A Cost Model for Adaptive Resource Management in Data Stream Systems

Michael Cammert, Jürgen Krämer, Bernhard Seeger, Sonny Vaupel

Department of Mathematics and Computer Science
University of Marburg
35032 Marburg, Germany
{cammert,kraemerj,seeger,sonny}@mathematik.uni-marburg.de

Abstract

Data stream management systems need to control their resources adaptively since stream characteristics as well as query workload vary over time. In this paper we investigate an approach to adaptive resource management for continuous sliding window queries that adjusts window sizes and time granularities to keep resource usage within bounds. In order to quantify the impact of both methods on a query plan, we develop an efficient cost model for estimating the resource allocation in terms of memory usage and processing costs. A thorough experimental study not only validates the accuracy of our cost model but also demonstrates the efficacy and scalability of the applied methods.

1 Introduction

Data stream management systems (DSMS) have emerged as a new technology to meet the challenging requirements for processing and querying data that continuously arrives in form of potentially infinite streams from autonomous sources. Standard database technology is not well suited for coping with data streams, and particularly, with the many long-running queries that have to be processed concurrently. Not only the long-running queries but also the fluctuating stream characteristics require a high degree of adaptivity in DSMS. This involves the dynamic re-optimization and (re-)distribution of system resources at runtime. A prerequisite for achieving these goals is an appropriate cost model which allows to evaluate the quality and efficacy of a (re-optimized) query plan. For that reason, we present an accurate cost model for continuous sliding window queries which enables a DSMS to estimate its resource usage in terms of memory and processing costs. Although this paper validates the cost model in the context of adaptive resource management, the model is general enough to serve as a foundation for cost-based query optimization as well.

In accordance with [8], our work assumes that resources spent on inter-operator queues are almost negligible for unsaturated systems as those are capable of keeping up with stream rates. Therefore, we focus on the operators inside a query plan and their resource allocation. With the aim to dynamically control the system resources allocated by a query plan, we proposed the following two techniques [11]:

1. **Window size adjustments** enable a DSMS to change the window sizes of queries within user-defined bounds at runtime. This technique affects all stateful operators, e. g., the join. A smaller window size implies a smaller state. Hence, less memory needs to be allocated. Furthermore, processing costs are generally reduced, for instance, the cost of probing in a nested-loops join.

2. **Time granularity adjustments** within user-defined bounds enable a DSMS to change the time granularity at which results, especially aggregates, are computed. For a lot of real-world queries, the user neither needs nor wants results at finest time granularity. A coarser time granularity shrinks the state of the aggregation which, in turn, saves memory and processing costs.

A DSMS can use those methods similarly to load shedding with the aim to avoid system saturation [24, 5]. Whenever resource utilization approaches the maximum system capacity, e. g., by installing new queries or increasing stream rates, the DSMS can scale down window sizes and scale up time granularities to remain in a steady, non-saturated state. The inverse action may be advisable if queries are removed from the system or stream rates decrease. Changes are restricted to user-defined Quality-of-Service (QoS) bounds, while an objective of a DSMS should be to maximize overall QoS.

In prior work [11], we defined the semantics of the methods mentioned above and explained their usage with regard to adaptive memory management. Thus, we published the framework together with its motivation. We presented the impact of those methods on a query plan neither formally nor experimentally. This paper overcomes those deficiencies while it seamlessly continues our previous work with novel insights. The main contributions are summarized as follows:

- We not only consider memory usage but also take processing costs into account.

- We develop a sound cost model for estimating the resource utilization of continuous sliding window queries. This model is more extensive than existing ones as it covers a variety of operators, e. g., aggregation, considers different time granularities, and shows reorganization costs for temporal expiration.

- We use this cost model to quantify the effects of changes to window sizes and time granularities on resource utilization. Furthermore, we analyze at which point in time these effects become fully observable.

- Our thorough experimental studies on synthetic and real-world data streams (i) show that adjustments to window sizes and time granularities are suitable for adaptive resource management, (ii) validate the accuracy of our cost model, (iii) illustrate the differences between window reduction and load shedding, and (iv) prove the scalability of both, the techniques and our cost model, based on a preliminary adaptive resource manager.

