $s > 1$, synthetic data)

following the join) is very selective. The performance graphs for this query over the real and synthetic data sets are shown in Figures 7 and 8 respectively. We observe that Greedy and FIFO perform much worse than Chain. Round-Robin compares well with Chain for the real data set but does really badly on the synthetic data set. As in the experiment described in Section 5.2, Greedy does badly because of the sliding-window join preceding the highly selective operator. The low priority of the join discourages Greedy from scheduling it so Greedy under-utilizes the highly selective operator following it. FIFO performs badly for the reasons mentioned earlier, namely the presence of a less selective operator (σ_3) with relatively high tuple processing time.

5.4 Multiple Queries

Finally, we compared the performance of different strategies over a collection of 3 queries: a sliding-window join query similar to the one presented in the last experiment (Section 5.3) and two single-stream queries with selectivities less than 1 similar to those presented in Section 5.1. The performance graphs for this query workload over real and synthetic data sets are shown in Figures 9 and 10 respectively. For improved legibility of Figure 9, we have not connected the points corresponding to Chain and FIFO by line segments. The graphs clearly show that Chain beats all others by a huge margin. The reason for the impressive performance of Chain is the increase in complexity and size of the underlying problem. This is because Chain is able to pick out the particular chain of operators that is most effective at reducing memory usage and will schedule it repeatedly during a burst of input arrival. On the other hand, since there are a larger number of operators in this multi-query experiment compared to the earlier single-query experiments, Round-Robin ends up executing the best operator much less frequently than if there were a lesser number of operators. In other words, as the number of operators increases, the fraction of time Round-Robin does the "right thing" decreases. The same statement holds for FIFO as well. Greedy performs badly for reasons mentioned earlier: the queries consist of highly selective operators sandwiched between not so selective ones, which Chain recognizes, but Greedy fails to recognize. This experiment suggests that as we increase the number of queries the benefits of an intelligent strategy like Chain become more pronounced. Indeed, the results in Figures 9 and 10 were obtained by going from a single query to a collection of merely three queries. In a real system with many more queries, the benefits will undoubtedly be even greater.

6. RELATED WORK

Recently, there has been considerable research activity pertaining to stream systems and data stream algorithms. An overview is provided in the recent survey paper by Babcock et al. [3]. Most closely related to our paper is the suite of research in adaptive query processing. (See the IEEE Data Engineering Bulletin special issue on Adaptive Query Processing [17]). Some of the recent papers from the Telegraph project [2, 6, 18, 24] pertain to novel architectures and strategies for adaptive query processing over streams. However, while their research focuses primarily on adapting to changing *data characteristics*, we focus on adapting to changing *arrival characteristics* of data, in particular the bursty nature of data as documented in the networking community [9, 16, 33, 34]. Earlier work on adaptive query processing includes the query scrambling work by Urhan et al. [31], the adaptive query execution system Tukwila [12] for data integration, and mid-query re-optimization techniques developed by Kabra and DeWitt [13]. More closely related to ours is the work on dynamic query operator scheduling by Amsaleg et al. [1] aimed at improving response times in the face of unpredictable and bursty data arrival rates, the Xjoin operator of Urhan and Franklin [29] which is optimized to reduce initial and intermittent delay, and the work on dynamic pipeline scheduling for improving interactive performance of online queries [30]. However, in all these cases, the focus is on improving response times, and run-time memory minimization is not the consideration.

Various operator scheduling strategies have been suggested for stream systems, ranging from simple ones like round-robin scheduling [19] to more complex ones that aim at leveraging intra- and inter-operator nonlinearities in processing [5]. However, to the best of our knowledge, none have considered the problem of scheduling with the aim of minimizing memory usage. Also, somewhat related to our work is the set of papers [5, 6, 14, 18] championing the use of sliding-window joins for stream systems. The rate-based optimization framework of Viglas and Naughton [32] considers the problem of static query optimization with the modified aim of maximizing the throughput of queries for stream systems; however, it

