The VLDB Journal (2004) / Digital Object Identifier (DOI) 10.1007/s00778-004-0132-6

# Operator scheduling in data stream systems

**Brian Babcock**[⋆], **Shivnath Babu**[⋆⋆], **Mayur Datar**[⋆⋆⋆], **Rajeev Motwani**[†], **Dilys Thomas**[‡]

Stanford University, ♣ please insert full address (e-mail: {babcock, shivnath, datar, rajeev, dilys}@cs.stanford.edu)

Edited by ♣. Received: ♣/ Accepted: ♣
Published online: ♣♣ 2004 – 

**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 runtime system memory usage as well as output latency. Our aim is to design a scheduling strategy that minimizes the maximum runtime system memory while maintaining the output latency within prespecified bounds. We first present *Chain scheduling*, an operator scheduling strategy for data stream systems that is near-optimal in minimizing runtime 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. However, during bursts in input streams, when there is a buildup of unprocessed tuples, Chain scheduling may lead to high output latency. We study the online problem of minimizing maximum runtime memory, subject to a constraint on maximum latency. We present preliminary observations, negative results, and heuristics for this problem. A thorough experimental evaluation is provided where we demonstrate the potential benefits of Chain scheduling and its different variants, compare it with competing scheduling strategies, and validate our analytical conclusions.

**Keywords:** Data treams – Scheduling – Memory management – Latency

## 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: (i) the volume of data is extremely high and (ii) on the basis of the data, decisions are arrived at and acted upon in close to real time. The combination of these two factors makes traditional solutions, where data are 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 [6], ranging from algorithms for data streams to full-fledged data stream systems such as Aurora [10], Gigascope [24], Hancock [14], Niagara [39], STREAM [45], Tangram [40,41], Tapestry [48], Telegraph [19], Tribeca [46], 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 runtime resource allocation and optimization. In this paper we focus on one aspect of runtime resource allocation, namely, *operator scheduling*.

There are some features unique to data stream systems, as opposed to traditional relational DBMSs, that make the runtime 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

⋆ Supported in part by a Rambus Corporation Stanford Graduate Fellowship and NSF Grant IIS-0118173.
⋆⋆ Supported in part by NSF Grants IIS-0118173 and IIS-9817799.
⋆⋆⋆ 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.
‡ Supported by NSF Grant EIA-0137761 and NSF ITR Award Number 0331640.

arrival rates over time. This phenomenon has been extensively studied in the networking context [17,32,54]. 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 [29]. Consequently, conditions in which data stream queries are *executed* are frequently quite different from the conditions in 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 [37], or take a holistic approach to adaptivity and do away with fixed query plans altogether, as in the Eddies architecture [3,34]. Like [1,49,51], in this paper we focus on adaptivity toward changing *arrival* characteristics of the data. 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 while ensuring that output latency is bounded.

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 a sudden decrease in overall system throughput As we will show in Sect. 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. We initially focus entirely on memory requirements, ignoring other metrics. However, because it is also important that a stream system ensure reasonable output latency, we formalize this latency requirement as a constraint that output latency should not exceed a certain threshold and study the constrained version of the memory minimization problem in the second part of the paper.

### *1.1 Road map*

The rest of this paper is organized as follows. We begin in Sect. 2 by briefly describing our model for the processing of stream queries and illustrating through an example that the runtime memory requirement can be greatly affected by the operator scheduling strategy, thereby motivating the need for an intelligent scheduling strategy. In Sect. 3 we formalize the problem of operator scheduling that we consider. Our analytical results on the runtime memory requirements of proposed scheduling strategies are provided in Sect. 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 Sect. 4.2 we introduce various other scheduling strategies and compare them qualitatively. In Sect. 4.3 we extend the operator scheduling strategies to queries involving windowed joins over multiple streams. In Sect. 5 we study a constrained version of the operator scheduling problem that takes output latency into consideration, presenting preliminary observations, negative results, and heuristics. In Sect. 6 we describe our experimental comparison of the performance of different scheduling strategies. We describe related work in Sect. 7 and conclude in Sect. 8.

## 2 Stream query processing: model and assumptions

Data stream systems are characterized by the presence of multiple continuous queries [10,12,34,37]. Query execution can be conceptualized as a *data flow diagram* (as in [10]), 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 the 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 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.

The use of interoperator queues is somewhat nonstandard in data processing systems. In the traditional iterator model, the flow of control within a query plan is "pull driven", with lower-level operators recursively invoked by the top-level operator. By contrast, the use of loosely coupled query operators connected by queues allows for much greater flexibility in the order in which operators are scheduled. The job of the operator scheduler is to select the order in which the query operators should be invoked; one option is to emulate the traditional "pull-driven" ordering, but many other scheduling strategies exist. In this paper, we show how this scheduling flexibility can be exploited to achieve a specific performance objective: minimizing memory usage.

The job of the operator scheduler is to select the order in which query operators are executed. We assume a single-processor system in which one operator is active at a time. Additionally, to avoid excessive context switching overhead, we assume that when an operator is scheduled, it should be allowed to run for at least some minimum time quantum before being preempted. Thus time can be seen as proceeding in discrete time steps equal in length to this minimum time quantum, and the role of the scheduler is to decide which query operator runs during each time step.

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 [55]. While operators like select and project do not need to maintain any runtime state, a symmetric hash join operator needs to maintain a 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 [10,12,26,34,37], 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 [26,37]. 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* that 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 average) produces $s$ tuples on processing 1 tuple. We assume that the runtime 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. In terms of this model, the goal of operator scheduling is to minimize the total memory requirement of all queues in the system subject to a user-specified latency constraint.

### 2.1 Alternate techniques for handling bursts

An important strategy for handling rapid and bursty stream arrival is *load shedding*, where input tuples are dropped based on some *quality-of-service* criteria to bring the system load down to manageable levels when the system is overloaded [7, 47]. 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. Load shedding is not the focus of this paper. The techniques for load shedding proposed in [7,47] are orthogonal to the operator-scheduling techniques we discuss in this paper and could be used in conjunction with our techniques. Instead, we focus on the following issue: Even when 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. We want to schedule operators efficiently in order to keep the peak memory usage at a minimum. We assume that the average arrival rate is within computational limits so that it is eventually possible to clear the backlog of unprocessed tuples, e.g., when the bursts of high arrival rate have receded.

An alternative approach to handling fast data stream arrival rates is by *rate throttling*, i.e., artificially restricting the rate at which tuples can be delivered to the system, thus shifting the burden of buffering unprocessed data from the data stream processing system to the data stream sources. However, many types of data stream sources may have little or no ability to regulate the rate at which they deliver streaming data. High-volume data streams will frequently be transmitted using UDP packets for efficiency, so the flow control capabilities of TCP may not be available. Furthermore, even data stream sources that are capable of responding to backpressure often have limited buffering capabilities (e.g., when the source is a low-cost sensor or embedded system). For these reasons, the rate throttling approach is not further considered in this paper.

### 2.2 Why operator scheduling matters

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 and lower than the system capacity, then there is not much to be done in terms of scheduling. Each tuple that arrives can be processed completely before the next tuple arrives, so 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 the memory threshold or not.

*Example 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 will not have an unbounded buildup 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 in which they arrive. A tuple is passed through both operators

[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 Sect. 3.

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 a limited amount of memory. Since the FIFO strategy is the only one available for systems that use the traditional, iterator-based model of query operators, the example also illustrates why buffering tuples in interoperator queues can be beneficial. 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.

## 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*. Every operator is a filter that operates on a tuple and produces $s$ tuples, where $s$ is the operator selectivity. (For operators other than the first one in an operator path, we are interested in the *conditional* selectivity, that is, the operator's selectivity on those tuples that have already passed through all the previous operators.) 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 a 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 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 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 Fig. 1.

The horizontal axis of the progress chart represents time and the vertical axis represents tuple size. The $m+1$ *operator points* $(t_0, s_0), (t_1, s_1), \ldots, (t_m, s_m)$, where $0 = t_0 < t_1 < t_2 < \ldots < t_m$ are positive integers, represent an operator path consisting of $m$ operators, where the $i$th $(1 \leq i \leq m)$ 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 $s_i/s_{i-1}$.

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 $\tau$ enters the system with size $s_0 = 1$. After being processed by the $i$th query operator, the tuple has received $t_i$ total processor 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 $i$th horizontal segment represents the execution of the $i$th 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

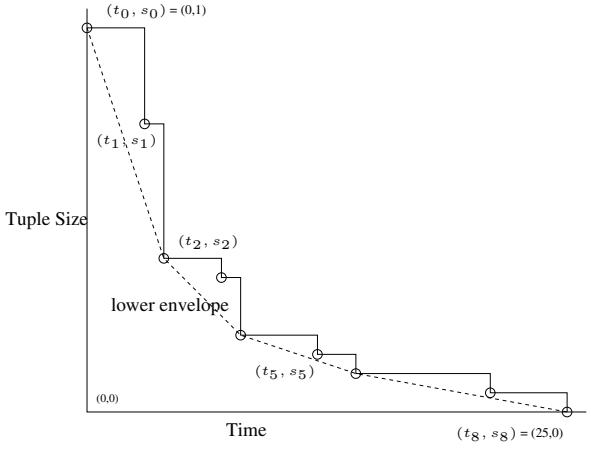


**Fig. 1.** Progress chart

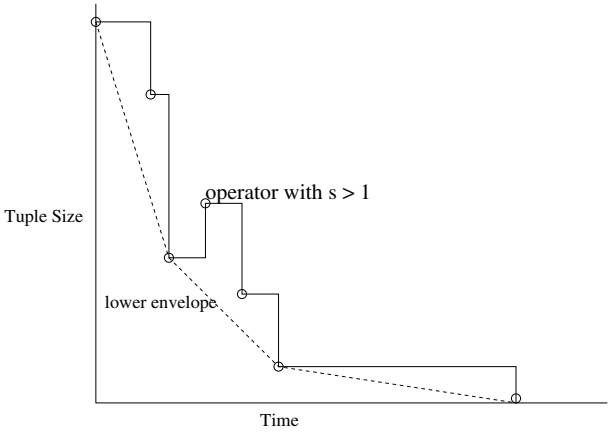


**Fig. 2.** Progress chart: one operator has selectivity $s > 1$

selectivities are less than or equal to 1, the progress chart is nonincreasing, as shown in Fig. 1. However, for query plans that include a join operator with selectivity greater than 1, the progress chart looks like the one shown in Fig. 2.