The rest of the paper is organized as follows. Section 2 briefly outlines important implementation aspects. Section 3 identifies stream characteristics and introduces cost formulas to estimate the resource usage of operators. Adaptive resource management on top of our cost

model is motivated in Section 4. Section 5 summarizes our experimental studies. Related work is discussed in Section 6. Finally, Section 7 concludes the paper.

2 Stream Query Processing

In order to understand the effects of the applied techniques [11] and the development of our cost model, it is necessary to gain insight into the semantics and implementation of continuous sliding window queries in PIPES [18, 10].

PIPES is an infrastructure providing all fundamental building blocks to implement a DSMS. Its core is an extensive operator algebra with a sound semantics [19]. In analogy to traditional database management systems, PIPES distinguishes between a logical and a physical operator algebra. Continuous queries formulated in CQL (Continuous Query Language) [3] can be expressed in PIPES due to its snapshot-reducible operators [21, 19].[1] The query optimizer in PIPES contains a parser that transforms CQL statements into a logical query plan.[2] Afterwards, the logical query plan is optimized by applying transformation rules and exploiting subquery sharing. In the next step, the optimized logical plan is mapped to the physical operator algebra which provides data-driven, i. e. push-based, stream-to-stream implementations.

2.1 Time and its Granularities

Let $\mathbb{T} = (T, \leq)$ be a discrete time domain with a total order $\leq$. We use $\mathbb{T}$ to model the notion of *application time*, not system time. For the sake of simplicity, let T be the non-negative integers $\{0, 1, 2, 3, \dots\}$.

According to [11], a *time granularity* G is a non-empty subdomain of the time domain $\mathbb{T}$, $G \subseteq T, 0 \in G$, with the same total order $\leq_{|G}$. We assume the time distance g between each pair of successive time instants in G to be equal. This condition holds for the conventional time granularities such as milliseconds, seconds, minutes, hours. A time granularity G' is coarser than G, if $G' \subset G$.

2.2 Semantics

The operations of our logical and physical operator are *snapshot-reducible* to their counterparts of the extended relational algebra. Figure 1 illustrates the temporal concept of snapshot reducibility [21].

DEFINITION 1 (SNAPSHOT REDUCIBILITY). *We denote a stream-to-stream operation op_T with inputs $S_1, \dots, S_n$ as snapshot-reducible if for each time instant $t \in T$, the snapshot at t of the results of op_T is equal to the results of applying its relational counterpart op to the snapshots of $S_1, \dots, S_n$ at time instant t.*

A snapshot of a stream at time t can be considered as a relation since it represents the multiset of all tuples valid at time instant t. Two query plans are denoted equivalent if they

[1] From the snapshot perspective, PIPES produces the same results as the *Positive-Negative Tuple Approach* [3]. However, it does not have the drawback of doubling stream rates due to sending positive and negative tuples.

[2] We extended CQL to the enable the user to specify QoS bounds for window sizes and time granularities [11].

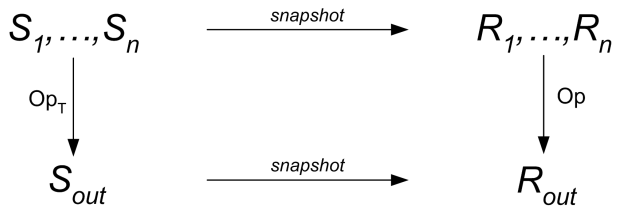

Figure 1: Snapshot reducibility

generate *snapshot-equivalent* results, i. e., if for each time instant t their snapshots at t are equal. Snapshot equivalence is the reason why the common relational transformation rules are still applicable to the stream algebra. Furthermore, it explains why the techniques in [11] produce exact query results with regard to the chosen QoS settings.

Figure 1 can also be used to motivate that our algebra is basically expressive enough to implement any continuous query defined in STREAM [4]. The abstract semantics proposed for STREAM transforms the input streams into relations, uses operations of the relational algebra for processing, and transforms the output relations back into streams. The window specifications are included in the mappings from streams to relations. Visually compared, our stream-to-stream approach works completely on the left side of Figure 1, whereas STREAM goes the way round on the right side.