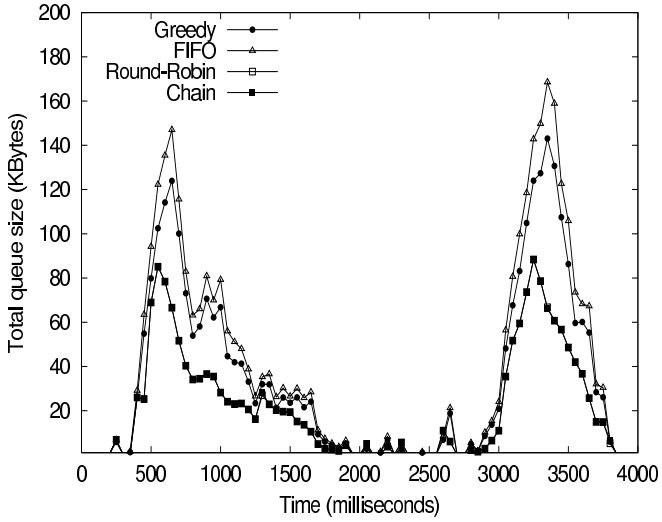


Figure 7: Queue size vs. Time (sliding-window join and 3 selections, real data set)

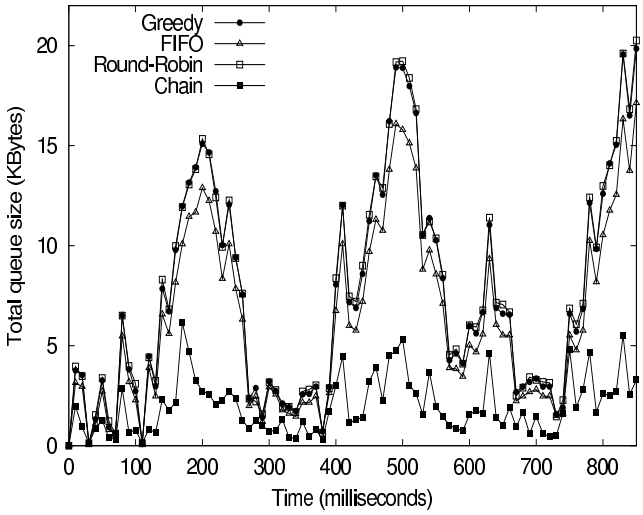


Figure 8: Queue size vs. Time (sliding-window join and 3 selections, synthetic data set)

does not address run-time scheduling.

The problem of allocating main memory among concurrent operators in a traditional DBMS in order to speed up query execution has been considered in [4, 8, 20]. These techniques do not extend directly to data stream systems.

7. CONCLUSION AND OPEN PROBLEMS

We studied the problem of operator scheduling in data stream systems, with the goal of minimizing memory requirements for buffering tuples. We proposed the Chain scheduling strategy and proved its optimality for the case of single-stream queries with selections, projections, and foreign key joins with static stored relations. Furthermore, we showed that Chain scheduling performs well for other types of queries, including queries with sliding-window joins. As ongoing work, we are considering an adaptation of our strategy to take into account additional objectives that we do not currently consider, such as minimizing latency and avoiding starvation. We suspect that a randomized strategy combining Chain with FIFO might do well for a metric that takes into consideration both memory requirements and tuple latency. Other open problems include designing scheduling strategies for adaptive query plans, where the structure of the query plan itself is allowed to change over time, and considering sharing of computation and memory in query plans.

8. REFERENCES

[1] L. Amsaleg, M. Franklin, and A. Tomasic. Dynamic query operator scheduling for wide-area remote access. *Journal of Distributed and Parallel Databases*, 6(3), July 1998.

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

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, June 2002.

[4] L. Bouganim, O. Kapitskaia, and P. Valduriez. Memory-adaptive scheduling for large query execution. In *Proc. of the 1998 ACM CIKM Intl. Conf. on Information and Knowledge Management*, pages 105–115, Nov. 1998.

[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*, Aug. 2002.

[6] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, Aug. 2002.

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

[8] B. Dageville and M. Zait. SQL memory management in Oracle9i. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, Aug. 2002.

[9] S. Floyd and V. Paxson. Wide-area traffic: The failure of poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[10] J. Hellerstein, M. Franklin, S. Chandrasekaran, A. Deshpande, K. Hildrum, S. Madden, V. Raman, and M. A. Shah. Adaptive query processing: Technology in evolution. *IEEE Data Engineering Bulletin*, 23(2):7–18, June 2000.

[11] Internet Traffic Archive. http://www.acm.org/sigcomm/ITA/.