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 [3]: divide time into fixed windows and collect statistics independently in each window; use the statistics from the $i$th 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 $j$th 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 point $(t, s)$ to an operator point to its right. The *steepest* derivative at point $(t, s)$ for $t_{i-1} \leq t < t_i$ is denoted by $D(t, s) = \max_{m \geq j \geq i} d(t, s, j)$, and the operator point for which the maximum is achieved 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 $d(t, s, j) = D(t, s)\}$.[2]

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 Figs. 1 and 2, 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 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 nonincreasing with $i$.*

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

*Proof.* The proof is by contradiction. 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. □

We are now ready to present our operator-scheduling strategies. In Sect. 4 we focus on scheduling operators so as to minimize the total memory requirement of all queues in the system. We consider the output latency metric in Sect. 5.

## 4 Scheduling strategies for runtime memory minimization

We begin by defining a framework for specifying an operator-scheduling strategy. Ideally, one can view a scheduling strategy as being invoked at the end of every unit of time, where a time unit is the smallest duration for which an operator should be run before it is replaced by another operator. 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 section 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 strategies 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, on the basis of 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 Sect. 4.1.
2. **Multistream 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. Multistream queries are discussed in Sect. 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 $\tau$ 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 $\tau$ *lies on* $l_i$ if $t_i \leq p(\tau) < t_{i+1}$. (Recall from Sect. 3 that $p(\tau)$ denotes the progress made by $\tau$ along the operator path.) Moreover, we say that $\tau$ 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 that 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 that 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.* 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 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 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 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 2 is true, recall that among 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 $\delta_i$ 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 4.2** *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.* 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 the 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 $\delta_i$, as per the definition of $\delta_i$. Proposition 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 3** *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 3 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 4.3** *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.* When the selectivities of all operators are at most 1, the progress chart is a nonincreasing step function, as shown in Fig. 1. Then, for a segment joining $(t_i, s_i)$ to $(t_{i+1}, s_{i+1})$, it must be the case that $\delta_i \leq s_i - s_{i+1}$. Consequently, the sum of $\delta_j$ 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 4.2 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. (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 $n$.) 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.

Even with clairvoyant knowledge of future tuple arrival patterns, no algorithm can efficiently determine the optimal schedule (assuming $P \neq NP$), as demonstrated by the following theorem:

**Theorem 4.4** *The offline problem of scheduling operators to minimize required memory is NP-complete.*

*Proof.* The memory requirements of a schedule can be easily verified in polynomial time, so the problem of finding the schedule that minimizes the memory required is in the class NP.

We will prove that the problem is NP-complete via a reduction from the knapsack problem. In the knapsack problem, the inputs are a knapsack capacity $C$ and a set of $n$ items $X = \{x_1, x_2, \ldots, x_n\}$ with associated *sizes* $s_1, s_2, \ldots, s_n$ and *benefits* $b_1, b_2, \ldots, b_n$. The objective is to find the subset of items $S \subset I$ that have maximum total benefit $\sum_{x_i \in S} b_i$ subject to the constraint that the total size $\sum_{x_i \in S} s_i$ cannot exceed the capacity $C$.

Construct $n$ progress charts each with two operators. The $i$th progress chart consists of an operator that takes time $s_i$ and has selectivity $1 - b_i$ followed by an operator that takes time $C + 1 - s_i$ and has selectivity zero. Consider the following sequence of tuple arrivals: at time 0 and at time $C$, one tuple arrives on each of the $n$ progress charts. No tuples arrive between time 0 and time $C$ and no tuples arrive after time $C$.

It is easy to see that the maximum memory usage occurs at time $C$. If $M$ denotes the amount of memory freed up due to processing during the first $C$ time units, then the amount of memory used at time $C$ is equal to $2n - M$. Processing both operators in a single progress chart to completion requires $C+1$ time units, so no more than one operator in each progress chart can be processed to completion within $C$ time units. Completing the first operator in the $i$th process chart frees $b_i$ units of memory and requires $s_i$ time units of processing, and no memory is freed by partially processing an operator. To convert a solution to the scheduling problem into a solution for the knapsack problem, add the $i$th knapsack item to $S$ if and only if the scheduling solution chooses to schedule the $i$th progress chart at least $s_i$ time units out of the first $C$ time units. The benefit gained by the knapsack solution is the same as the memory freed by the scheduling solution during the first $C$ time units, so an optimal schedule yields an optimal knapsack solution. □

Although the above proof uses a reduction from knapsack to the scheduling problem with multiple progress charts, the reduction can be adapted to produce a single progress chart, meaning that the scheduling problem is NP-complete even for the special case of a single query with $n$ operators.

### *4.2 Comparison with other operator-scheduling 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 for a fixed time unit or until 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 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 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 will 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 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, noncommutativity 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. We address this problem in Sect. 5.

*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 [37,43]. 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 [10]. 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 Sect. 6 validate the intuitive statements and analytical results presented in this section.

### *4.3 Multistream 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, which was the case in the previous sections.

Recall that we assume tuple-based sliding-window joins. The semantics of these joins is defined based on the timestamps of tuples in the joining streams. (We refer the reader to [2] for a comprehensive description of the semantics and implementation issues of 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 $r \in R$ with timestamp $t_r$ is processed, $r$ will be joined with the $w_S$ tuples with highest timestamps $\leq t_r$ in $S$ (where $w_S$ is the size of the sliding window on stream $S$). Symmetric processing occurs for $S$ tuples. When tuples with timestamps $t_1$ and $t_2$ are joined, the timestamp of the result tuple is $max(t_1, t_2)$. The result tuples have to be timestamped because the output of a join operator may be input to another join operator in a tree of join operators implementing a multiway join.

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 for input to upstream operators in a tree of joins, 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 $t$. Since we cannot predict the timestamps of future tuples for general streams, a sliding-window join operator will be blocked 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 multistream 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. An example query is shown in Fig. 3. This query contains a sliding-window join ($\bowtie$) between two streams $R$ and $S$, followed by a selection condition ($\sigma$) on the output of the join. Additionally, there is a project operator ($\pi$) 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 be blocked if any one of its input queues is empty.

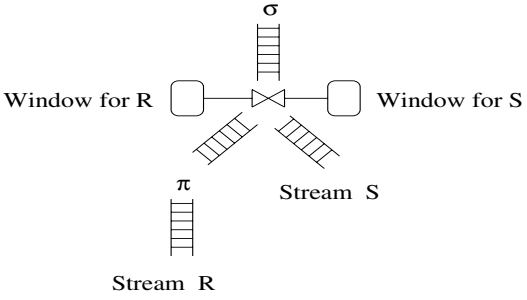


**Fig. 3.** Example sliding-window join query

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 [26]. Let the average number of tuples in a stream $S$ per unit time (as per the timestamp on tuples) be $\lambda_S$. Like average tuple selectivity, $\lambda_S$ 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 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 semijoin 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 Sect. 3. Our basic Chain strategy remains unchanged. The only difference is the following: earlier an operator could be blocked (could not be scheduled) only when an input queue was 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 multistream 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 Incorporating latency constraints

Our development of the Chain algorithm has focused solely on minimizing the maximum runtime memory usage, ignoring the important aspect of output latency. During bursts in input streams, Chain suffers from tuple starvation, i.e., Chain prefers to operate on new tuples that lie on steeper slope segments of the lower envelope, neglecting older tuples in the system that lie on segments with gentler slopes, thereby incurring a high latency for these old tuples. While minimizing runtime memory usage is important, many stream applications also require responses to stream inputs within specified time intervals. Although Chain as described so far cannot be used in such scenarios, we now turn to the task of modifying Chain to handle the latency issue. Our approach is to design a scheduling strategy that minimizes peak memory usage, subject to the constraint that output latency should not exceed some user-specified threshold. As we show in this section, simple extensions to Chain provide efficient solutions to this problem.

Throughout this discussion we assume that there exists a feasible scheduling strategy that adheres to the latency constraint. This imposes the restriction on the input pattern that it cannot have very high-volume bursts. If this restriction is not satisfied, then it is impossible to meet the latency constraint for all tuples. In such cases, one must either relax the latency constraint or drop some of the input tuples, i.e., perform *load shedding* [7,47]. Load shedding is an orthogonal issue and is not discussed in this paper.

Let $L$ denote the maximum allowable latency, as specified by the system administrator or as part of each user query. This parameter imposes the following latency constraint: Every input tuple with processing time $T$ that arrives at time $t$ should be output no later than $t + L$. If we let $T$ denote the total processing time for a single block of tuples, i.e., the $x$-coordinate of the last operator point on the progress chart, then the latency bound can be expressed as a multiple $f$ of the processing time, i.e., $f = L/T$. Typical values for $f$ might be in the thousands, based on an $L$ value on the order of a few seconds and a $T$ value of a few milliseconds.

It is well known that if there exists any schedule that can process all tuples before their deadlines, then "earliest deadline first" (EDF) scheduling will also process all tuples before their deadlines [28]. Moreover, EDF is an online strategy that does not need to know the input sequence a priori. When there is a single progress chart (i.e., a single query), EDF degenerates into the FIFO scheduling strategy.