2.3 Streams

The physical operator algebra works on so-called *physical streams*.

DEFINITION 2 (PHYSICAL STREAM). *A physical stream S is a potentially infinite, ordered sequence of elements $(e, [t_S, t_E))$ composed of a tuple e belonging to the schema of S and a half-open time interval $[t_S, t_E)$ where $t_S, t_E \in T$. A physical stream is non-decreasingly ordered by start timestamps.*

The interpretation of a physical stream element $(e, [t_S, t_E))$ is that a tuple e is valid during the time interval $[t_S, t_E)$.

Many stream applications provide a DSMS with a continuous sequence of elements equipped with a timestamp attribute, but no time interval. We denote such a stream as *source stream*. We assume that only a finite number of elements in a source stream have the same timestamp. Source streams are usually ordered by the timestamp attribute. A source stream can be converted into a physical stream by mapping each incoming element e with its internal timestamp t to $(e, [t, t + 1))$, where $+1$ indicates a time period at the finest time granularity.

2.4 Operators

We denote our stream operators derived from the temporal relational algebra [21] as *standard* operators. Examples for standard operators are selection, projection, union, join, aggregation, etc. Standard operators are *snapshot-reducible* to their counterparts of the extended relational algebra. However, a stream algebra consisting only of standard operators is not expressive enough

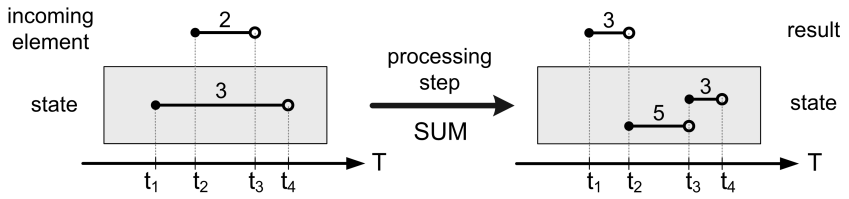

Figure 2: Aggregation example

for our purpose because it is not possible to express sliding windows and time granularities. For that reason, the operator algebra in PIPES contains further operators.

2.4.1 Standard Operators

A standard operator in the physical algebra processes physical streams as input and produces a physical stream as output. The following implementation sketches of the join and aggregation are aimed at getting a feeling for the implementation of snapshot-reducible operators.

The join operator ($\bowtie$) satisfies the following two conditions [19, 17]: (a) for two participating stream elements, the join predicate has to be fulfilled and (b) their time intervals have to intersect. The time interval associated with the join result is set to the intersection of the two participating time intervals.

Algorithm 1: Scalar Aggregation (for distributive aggregates)

```
1  foreach (e, [t_S, t_E)) ∈ input stream do
2      foreach t ∈ {t_S, t_E} do
3          foreach (a, [t'_S, t'_E)) ∈ state where t'_S < t < t'_E do
4              Replace (a, [t'_S, t'_E)) by (a, [t'_S, t)) and (a, [t, t'_E));
5      foreach (a, [t'_S, t'_E)) ∈ state where [t'_S, t'_E) ⊆ [t_S, t_E) do
6          Replace (a, [t'_S, t'_E)) by (agg(a, e), [t'_S, t'_E));
7      t̂_E ← max{t'_E | ∃(a, [t'_S, t'_E)) ∈ state};
8      if t_S ≥ t̂_E then
9          Add new aggregate (agg(⊥, e), [t_S, t_E)) to state;
10     else if t_E > t̂_E then
11         Add new aggregate (agg(⊥, e), [t̂_E, t_E)) to state;
12     foreach (a, [t'_S, t'_E)) ∈ state where t'_E ≤ t_S do
13         Remove (a, [t'_S, t'_E)) from state and append it to the output stream;
```

Algorithm 1 shows the implementation of the scalar aggregation operator (α) where a stands for an aggregation value and agg denotes an aggregation function. The aggregation function is applied iteratively, as illustrated by Figure 2 for the sum. An aggregate is initialized by calling the aggregation function with $\perp$ as first parameter. In the case of a finite input stream, all elements in the state are appended to the output stream after the last element was processed.