[12] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[13] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 106–117, June 1998.

[14] J. Kang, J. F. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[15] J. Kleinberg. Bursty and hierarchical structure in streams. In *Proc. of the 2002 ACM SIGKDD Intl. Conf. on Knowledge*

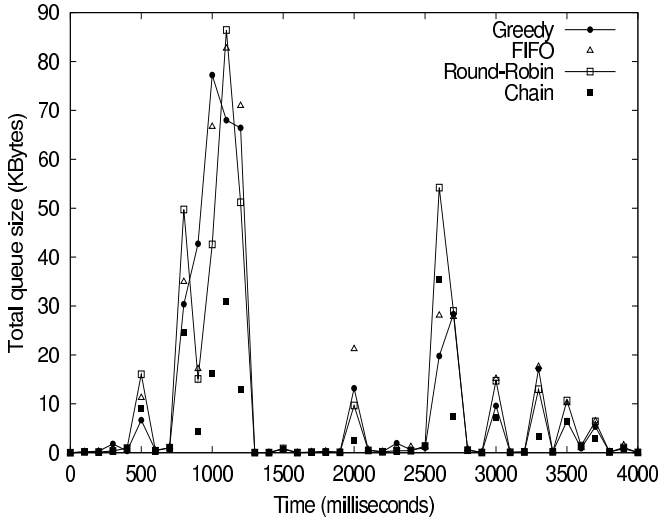


Figure 9: Queue size vs. Time (plan with multiple queries, real data set)

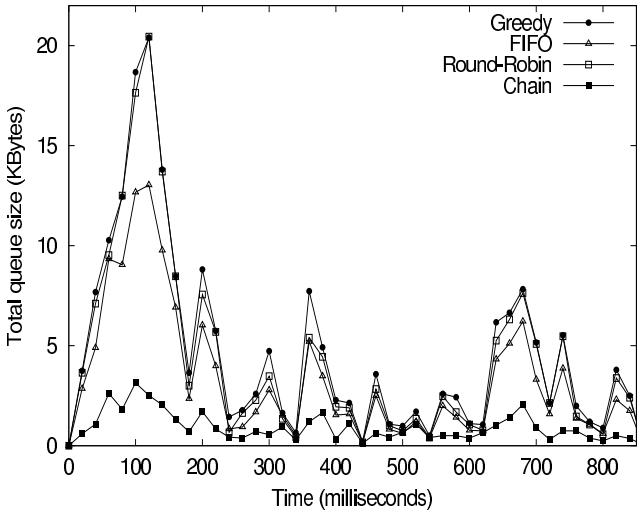


Figure 10: Queue size vs. Time (plan with multiple queries, synthetic data set)

Discovery and Data Mining, Aug. 2002.

[16] W. Leland, M. Taqqu, W. Willinger, and D. Wilson. On the self-similar nature of ethernet traffic. *IEEE/ACM Transactions on Networking*, 2(1):1–15, Feb. 1994.

[17] D. Lomet and A. Levy. Special issue on adaptive query processing. *IEEE Data Engineering Bulletin*, 23(2), June 2000.

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

[19] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[20] B. Nag and D. J. DeWitt. Memory allocation strategies for complex decision support queries. In *Proc. of the 1998 ACM CIKM Intl. Conf. on Information and Knowledge Management*, pages 116–123, Nov. 1998.

[21] Niagara Project. http://www.cs.wisc.edu/niagara/.

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

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

[24] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[25] M. Shah, S. Madden, M. Franklin, and J. Hellerstein. Java support for data-intensive systems: Experiences building the telegraph dataflow system. *SIGMOD Record*, 30(4):103–114, Dec. 2001.

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

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

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

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

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

[31] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost-based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

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

[33] W. Willinger, V. Paxson, R. Riedi, and M. Taqqu. Long-range dependence and data network traffic. In *Long-range Dependence: Theory and Applications, P. Doukhan, G. Oppenheim and M. S. Taqqu, eds., Birkhauser*, 2002.

[34] W. Willinger, M. Taqqu, and A. Erramilli. A bibliographical guide to self-similar traffic and performance modeling for modern high-speed networks. In F. P. Kelly, S. Zachary, and I. Ziedins, editors, *Stochastic Networks: Theory and Applications*, pages 339–366. Oxford University Press, 1996.

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