Note that EDF is a "tuple-based" scheduling strategy, i.e., it makes scheduling decisions based on the runtime state of all the tuples, which includes knowledge about their individual deadlines and their progress so far. Such tuple-based strategies have a higher overhead in scheduling as opposed to "operator-based" scheduling strategies like Chain, which have a static operator priority. Some of the scheduling strategies we consider will be tuple based, despite the potentially higher implementation overheads associated with them. Techniques for improving the efficiency of certain tuple-based strategies are covered in Sect. 5.2. We begin with a few preliminary observations, followed by some negative results that show the difficulty of achieving near-optimal memory usage in the presence of latency constraints. For ease of exposition, we will restrict ourselves to the scenario in which the query plan consists of a single operator path (i.e., progress chart), although the negative results clearly extend to the case when there are many of them.

As mentioned earlier, if there exists a feasible scheduling strategy, the input must adhere to certain restrictions. In particular, it is easy to see that, for all $p$, the number of tuples input during a consecutive period of $pT$ time units cannot exceed $\lfloor f + p \rfloor$. Moreover, at any time all tuples present in the system were input within the last $fT$ time units. Putting together the two observations, at any point in time there are at most $2f - 1$ tuples that are not completely processed.

We next present a definition of *c-efficiency*, which measures the additive amount by which an algorithm diverges from optimal.

**Definition 5.1** c-Efficient algorithm: An online algorithm $A$ is said to be $c$-efficient for the operator scheduling problem if it requires no more than $x + c$ units of memory, where $x$ is the memory requirement of the best offline algorithm.

The following theorem shows that no online scheduling algorithm is better than $\Omega(f)$-efficient. In other words, no c-efficient online algorithm exists with a $c$ value that is sublinear in $f$. For progress charts where all operator selectivities are at most 1 (i.e., there are no non-foreign-key joins), this lower bound is asymptotically tight since the memory requirement of any feasible algorithm is bounded by $2f - 1$, the maximum number of unprocessed tuples.

The additive memory deviation increases linearly with the latency threshold $f = L/T$ because we only consider those input patterns for which there exists a feasible scheduling strategy. A larger threshold allows input patterns that can be more difficult for an online algorithm as compared to an offline scheduler. However, for a given pattern of input arrivals, memory usage tends to go down if the latency threshold is increased.

**Theorem 5.2** *There exist progress charts and arrival patterns for which no online algorithm can be* $(\frac{f}{12} - \delta)$*-efficient, for any positive number* $\delta$.

*Proof.* The proof is by constructions: we specify a particular progress chart and an adversarial sequence of tuple arrivals. For ease of exposition, in this proof all times are normalized by dividing by the total per-tuple processing time $T$. Consider a progress chart whose lower envelope is $(0, 1) - (t, 0.5 + \epsilon) - (3t, 2\epsilon) - (1, 0)$, where $t < \frac{1}{3f}$ and $\epsilon < \frac{6\delta}{f}$ are small

positive numbers. Consider the following arrival pattern in which tuples are inserted into the system: starting at time 0, $f$ tuples are inserted into the system at uniform time intervals of width $3t$ each. Note that all $f$ tuples have been inserted by time 1 since $t < \frac{1}{3f}$. The $(f+1)$th tuple is inserted at time $1+s$, where $s = ft$ is a slack value. Since each tuple must complete its execution within $f$ time units of its arrival, the $i$th tuple must complete execution by time $3(i-1)t + f$ for $0 < i \leq f$. The $(f+1)$st tuple must be completely processed by time $(f+1+s)$. Since $(f+1)$ tuples take $(f+1)$ time units to process, this leaves a slack of $s$ time units, which could go toward processing other tuples besides the $f+1$ specified earlier. (Note that there exists an algorithm whose maximum memory requirement is $1+2f\epsilon$ for the input specified so far.)

At time $f+1-3tf$, the adversary begins injecting a second batch of tuples starting at uniform intervals of $3t$. The number of tuples inserted in the second batch will be decided by an adversary at runtime; there will be either $f/3$ tuples or $f$ tuples in the batch.

Note that the slack $s = ft$ allows a combined processing of $ft$ time units on tuples from the second batch before the first $f+1$ tuples need to be processed to completion. In order to reduce memory usage, it would be helpful to process as many of the newly arriving tuples as possible for at least $3t$ time units each. If only $f/3$ tuples are going to be injected in the second batch, we could afford to process each of them for $3t$ units of time. However, since the available slack is only $ft$ time units, if $f$ tuples arrive in the second batch, it would be preferable to process each tuple in the second batch for only $t$ units, so that the overall reduction in memory usage is as large as possible, before it becomes necessary to process the $(f+1)^{st}$ tuple to completion in order to meet its deadline. Since it does not know the future, any online algorithm cannot at this juncture make the optimal choice of how many tuples from the second batch to process for $t$ time units, and how many to process for $3t$ units.

Consider an intermediate point where $\alpha = f/3$ tuples of the second batch have arrived in the system. There are two choices the scheduler could have taken for each tuple: either (1) process this tuple for $3t$ units so that its memory requirement is brought down to nearly zero or (2) process this tuple for only $t$ units so that its memory requirement is reduced to 0.5, and then spend $2t$ time units processing one of the first $f+1$ tuples that arrived during the first batch. (Without loss of generality we can assume that the online algorithm will process for at least $t$ time units each of the tuples that have arrived until now, since the first $t$ time units is the steepest segment of the progress chart). Choice (2) achieves a lesser reduction in memory as compared to choice (1), but the advantage of choice (2) is that less slack is consumed, so a greater number of tuples that may arrive in the future can be processed for at least $t$ time units. Note that the first $t$ time units of processing per tuple are most important, as spending time on the first segment of the progress chart decreases the memory usage faster than the second segment. At this point, after $\alpha$ tuples in the second batch have arrived, let us assume that the online algorithm processed $k$ of the tuples for time $3t$ and $(\alpha - k)$ of the tuples for time $t$.

The amount of memory consumed so far by the online algorithm during the second batch of tuples is $(0.5-\epsilon)(\alpha+k)$. The slack remaining is now $s - 3kt - (\alpha - k)t$, which is $(f - 2k - \alpha)t$, since $s = ft$. Consider the following two scenarios:

1. In Scenario 1, $(f-\alpha)$ more tuples enter the system in this batch at intervals of $3t$ than before. The online algorithm only has $(f-2k-\alpha)t$ slack remaining, hence it will only be able to consume at most $(0.5-\epsilon)(f-2k-\alpha)$ memory now. Thus the total memory consumed by the online algorithm during the second batch of tuples is $(0.5-\epsilon)(f-k)$. The optimal offline algorithm for this scenario would have used its $ft$ slack by processing each of the $f$ tuples in this batch for time $t$ when it arrives, consuming a total of $(0.5-\epsilon)f$ memory in the second batch. Hence in this scenario, the online algorithm fails to be $(0.5-\epsilon)k$-efficient.
2. In Scenario 2, no more tuples enter the system after this batch of $\alpha$ tuples. The optimal offline algorithm for this case would have processed each of the $\alpha$ tuples from the second batch for $3t$ time units when they arrived, achieving a memory reduction of $2(0.5-\epsilon)\alpha$, as opposed to the online algorithm that only reduced memory usage by $(0.5-\epsilon)(\alpha+k)$. Hence in this scenario, the online algorithm fails to be $(0.5-\epsilon)(\alpha-k)$-efficient.

The online algorithm could select $k$ so as to minimize its loss against the adversarial input pattern of tuples. The loss is given by the maximum of the two scenarios, which is selected by the adversary and is $(0.5-\epsilon)\max(k, \alpha-k)$. The online algorithm can minimize this by choosing $k = 0.5\alpha = f/6$. Since $\epsilon < \frac{6\delta}{f}$, we obtain that no online algorithm can be $(f/12-\delta)$-efficient. □

We can improve the constant in the above argument with a more complicated argument, but asymptotically it does not change the result.

The above result holds for general progress charts. However, for specific types of charts, for example those progress charts that have only two segments, it is possible to get a near-optimal online algorithm, as illustrated in the next subsection. (Note that for the lower bound in Theorem 5.2, we used a progress chart with three segments.)

### *5.1 Chain-Flush algorithm*

*Chain-Flush* is a simple modification of Chain for dealing with latency constraints. The Chain-Flush algorithm proceeds exactly like Chain until deadline constraints become tight, forcing a change in strategy. Suppose there are $n$ unprocessed (or partially processed) tuples in the system. Let $r_1 \leq r_2 \leq .... \leq r_n$ be the times remaining until the deadlines for these $n$ tuples, and let $t_1, t_2, ....t_n$ be the amounts of further processing time required for each tuple. (Note that the tuples are ordered by arrival order so that $t_1$ is the oldest unfinished tuple.) When $\Sigma_{j=0}^{i} t_j = r_i$, for some $0 < i \leq n$, and $\Sigma_{j=0}^{i'} t_j < r_{i'}$, for all $0 < i' < i$, then the $i$th tuple is on the verge of missing its deadline. At this point, no more processing on later-arriving tuples can be performed until the $i$th tuple and all earlier tuples have been completely processed. At this point, Chain-Flush is recursively applied, but now processing is restricted to only tuples $1, 2, \ldots, i$ until these $i$ tuples have been completely processed. At that point, Chain-Flush

resumes processing on all remaining unprocessed tuples, including any new tuples that may have arrived in the interim.