Thereafter, the state can be cleared. Note that this kind of aggregation is compatible with the user-defined aggregates presented in [20].

2.4.2 Window Operator

The window operator (ω) assigns a validity according to its window size to each element of its input stream. For a *time-based* sliding window, the window size $w \in T$ represents a period in application time. In general, the window operator produces for each incoming element $(e, [t_S, t_E))$ a set of elements by extending the validity of each single time instant by the window size w. That means for an element $(e, [t_S, t_S + 1))$ of a source stream, the window size has only to be added to the end timestamp, $(e, [t_S, t_S + 1 + w))$. This is intuitive for a time-based sliding window query since the stateful operation downstream of a window operator has to consider an element for additional w time instants. Note that a window operator does not affect stateless operations, such as selection and projection.

Tuple-based sliding windows [3] can be implemented by setting the end timestamp of an incoming element to the start timestamp of the c-th future incoming element, where c is the window size. Our cost model does not explicitly address tuple-based windows. In our cost model, a tuple-based window is treated as a time-based window of size $w = \frac{c}{\lambda}$, where λ denotes the average stream rate.

Adjusting the window size of a window operator means to change it to a new value $w' \in T$ at a certain application time instant $t \in T$.

2.4.3 Granularity Operator

A substream of a physical stream is said to have time granularity G if the starting and ending time instants of all time intervals assigned to its elements belong to G. Adjusting the time granularity means that it is changed to a new time granularity G' at a certain application time instant $t \in T$ by the granularity operator (Γ). From this time instant onwards, a new substream starts having granularity G'. This is achieved by rounding the start and end timestamps of incoming time intervals to $t'_S = \left\lceil \frac{t_S}{g'} \right\rceil g'$ and $t'_E = \left\lfloor \frac{t_E}{g'} \right\rfloor g'$, where g' is the fixed time distance of G'.

Under the assumption that the time granularity is only allowed to be changed from a coarser to a finer granularity at time instants belonging to the coarser granularity, rounding the interval starts and endings does not conflict with the temporal ordering of the output stream. This restriction limits the feasibility only marginally, but facilitates the implementation towards a mapping that rounds the start and end timestamps of each incoming element according to the new time granularity.

2.4.4 Temporal Expiration

Windowing constructs (i) restrict the resource usage and (ii) unblock otherwise blocking operators over infinite streams like aggregation. In stateful operators, e. g., join, elements in the state expire due to the validity assigned by the window operator. A stateful operator considers an element $(e, [t_S, t_E))$ in its state as expired if it is guaranteed that no element in one of its input

streams will arrive in future whose time interval will overlap with $[t_S, t_E)$. According to the total order claimed for streams, this condition holds if the minimum of all start timestamps of the latest incoming element from each input stream is larger than t_E. A stateful operator can delete all expired elements from its state. Some operators such as the aggregation emit these expired elements as results prior to their removal.

Heartbeats [22] can be used to explicitly trigger additional expiration steps in those cases where application-time skew between streams and latency in streams become an issue.

2.5 Continuous Queries

Query plan construction is basically the same as in conventional database systems, except that the query optimizer has to place window and granularity operators in addition. PIPES manages its continuous queries in a directed acyclic graph composed of operators and source streams. A window operator is placed downstream of the source for which the window has been specified in the corresponding CQL query. A granularity operator is typically placed upstream of the aggregation for which it has been specified [11]. This placement is performed by the query optimizer when transforming the CQL query into the logical query plan.

3 Cost Model

This section presents our cost model for estimating the resource utilization of continuous queries. It is structured as follows. First, the general purpose of our cost model is motivated. Then, we define our model parameters to describe a physical stream, called *stream characteristics* in the following. Based on the input stream characteristics of an individual operator, we estimate its output stream characteristics and resource usage. The resource utilization of a continuous query is finally obtained by summing up the resource usage of the individual operators it is composed of.

3.1 Purpose

A solid cost model is crucial to DSMS for the following reasons:

- While measurements only give information about the presence and past, a cost model makes it possible to estimate resource usage in the future.
- Continuously monitoring processing and memory costs at operator-, query-, and system level may be very expensive. Therefore, our cost model provides efficient formulas that are re-computed solely when current stream characteristics significantly differ from underlying past ones.
- Whenever a new query is posed, it allows deciding in advance whether sufficient resources are available to run the query or not.
- Our cost model can also be used to estimate the impact of (re-)optimizations on query plans, which includes information about stream rates and resource usage. Thus, it serves

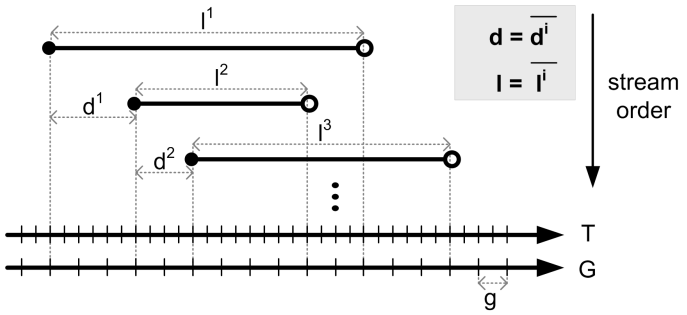

Figure 3: Model parameters

as an adequate basis for cost-based query optimization and adaptive resource management.

3.2 Model Parameters

During our work we identified three parameters, which characterize a data stream, as crucial to resource estimation. We model the *characteristics* of a physical stream S as a triple (d, l, g) where

- d is the average time *distance* between successive starting instants of time intervals in S,
- l is the average *length* of time intervals in S,
- g is the constant time distance of the time *granularity* of S.

Figure 3 demonstrates how these parameters are derived from a physical stream.[3] Note that all three parameters refer to application time.

3.2.1 Meaning

For a stateful operator, the knowledge of these three parameters for each input stream allows us to estimate the size of the status structures comprising its state. Informally, l describes the average time an element is relevant to the operator state, while d represents the average progress of time within a stream. If parameter d is known for each input stream, the average number of elements entering the operator state per application time unit can be computed. The additional information about l for each input stream makes it possible to determine the average number of elements being removed from the state per application time unit due to temporal expiration.

Recall that the operator state depends on the time granularity of its input streams as well. For most operators, those effects are marginal but not for the aggregation. The reason is that time intervals belonging to elements in the state are split and hence the size of the state increases (see line 4 in Algorithm 1). The larger the time granularity the smaller the state gets because

[3] Our model parameters can easily be transferred to the Positive-Negative Tuple Approach [3] which widens the applicability of our cost model. In that case, d corresponds to the time distance between two successive positive tuples and l is average time distance between a positive and its corresponding negative tuple in a stream.

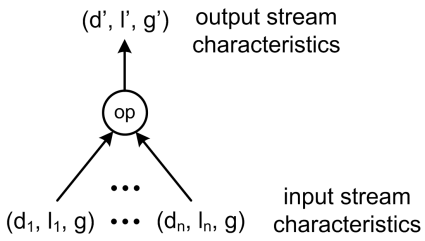

Figure 4: Stream characteristics

less splits occur due to the reduced number of possible start and end timestamps. Consequently, we need g as third parameter.

3.2.2 Source Streams

As our cost model works bottom up in a query plan starting at the sources, the model parameters of the source streams have to be obtained first. Parameter d is either known in advance because it is given for a certain application or the DSMS can easily compute it by online average. For instance, if an autonomous data source such as a sensor produces a stream element every k time units, d would simply be k. Thus, $\frac{1}{d}$ corresponds to the application stream rate. If the time granularity of a source stream is not provided by the application, it is set to the finest time granularity available in the DSMS by default. As elements in a source stream are only valid for a single time instant, $l = 1$.

Our definition of parameter d assumes stream rates to be steady on average. This is a common assumption for cost models [25, 5], which for instance holds for Poisson-distributed arrival rates. Whereas this seems not adequate for many applications in the long term because stream rates can change over time, it is usually appropriate in the short term. In order to improve the accuracy of our cost model and, in turn, to increase the degree of adaptivity of the DSMS, we therefore *update* parameter d whenever the difference between its current value and the one used by our cost model exceeds a certain threshold. The update can either be triggered by the application or by an online average over d with exponential smoothing as temporal weighting strategy [14]. Whenever parameter d was updated for a source stream, all dependent cost estimations should be re-computed. The trade-off between accuracy and computational overhead is controlled by the threshold parameter.