**Theorem 5.3** *For a progress chart consisting of operators with selectivity at most 1 and having a lower envelope composed of at most two segments, Chain-Flush is 1-efficient.*

*Proof.* The total processing time $t$ spent on all tuples up until the present moment can be divided into $t_1$, the time spent by Chain-Flush on the first, steeper segment of the progress chart, and $t_2$, the time spent on the second segment, so that $t = t_1 + t_2$. If $s_1$ and $s_2$ are the slopes of the two segments, with $s_1 \geq s_2$, the total memory consumed is $t_1 s_1 + t_2 s_2$. Suppose some other algorithm divides its time differently, spending $t_1{}'$ on the first segment and $t_2{}' = t - t_1{}'$ on the second segment. Since Chain-Flush greedily prefers to execute operators from the first segment whenever possible, only scheduling the second segment the minimum number of times necessary to meet the latency constraints, $t_1 \geq t_1{}'$ and $t_2 \leq t_2{}'$. We can show that, based on the lower envelope of the progress chart, the memory consumed by the alternate algorithm is less than the memory consumed by Chain-Flush: $t_1^{'} s_1 + t_2^{'} s_2 \leq t_1^{'} s_1 + (t_1 - t_1^{'}) s_1 - (t_1 - t_1^{'}) s_2 + t_2^{'} s_2 = t_1^{'} s_1 + (t_1 - t_1^{'}) s_1 - (t_2^{'} - t_2) s_2 + t_2^{'} s_2 = t_1 s_1 + t_2 s_2$. The memory requirement at any instant is the difference in the total memory of tuples entering the system less the memory consumed up to that point. If memory consumption is measured according to progress along the lower envelope, then as Chain-Flush has the maximum memory consumption at every instant, it has the minimum memory requirement. However, since the memory consumption indicated by the lower envelope only approximates the actual progress chart, the actual memory usage of Chain-Flush may be somewhat greater. The same argument used in the proof of Theorem 4.3 can be applied here to demonstrate that the total amount of additional memory used does not exceed 1. □

### *5.2 Implementing Chain-Flush efficiently*

Unlike the basic Chain algorithm, Chain-Flush is tuple based and involves considerable scheduling overhead. For every tuple $t$ queued in the system, we need to keep track of the total processing time remaining across all tuples that arrived before $t$ that are still queued in the system. This overhead makes a direct implementation of Chain-Flush unsuitable for data stream systems where many continuous queries can run concurrently. Of course, the overhead can be reduced by tracking the remaining processing time for blocks of tuples as opposed to individual tuples, at the potential cost of missing some deadlines. Nevertheless, the fact remains that the scheduling overhead of Chain-Flush as described in Sect. 5.1 is proportional to the total amount of data queued in the system. Fortunately, many stream applications that we are aware of [44] have *soft latency constraints* [27], so missing some deadlines by small margins is an acceptable tradeoff in these applications to improve overall throughput or resource utilization. In this section we describe an efficient, but approximate, implementation of Chain-Flush that leverages this property.

In the following discussion we assume that deadlines are specified at the level of a query as the maximum latency that can be tolerated for the output tuples of the query, denoted the *latency threshold* for the query. For the purposes of this discussion we make the following assumptions:

1. The timestamp of an input stream tuple is the wall-clock time at which the tuple arrived at the system.
2. We consider plans consisting of selection and join operators only. The timestamp of a tuple passed by a selection operator remains unchanged. The timestamp of a joined tuple produced by a join operator is the higher of the timestamps of the joining tuples.
3. The latency of a tuple output by the system is the difference between the timestamp of the tuple and the wall-clock time at which the tuple was output.

We maintain a data structure, denoted $\mathcal{Q}$, where for each queue $q$ in the system we store a four-tuple of the form $\langle q, t_h, r_q, p_q \rangle$, where $t_h$ is the timestamp of the head tuple of the queue, $r_q$ is the latency threshold for the query corresponding to the query plan containing $q$, and $p_q$ is the sum of the average tuple-processing times for all operators starting from the operator that reads from $q$ to the output operator of the plan containing $q$. Consider the head tuple of $q$ and let $\tau$ denote the current time. By definition, assuming the head tuple produces an output tuple eventually, $\tau + p_q$ is the earliest time at which we can produce this output tuple. Furthermore, the deadline for this output tuple is at $t_h + r_q$. $\mathcal{Q}$ is maintained incrementally in nondecreasing order of $t_h + r_q - p_q$ as operators execute.

Recall from Sect. 4.2 that Chain needs to make a scheduling decision only when an operator finishes processing all tuples in its input queue, or when a new block of tuples arrives in an input stream. However, Chain-Flush needs to make scheduling decisions more frequently to avoid an operator with many input tuples from taking much more than its share of the processor and thereby causing tuples in other parts of the query plan(s) to be delayed indefinitely. To ensure that scheduling decisions can be made in a timely manner without increasing scheduling overhead substantially, each operator on being scheduled is given a maximum number of tuples, e.g., 1000, that it can process before it returns control back to the scheduler.

At each scheduling step the scheduler checks whether the first entry $\langle q, t_h, r_q, p_q \rangle$ of $\mathcal{Q}$ satisfies $\tau + p_q \geq t_h + r_q$, where $\tau$ is the current time. If not, it schedules the operator that Chain would have scheduled at that point, i.e., the operator with the highest priority. If so, the scheduler switches to the flush mode of processing to process the head tuple of $q$ to completion. Effectively, it creates a virtual segment consisting of all operators starting from the reader operator of $q$ to the output operator of the plan containing $q$. The operators in this segment are scheduled in succession so that the head tuple of $q$ and all tuples in intermediate queues between operators in this segment are processed to completion.

Our implementation of Chain-Flush does not provide hard guarantees about meeting latency constraints. However, as our experiments in Sect. 6.2 show, in practice our implementation is able to keep output latency extremely close to the specified latency threshold. Furthermore, the scheduling overhead of our Chain-Flush implementation is comparable to that of Chain and is significantly lower than that of a direct implementation of Chain-Flush based on Sect. 5.1. Our Chain-Flush

implementation invokes the scheduler more often than does Chain, but the extra work per scheduling step simply involves accessing the first entry in $\mathcal{Q}$ and comparing it with the current time.

### *5.3 Mixed algorithm*

*Mixed* is another simple modification of Chain to deal with latencies caused by multiple segments of low slope toward the end of the progress chart. Mixed($\gamma$) clips those segments in the lower envelope that have a slope below a threshold $\gamma$ so that FIFO is applied to those segments. In other words, Mixed($\gamma$) modifies the lower envelope to combine the segments with slope less than $\gamma$ to form a unified segment.

*Example 2* Consider a simple progress chart whose lower envelope is given by the points (0,1), (1,0.1), (99,0.001), (100,0). Although the second segment has a steeper slope than the third, Mixed(0.01) combines both of them to form a single segment in the modified lower envelope. Mixed now applies Chain on this modified lower envelope.

We compare FIFO, Chain, and Mixed with respect to the maximum memory requirement and the latency of the output produced. Note that the latency of a tuple is the difference in time between its being completely processed and its being available for processing. The average latency is the latency averaged over all output tuples. Consider the following two arrival patterns.

1. One hundred tuples arrive, with the $i$th tuple arriving at time $99 \times i$. For this arrival pattern the Chain algorithm always selects the arriving tuple for processing the next one immediately.

| Metrics | Chain | FIFO | Mixed |
|---|---|---|---|
| Max. memory | 1.099 | 2 | 2 |
| Avg. latency | 5000 | 150 | 150 |
| Max. latency | 9901 | 199 | 199 |

2. Now consider the arrival pattern where ten tuples arrive, with the $i$th tuple arriving at time $10 \times i$.

| Metrics | Chain | FIFO | Mixed |
|---|---|---|---|
| Max. memory | 1.9 | 9 | 1.9 |
| Avg. latency | 595 | 550 | 595 |
| Max. latency | 910 | 910 | 910 |

The first pattern shows how tuples arriving at rates just above the processing rates can cause large latencies in Chain. The second pattern shows how FIFO, being nonadaptive, can have large memory requirements. The Mixed algorithm offers a way to avoid the worst excesses of both Chain and FIFO.

Mixed can be viewed as a heuristic strategy that is intermediate between Chain and FIFO. The slope threshold $\gamma$ serves as a tuning parameter that modulates the behavior of Mixed: higher values of $\gamma$ cause Mixed to behave more like FIFO, leading to increased memory usage but improved latency, whereas lower values of $\gamma$ make Mixed behave more like Chain, leading to lower memory usage at the cost of higher latency during bursts.

## 6 Experiments

In this section we describe the results of various experiments that we conducted to compare the performance of the various operator-scheduling policies described in this paper. We begin with a brief description of our simulation framework. In Sect. 6.1, we compare the performance of the scheduling policies in terms of the total memory requirement of all queues in the system, and in Sect. 6.2 we compare the performance in terms of the output latency.

Our implementation of 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 unit. The size of the time unit does affect the performance of Round-Robin, but it does not change the nature of the results presented here.

The notion of the 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.

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

1. **Synthetic data set:** The networking community has conducted 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 et al. [53]. Based on their "ON/OFF" model we generate synthetic bursty traffic by flows that begin according to a Poisson process (with mean interarrival time equal to 1 time unit) and then send packets continuously for some duration chosen from a heavy-tailed distribution. We use 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 use $k = 1$ 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 [22] is a good source of real-world stream data sets. One of their traces, named "DEC-PKT", contains 1 hour's worth of all wide-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.

### *6.1 Memory requirement of queues*

We now present the experimental evaluation of the total memory requirement of all queues in the system for the Chain, FIFO, Greedy, and Round-Robin policies. Section 6.1.1 presents the experimental results for single-stream queries, Sect. 6.1.2 considers queries having joins with stored relations, Sect. 6.1.3 considers sliding-window join queries, and Sect. 6.1.4 considers multiple queries. Chain-Flush and Mixed are evaluated in Sect. 6.2 and are not considered in this section. We ignore the cost of context switching in our experiments in this section. As discussed in Sect. 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.

#### 6.1.1 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 Sect. 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 1.

Figures 4 and 5 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 increases, the performance of Round-Robin degrades. FIFO performs badly with respect to Chain since it underutilizes 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 is hidden behind the second operator that has 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 Figs. 6 and 7, 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 Figs. 6 and 7 respectively, both of which perform nearly as badly 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 schedules the very selective and fast third operator, although it is hidden behind an a less selective operator. Notice that, unlike the previous case, Round-Robin does badly compared to Chain in Fig. 6.

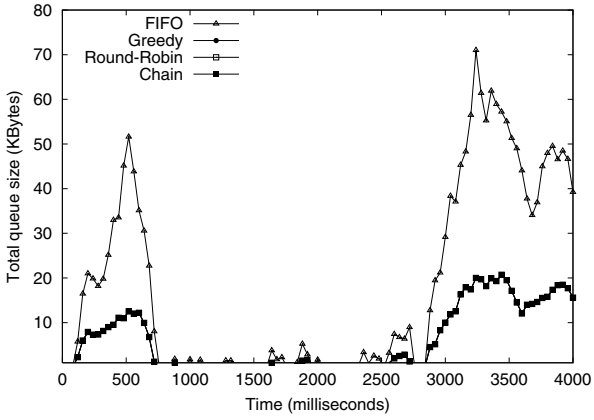


**Fig. 4.** Queue size vs. time (single stream, two operators, real data set)

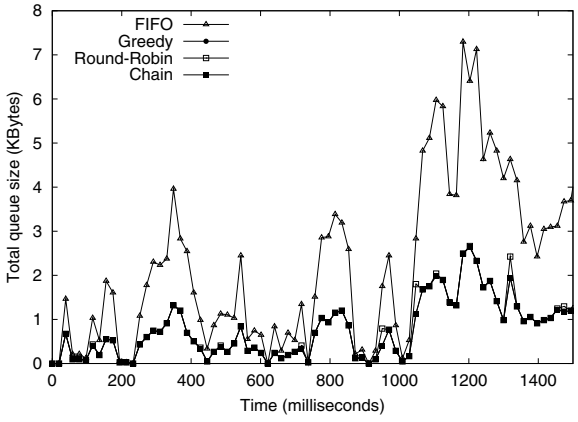


**Fig. 5.** Queue size vs. time (single stream, two operators, synthetic data set)

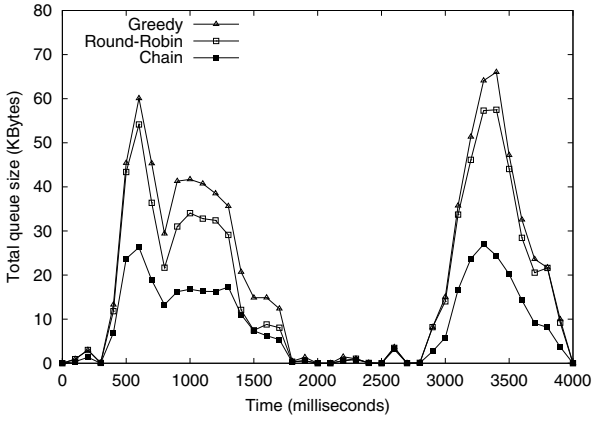


**Fig. 6.** Queue size vs. time (single stream, four operators, real data set)

#### 6.1.2 Queries having joins with stored relations

Recall from Sect. 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

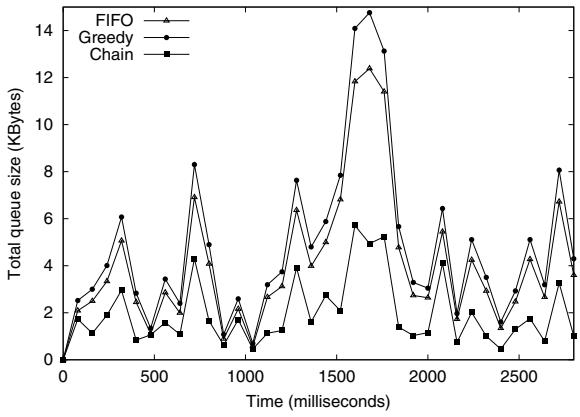


**Fig. 7.** Queue size vs. time (single stream, four operators, synthetic data set)

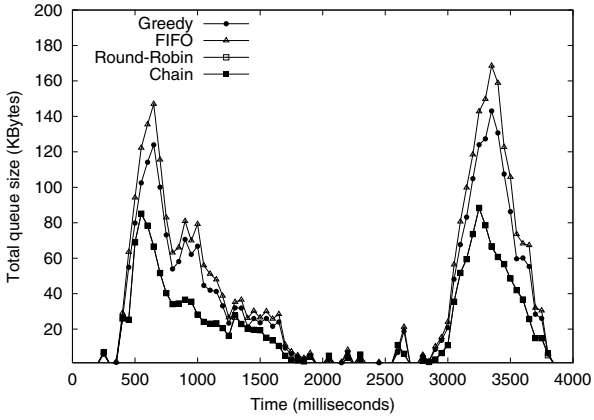


**Fig. 9.** Queue size vs. time (sliding-window join and three selections, real data set)

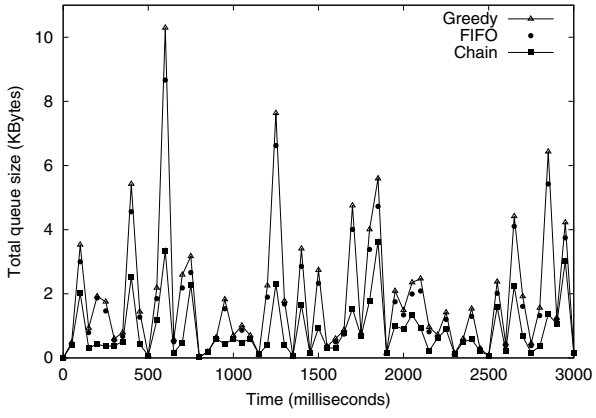


**Fig. 8.** Queue size vs. time (single stream, $s > 1$, synthetic data)

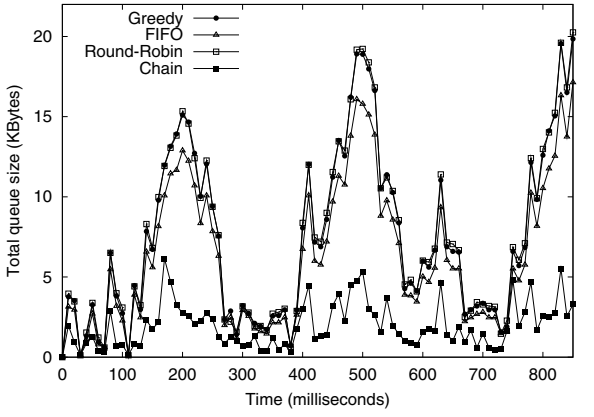


**Fig. 10.** Queue size vs. time (sliding-window join and three selections, synthetic data set)

$(4000, 0)$, where the second operator is a join with a stored relation. Figure 8 shows the performance of different scheduling strategies for the bursty synthetic data. (For legibility, we have not shown the performance of Round-Robin in Fig. 8, 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 (Sect. 6.1.1). Because of the low priority of the join operator, during bursts Greedy does not utilize the fast and selective operator that follows the join. On the other hand, the first three operators comprise a single segment in Chain, so the fast and selective third operator is used during bursts, leading to substantial benefits over Greedy (Fig. 8).

### 6.1.3 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 $\sigma_2$ and $\sigma_3$. Furthermore, before joining with $S$, stream $R$ passes through a selection condition $\sigma_1$. The selection conditions $\sigma_1$ and $\sigma_3$ are not very selective, while $\sigma_2$ (the selection following the join) is very selective. The performance graphs for this query over the real and synthetic data sets are shown in Figs. 9 and 10, 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 very badly on the synthetic data set. As in the experiment described in Sect. 6.1.2, 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 underutilizes the highly selective operator following it. FIFO performs badly for the reasons mentioned earlier, namely, the presence of a less selective operator ($\sigma_3$) with relatively high tuple processing time.

### 6.1.4 Multiple queries

Finally, we compared the performance of different strategies over a collection of three queries: a sliding-window join query similar to the one presented in the last experiment (Sect. 6.1.3) and two single-stream queries with selectivities less than 1 similar to those presented in Sect. 6.1.1. The performance graphs for this query workload over real and synthetic data sets are shown in Figs. 11 and 12, respectively. For improved legibility of Fig. 11, we have not connected the points correspond-

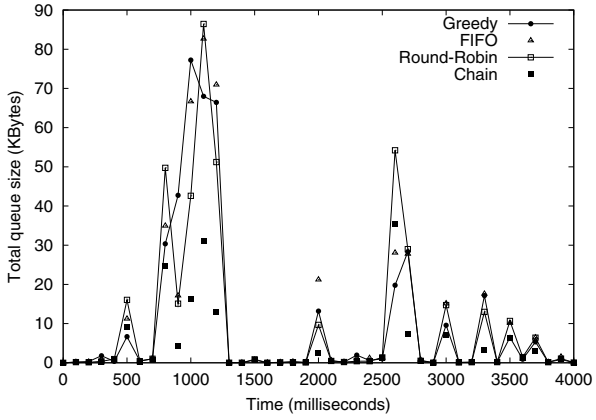


**Fig. 11.** Queue size vs. time (plan with multiple queries, real data set)

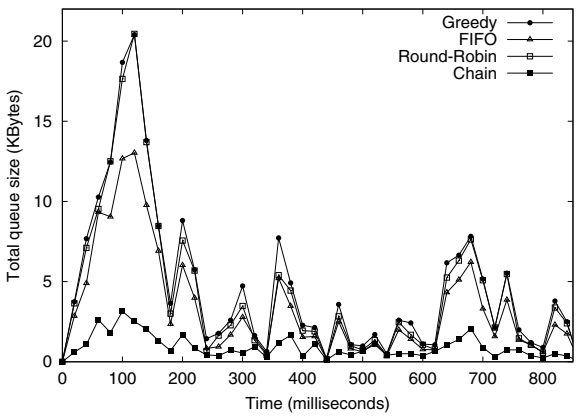


**Fig. 12.** Queue size vs. time (plan with multiple queries, synthetic data set)

ing to Chain and FIFO by line segments. The graphs show that the maximum memory requirement of Chain is much lower than that of other schedulers. The reason for the impressive performance of Chain is the increase in complexity and size of the underlying problem. 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 arrivals. On the other hand, since there is a larger number of operators in this multiquery 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 schedules the most selective operator decreases. This holds for FIFO as well. Greedy performs badly for reasons mentioned earlier: the queries consist of highly selective operators hidden behind 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 Chain become more pronounced. Indeed, the results in Figs. 11 and 12 were obtained by going from a single query to a collection of only three queries. In a real system with many more queries, the benefits would be even greater.

We have found the performance of Chain to be very robust to small deviations of selectivities from those assumed in the progress chart. Recall that in our experiments we choose specific progress charts first and then choose selection and join predicates with selectivities that closely reflect the progress charts on the data sets. In the experiments with real data, the average selectivity over the entire data set is used while choosing these predicates. In most cases, these selectivities show short-term deviations from the global average. As Figs. 4, 6, and 9 show, and as we have observed throughout in our experiments, the performance of Chain is fairly robust to such deviations.

### *6.2 Output latency*

The experiments in the previous section show how Chain keeps the total queue memory requirement much lower than that of FIFO, Greedy, or Round-Robin. We now evaluate the performance of all scheduling policies in terms of the latency of the output produced. Specifically, our experiments answer the following questions:

1. How does Chain compare with the other policies in terms of output latency?
2. How do Chain-Flush and Mixed compare to Chain and the other policies in terms of both the memory requirement and the output latency?

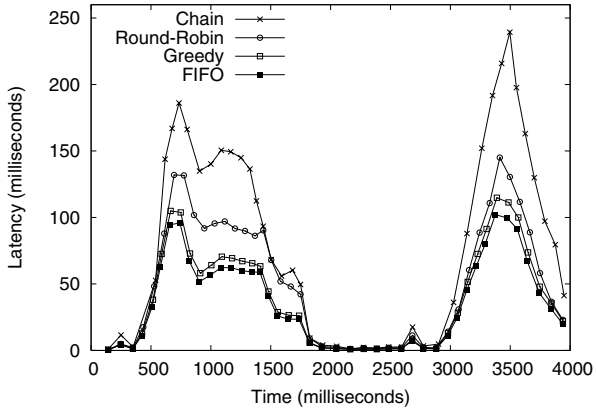


**Fig. 13.** Output latency vs. time (single stream, four operators, real data set)

Figures 13–16 show the output latency for the different scheduling policies. Figures 13 and 14 use the same real data set and progress chart as the four-operator query described in Sect. 6.1.1 and whose total queue size over time is shown in Fig. 6. Figures 15 and 16 use the same real data set and set of progress charts as the queries described in Sect. 6.1.4 and whose total queue size over time is shown in Fig. 11. To reduce clutter, we plot the average latency over consecutive blocks of 400 output tuples in all these graphs.

It is clear from Figs. 13 and 15 that Chain introduces considerable delays in producing tuples during periods of long bursts. For clarity, the plots for Greedy and Round-Robin are not shown in Fig. 15. Their performance is similar to that observed in Fig. 13. An interesting observation from Fig. 15 is that there are intervals of time when Chain outputs tuples with latency lower than that produced by FIFO. The reason for this behavior is as follows. In the presence of multiple queries,

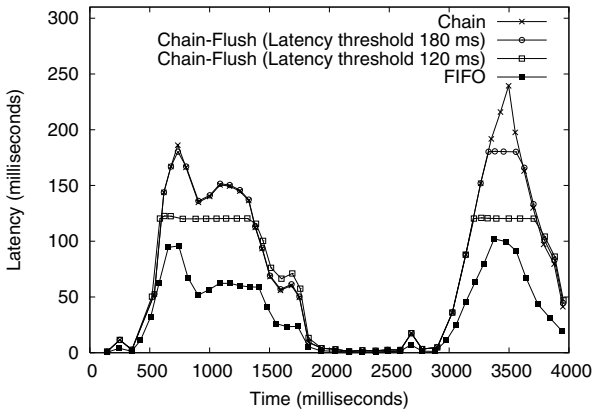


**Fig. 14.** Output latency vs. time (single stream, four operators, real data set)

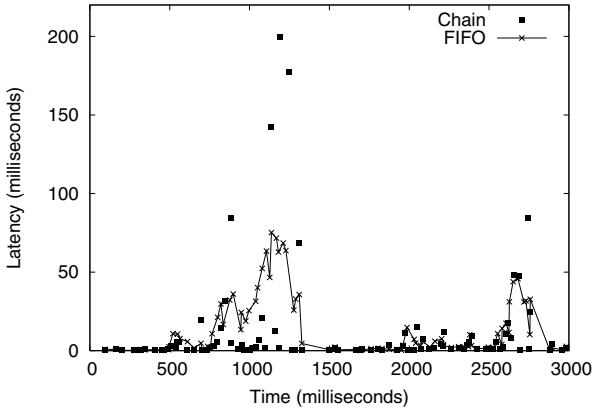


**Fig. 15.** Output latency vs. time (plan with multiple queries, real data set)

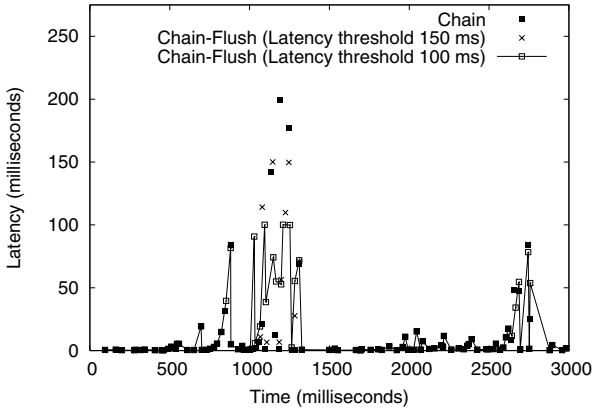


**Fig. 16.** Output latency vs. time (plan with multiple queries, real data set)

FIFO is restricted to processing blocks of tuples in global order of arrival across all input streams. Since Chain does not have a similar restriction, it might schedule query plans that produce output tuples faster temporarily. However, it is clear from Fig. 15 that the peak latency of Chain is much higher than that of FIFO.

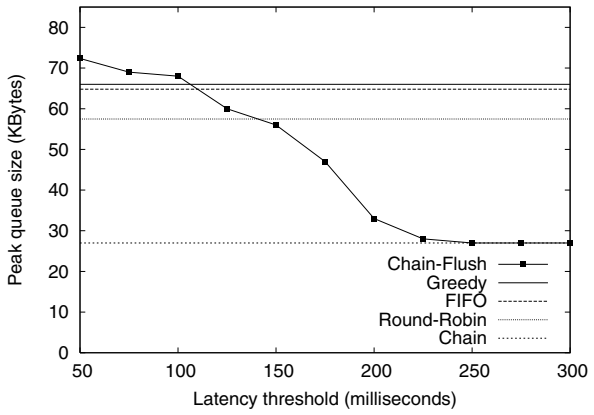


**Fig. 17.** Peak queue size vs. latency threshold (single stream, four operators, real data set)

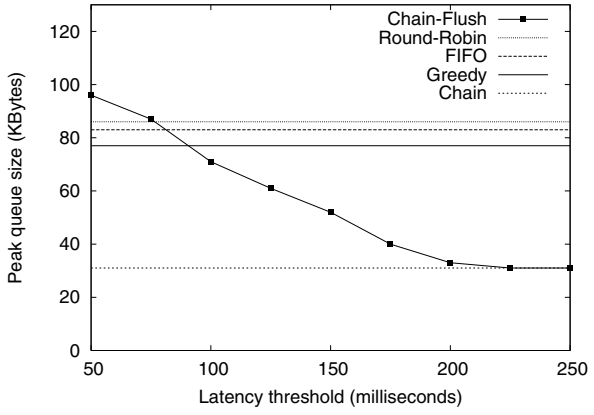


**Fig. 18.** Peak queue size vs. latency threshold (plan with multiple queries, real data set)

In Fig. 14 we show the output latency produced by Chain-Flush for two values of the latency threshold – 120 ms and 180 ms. Notice that in both cases Chain-Flush imitates Chain closely as long as Chain's output latency remains below the threshold, and Chain-Flush prevents the output latency from going above the threshold otherwise. A similar observation can be made from Fig. 16.

Figures 17 and 18 show the tradeoff between the peak queue memory requirement on the $y$-axis and latency threshold on the $x$-axis for the Chain-Flush algorithm. Figure 17 uses the same real data set and progress chart as the four-operator query in Sect. 6.1.1 and whose total queue size over time is shown in Fig. 6. As seen in Fig. 6, the peak queue memory requirement for this query happens during the burst around 3500 ms. Figure 18 uses the same real data set and set of progress charts as the queries in Sect. 6.1.4 and whose total queue size over time is shown in Fig. 11. As can be seen in Fig. 11, the peak queue memory requirement in this experiment happens during the burst around 1000 ms.

If the latency threshold is high enough, then Chain-Flush never needs to invoke the flush step and behaves identically to Chain. As the latency threshold is reduced, Chain-Flush starts to deviate from Chain and consequently has a higher peak queue memory requirement. When the latency threshold is set to the minimum value that can be sustained for these

workloads, which is the peak output latency of FIFO – around 100 ms for Fig. 17 (Fig. 13) and around 75 ms for Fig. 18 (Fig. 15) – Chain-Flush has almost the same queue memory requirement as FIFO. For lower values of the latency threshold, the memory requirement of Chain-Flush increases beyond that of FIFO because, for efficiency reasons, our implementation of Chain-Flush does not mirror FIFO while in the flush mode (Sect. 5.2).

Figures 19 and 20 compare the Mixed algorithm to the Chain and FIFO algorithms in terms of the runtime queue memory requirement and in terms of the output latency. For these experiments we used the real data set and a single stream query with four operators whose progress chart in terms of coordinates $(t_i, s_i)$ of operator points is: $(0, 1)$, $(1000, 0.3)$, $(1990, 0.2)$, $(3490, 0.1)$, and $(5490, 0)$. The four operators in this progress chart are in decreasing order of selectivity per unit time, so Chain puts each operator in a separate segment. Figures 19 and 20 show the runtime queue memory requirement and the output latency respectively for Chain, FIFO, and three invocations of Mixed – Mixed$(0.0001)$, Mixed$(0.0002)$, and Mixed$(0.0008)$ – with the slope threshold set to $0.0001$, $0.0002$, and $0.0008$, respectively. Mixed$(0.0001)$ ends up combining the third and fourth operators into a single segment and runs Chain on the three resulting segments. Similarly, Mixed$(0.0002)$ combines the second, third, and fourth operators into a single segment, and Mixed$(0.0008)$ combines all operators into a single segment. Figures 19 and 20 show that as the slope threshold for Mixed is increased, its behavior deviates more and more from Chain and becomes more and more like FIFO – note the performance of all policies during the burst around 3000 ms in Figs. 19 and 20.

Table 1 compares the performance of Chain-Flush and Mixed for different values of the respective latency and slope thresholds. The performance of Chain and FIFO are also shown for reference. We used the same data set and progress chart as in Figs. 19 and 20. The conclusions from Table 1 are similar to those from Figs. 17–20: The higher (lower) the latency (slope) threshold for Chain-Flush (Mixed), the more it deviates from FIFO and behaves like Chain. In reality, the latency threshold for Chain-Flush will be specified based on application requirements. However, the appropriate slope threshold for Mixed will have to be chosen by trial and error as part of query performance tuning. An interesting avenue for future work is to design adaptive algorithms for choosing the Mixed slope threshold based on the tradeoff between latency and memory observed at runtime.

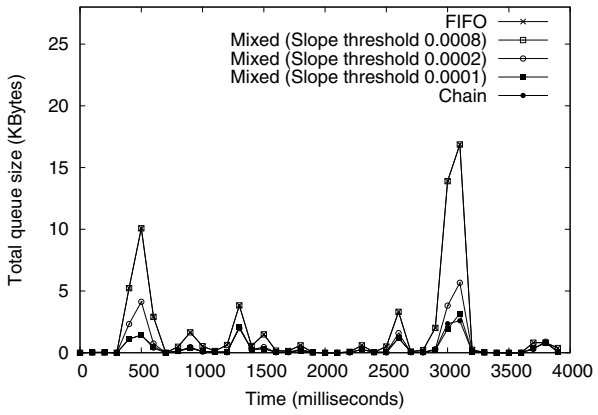


**Fig. 19.** Queue size vs. time (single stream, four operators, synthetic data set)

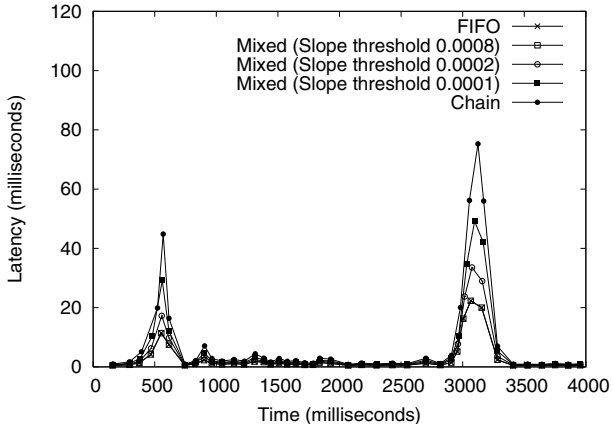


**Fig. 20.** Output latency vs. time (single stream, four operators, real data set)

## 7 Related work

This paper is an extended version of our paper that appeared in the proceedings of SIGMOD 2003 [5]. The basic Chain algorithm and its theoretical and experimental analysis were reported in the SIGMOD paper. The NP-completeness result showing the intractability of the problem of minimizing memory in Sect. 4 and the theoretical results and experiments for handling latency constraints in Sects. 5 and 6.2, respectively, are being presented for the first time in this paper.

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 Golab and

**Table 1.** Comparing Chain-Flush and Mixed

| Algorithm | Peak queue size (KB) | Peak latency (ms) |
|---|---|---|
| Chain | 2.5 | 76.2 |
| FIFO | 17.1 | 24.1 |
| Chain-Flush (latency threshold = 50 ms) | 9.2 | 50.9 |
| Chain-Flush (latency threshold = 10 ms) | 17.6 | 10.6 |
| Mixed (slope threshold = 0.0001) | 2.9 | 48 |
| Mixed( slope threshold = 0.0002) | 6.3 | 34.4 |
| Mixed (slope threshold = 0.0008) | 17.1 | 24.2 |

Ozsu [18]. 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 [33]). The novel *Eddies* architecture [3,12,34,42] enables very fine-grained adaptivity by eliminating query plans entirely, instead *routing* each tuple adaptively across the operators that need to process it. Unlike Eddies, which takes a holistic approach to adaptivity, we focus on adapting to changing *arrival characteristics* of data, in particular the bursty nature of data as documented in the networking community [17, 32,53,54]. Although we assumed fixed query plans in this paper, our scheduling algorithms are applicable to plan-based stream systems with coarse-grained adaptivity, e.g., STREAM [8]. In the future we plan to consider how our techniques can be applied to Eddy-like architectures with fine-grained adaptivity; see Sect. 8. Earlier work on adaptive query processing includes the query scrambling work by Urhan et al. [51], the adaptive query execution system Tukwila [23] for data integration, and mid-query reoptimization techniques developed by Kabra and DeWitt [25]. 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 [49], which is optimized to reduce initial and intermittent delay, and the work on dynamic pipeline scheduling for improving interactive performance of online queries [50]. However, in all these cases, the focus is exclusively on improving response times without considering runtime memory minimization.

Various operator-scheduling strategies have been suggested for stream systems, ranging from simple ones like Round-Robin scheduling [37] to more complex ones that aim at leveraging intra- and interoperator nonlinearities in processing [10,11]. To the best of our knowledge, ours is the first work to address the problem of scheduling with the aim of minimizing memory usage with and without latency constraints. Carney et al. [11] propose separate scheduling algorithms for memory minimization, latency minimization, and throughput maximization. Their algorithm for memory minimization is closely related to the Greedy algorithm described in Sect. 4.2.

Job scheduling has been a rich area of research in theoretical computer science and operations research. Excellent surveys of research in scheduling theory have been written by Karger et al. [28] and Lawler et al. [31]. Typically, this work aims to minimize a metric related to latency, such as the average or maximum job completion time or waiting time. While latency is an important secondary concern in our work, our primary objective of minimizing memory usage differentiates our work from the problems considered in the job-scheduling literature, where the notion of memory consumption is not applicable.

Our technique of combining adjacent operators that lie on the same lower envelope segment into operator chains is an adaptation of the technique introduced by Monma and Sidney [35] for scheduling jobs under series-parallel precedence constraints. The results in [35] require that the objective function satisfy a property known as the *adjacent sequence interchange* (ASI) property. The ASI property is not satisfied by our objective function (although our Lemma 4.2 can be interpreted as stating that the ASI property holds in an approximate sense), and thus the results from [35] do not directly apply to our problem. The scheduling technique of [35] has previously been applied to other database problems such as join order selection [21,30] and ordering of expensive predicates [13,20]. The ranking function we use to prioritize operators (i.e., the slope of the lower envelope segment) was also used in [13,20, 21,30], although the objective in these papers was to minimize execution time rather than to minimize memory requirements.

In this paper we assume that the average input stream arrival rate is such that it is eventually possible to clear backlogs of unprocessed tuples (e.g., when bursts of high arrival rate have receded). References [7,16,47] propose algorithms for the scenario when the stream system has to drop input tuples to bring the system load down to manageable levels. These algorithms are largely orthogonal to the algorithms we propose in this paper and could be used in conjunction with our algorithms. Algorithms to use disk efficiently in stream systems are proposed in [36]. Again, these algorithms are orthogonal to our algorithms.

Also related to our work is the set of papers [10,12,16,26, 34] championing the use of sliding-window joins for stream systems. The rate-based optimization framework of Viglas and Naughton [52] considers the problem of static query optimization with the modified aim of maximizing the throughput of queries for stream systems; however, it does not address runtime 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 [9,15, 38]. These techniques do not extend directly to data stream systems.

A recent paper by Ayad and Naughton [4] makes the observation that in some cases, the query plan that is most efficient (i.e., has the highest throughput) when stream rates are low can perform suboptimally when arrival rates become high. For this reason, [4] argues that the performance of a query plan in overloaded conditions should be one factor taken into consideration during query optimization. This suggests a possible additional usage for the operator-scheduling algorithms that we have presented: our scheduling policies could help the query optimizer estimate the memory usage of various query plans during bursts of various sizes, allowing the optimizer to use peak memory usage as an additional criterion (along with throughput) for determining the best query plan.[3]

## 8 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 near-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. We demonstrated that Chain can result in high-output latency in some scenarios and proposed two adaptations of the Chain algorithm, Chain-Flush and Mixed, that are designed to simultaneously achieve low memory usage and low latency.

[3] We are grateful to an anonymous reviewer for pointing out this possibility.

In this work, we made some simplifying assumptions about system architecture that might be violated in real systems. For example, the following capabilities violate our assumptions but may be desirable in real systems: the ability to generate adaptive query plans, where the structure of the query plan itself is allowed to change over time; sharing of computation and memory buffers across query plans; and multithreaded query processing for multiprocessor systems. A better understanding of how the introduction of such capabilities impacts operator scheduling for memory minimization is an interesting open problem.

## References


1. Amsaleg L, Franklin M, Tomasic A (1998) Dynamic query operator scheduling for wide-area remote access. J Distrib Parallel Databases 6(3):217–246
2. Arasu A, Babu S, Widom J (2002) An abstract semantics and concrete language for continuous queries over streams and relations. Technical report, Stanford University Database Group. http://dbpubs.stanford.edu/pub/2002-57
3. Avnur R, Hellerstein J (2000) Eddies: continuously adaptive query processing. In: Proc 2000 ACM SIGMOD international conference on management of data, pp 261–272
4. Ayad AM, Naughton JF (2004) Static optimization of conjunctive queries with sliding windows over infinite streams. In: Proc 2004 ACM SIGMOD international conference on management of data
5. Babcock B, Babu S, Datar M, Motwani R (2003) Chain: operator scheduling for memory minimization in data stream systems. In: Proc 2003 ACM SIGMOD international conference on management of data
6. Babcock B, Babu S, Datar M, Motwani R, Widom J (2002) Models and issues in data stream systems. In: Proc 2002 ACM symposium on principles of database systems
7. Babcock B, Datar M, Motwani R (2004) Load shedding for aggregation queries over data streams. In: Proc 2004 international conference on data engineering, pp 350–361
8. Babu S, Motwani R, Munagala K, Nishizawa I, Widom J (2004) Adaptive ordering of pipelined stream filters. In: Proc 2004 ACM SIGMOD international conference on management of data
9. Bouganim L, Kapitskaia O, Valduriez P (1998) Memory-adaptive scheduling for large query execution. In: Proc 1998 ACM CIKM international conference on information and knowledge management, pp 105–115
10. Carney D, Cetintemel U, Cherniack M, Convey C, Lee S, Seidman G, Stonebraker M, Tatbul N, Zdonik S (2002) Monitoring streams – a new class of data management applications. In: Proc 28th international conference on very large data bases
11. Carney D, Cetintemel U, Rasin A, Zdonik S, Cherniack M, Stonebraker M (2003) Operator scheduling in a data stream manager. In: Proc 2003 international conference on very large data bases
12. Chandrasekaran S, Franklin M (2002) Streaming queries over streaming data. In: Proc 28th international conference on very large data bases
13. Chaudhurim S, Shim K (1999) Optimization of queries with user-defined predicates. ACM Trans Database Sys 24(2):177–228
14. Cortes C, Fisher K, Pregibon D, Rogers A, Smith F (2000) Hancock: a language for extracting signatures from data streams. In: Proc 2000 ACM SIGKDD international conference on knowledge discovery and data mining, pp 9–17
15. Dageville B, Zait M (2002) SQL memory management in Oracle9i. In: Proc 2002 international conference on very large data bases
16. Das A, Gehrke J, Riedewald M (2003) Approximate join processing over data streams. In: Proc 2003 ACM SIGMOD international conference on management of data
17. Floyd S, Paxson V (1995) Wide-area traffic: the failure of poisson modeling. IEEE/ACM Trans Network 3(3):226–244
18. Golab L, Ozsu T (2003) Issues in data stream management. SIGMOD Record 32(2):5–14
19. Hellerstein J, Franklin M, Chandrasekaran S, Deshpande A, Hildrum K, Madden S, Raman V, Shah MA (2000) Adaptive query processing: technology in evolution. IEEE Data Eng Bull 23(2):7–18
20. Hellerstein J, Stonebraker M (1993) Predicate migration: optimizing queries with expensive predicates. In: Proc 1993 ACM SIGMOD international conference on management of data, pp 267–276
21. Ibaraki T, Kameda T (1984) On the optimal nesting order for computing n-relational joins. ACM Trans Database Sys 9(3):482–502
22. Internet Traffic Archive: http://www.acm.org/sigcomm/ITA/
23. Ives Z, Florescu D, Friedman M, Levy A, Weld D (1999) An adaptive query execution system for data integration. In: Proc 1999 ACM SIGMOD international conference on management of data, pp 299–310
24. Johnson T, Cranor C, Spatsheck O, Shkapenyuk V (2003) Gigascope: a stream database for network applications. In: Proc 2003 ACM SIGMOD international conference on management of data
25. Kabra N, DeWitt DJ (1998) Efficient mid-query re-optimization of sub-optimal query execution plans. In: Proc ACM SIGMOD international conference on management of data, pp 106–117
26. Kang J, Naughton JF, Viglas S (2003) Evaluating window joins over unbounded streams. In: Proc 2003 international conference on data engineering
27. Kao B, Garcia-Molina H (1995) An overview of real-time database systems. In: Son SH (ed) Advances in real-time systems. Prentice Hall, Englewood Cliffs, NJ, pp 463–486
28. Karger D, Stein C, Wein J (1997) Scheduling algorithms. In: Atallah MJ (ed) Handbook of algorithms and theory of computation. CRC, Boca Raton, FL
29. Kleinberg J (2002) Bursty and hierarchical structure in streams. In: Proc 2002 ACM SIGKDD international conference on knowledge discovery and data mining
30. Krishnamurthy R, Boral H, Zaniolo C (1986) Optimizing nonrecursive queries. In: Proc 1986 international conference on very large data bases, pp 128–137
31. Lawler EL, Lenstra JK, Rinnooy Kan AHG, Shmoys DB (1993) Sequencing and scheduling: algorithms and complexity. In: Graves SC, Zipkin PH, Rinnooy Kan AHG (eds) Logistics of production and inventory, Handbooks in operations research and management science, vol 4, North-Holland, Amsterdam, pp 445–522
32. Leland W, Taqqu M, Willinger W, Wilson D (1994) On the self-similar nature of ethernet traffic. IEEE/ACM Trans Network 2(1):1–15
33. Lomet D, Levy A (2000) Special issue on adaptive query processing. IEEE Data Eng Bull 23(2):1–48
34. Madden S, Shah M, Hellerstein J, Raman V (2002) Continuously adaptive continuous queries over streams. In: Proc 2002 ACM SIGMOD international conference on management of data

35. Monma C, Sidney J (1987) Optimal sequencing via modular decomposition: characterization of sequencing functions. Math Oper Res 12:22–31
36. Motwani R, Thomas D (2004) Caching queues in memory buffers. In: Proc 2004 annual ACM-SIAM symposium on discrete algorithms
37. Motwani R, Widom J, Arasu A, Babcock B, Babu S, Datar M, Manku G, Olston C, Rosenstein J, Varma R (2003) Query processing, approximation, and resource management in a data stream management system. In: Proc 1st biennial conference on innovative data systems research (CIDR)
38. Nag B, DeWitt DJ (1998) Memory allocation strategies for complex decision support queries. In: Proc 1998 ACM CIKM international conference on information and knowledge management, pp 116–123
39. Niagara Project. http://www.cs.wisc.edu/niagara/
40. Parker DS, Muntz RR, Chau HL (1989) The tangram stream query processing system. In: Proc 1989 international conference on data engineering, pp 556–563
41. Parker DS, Simon E, Valduriez P (1992) SVP: a model capturing sets, lists, streams, and parallelism. In: Proc 1992 international conference on very large data bases, pp 115–126
42. Raman V, Deshpande A, Hellerstein J (2003) Using state modules for adaptive query processing. In: Proc 2003 international conference on data engineering
43. Shah M, Madden S, Franklin M, Hellerstein J (2001) Java support for data-intensive systems: experiences building the telegraph dataflow system. SIGMOD Record 30(4):103–114
44. SQR – a stream query repository. http://www-db.stanford.edu/stream/sqr
45. Stanford Stream Data Management (STREAM) Project. http://www-db.stanford.edu/stream
46. Sullivan M (1996) Tribeca: a stream database manager for network traffic analysis. In: Proc 1996 international conference on very large data bases, p 594
47. Tatbul N, Cetintemel U, Zdonik S, Cherniack M, Stonebraker M (2003) Load shedding in a data stream manager. In: Proc 2003 international conference on very large data bases, pp 309–320
48. Terry D, Goldberg D, Nichols D, Oki B (1992) Continuous queries over append-only databases. In: Proc 1992 ACM SIGMOD international conference on management of data, pp 321–330
49. Urhan T, Franklin M (2000) Xjoin: a reactively-scheduled pipelined join operator. IEEE Data Eng Bull 23(2):27–33
50. Urhan T, Franklin MJ (2001) Dynamic pipeline scheduling for improving interactive performance of online queries. In: Proc 2001 international conference on very large data bases
51. Urhan T, Franklin MJ, Amsaleg L (1998) Cost-based query scrambling for initial delays. In: Proc 1998 ACM SIGMOD international conference on management of data, pp 130–141
52. Viglas S, Naughton J (2002) Rate-based query optimization for streaming information sources. In: Proc 2002 ACM SIGMOD international conference on management of data
53. Willinger W, Paxson V, Riedi R, Taqqu M (2002) Long-range dependence and data network traffic. In: Doukhan P, Oppenheim G, Taqqu MS (eds) Long-range dependence: theory and applications. Birkhäuser, Basel, Switzerland
54. Willinger W, Taqqu M, Erramilli A (1996) A bibliographical guide to self-similar traffic and performance modeling for modern high-speed networks. In: Kelly FP, Zachary S, Ziedins I (eds) Stochastic networks: theory and applications. Oxford University Press, Oxford, UK, pp 339–366
55. Wilschut AN, Apers PMG (1991) Dataflow query execution in a parallel main-memory environment. In: Proc 1991 international conference on parallel and distributed information systems, pp 68–77