Be aware that the application stream rate $\frac{1}{d}$ is independent of the internal stream rates of a DSMS which highly depend on scheduling, i. e., the number of elements transferred per unit in system time [25]. Our parameter d is not affected by scheduling at all.

3.3 Parameter Estimation

Before cost formulas for the output stream characteristics of the operators are derived, we specify underlying assumptions and explain how our parameter estimation is used.

3.3.1 Overview

PIPES stores stream characteristics as metadata information in its runtime environment. This involves characteristics of all physical streams belonging to one of the currently executed queries.

Our cost model provides cost formulas for each individual operator. The output stream characteristics of an operator are computed by applying the corresponding operator cost formula to its input stream characteristics (see Figure 4). For a given query, we compute all intermediate stream characteristics by starting at the sources and applying our cost formulas bottom up. For the case of subquery sharing, i. e., when a new query is placed on top of already running queries, we use the stream characteristics at the connection points to initialize the cost formulas of the new parts. Starting at the connection points, the unknown stream characteristics of the new parts are then replaced by applying the cost formulas bottom up.

3.3.2 Assumptions

The formulas for the various operations only give approximate values as they rely on the following assumptions. First, we assume the streams to be infinite. Second, the parameters $l, d \in \mathbb{R}$ with $l, d > 0$, usually, $d \ll l$. The *empty* stream is defined by $d = \infty$ and $l = 0$. Third, for operations with multiple input streams, we demand the time granularities, i. e., model parameter g, to be equal. This can be achieved by applying the granularity operator to all inputs, setting the granularity to the finest common subgranularity. Finally, we assume no correlation between the values appearing in a data stream and the timestamps. The re-estimations performed for adaptation purposes, however, diminish possibly existing correlation effects.

3.3.3 Operator Formulas

In the following, (d, l, g) and $(d_i, l_i, g), i \in \{1, \ldots, n\}$ denote input stream characteristics, whereas (d', l', g') denotes the output stream characteristics as shown in Figure 4. For the special case that all input streams of an operator are empty, the operator has the empty stream as output by default – a case not explicitly handled in the following.

COST FORMULA 1 (PROJECTION π). For the projection,

$$d' = d, l' = l, \text{ and } g' = g. \tag{1}$$

DISCUSSION. Because the projection function is applied to the tuple component of stream elements, the projection has no effect on the assigned time intervals. Thus, the parameters modeling the characteristics of the input stream remain the same for the output stream.

□

COST FORMULA 2 (SELECTION σ). Let sel be the selectivity of the selection predicate, $0 < sel \leq 1$. For the selection,

$$d' = \frac{1}{sel} d, l' = l, \text{ and } g' = g. \tag{2}$$

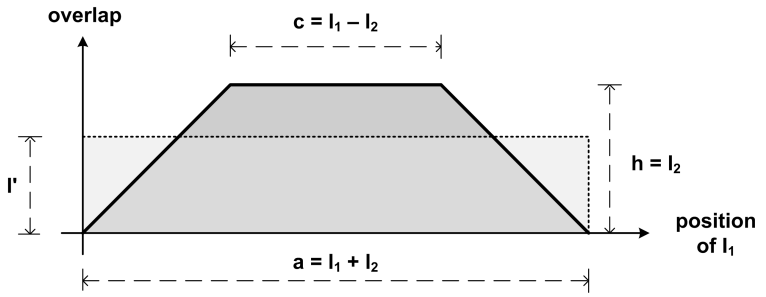

Figure 5: Estimation of l' for the Cartesian Product

DISCUSSION. According to our assumptions, a stream element is dropped with a probability of sel on average. Thus, the average time distance between successive elements is increased to $d' = \frac{1}{sel}d$ and the average length of the associated time intervals remains unchanged.

□

COST FORMULA 3 (UNION $\cup$). For the union of n input streams,

$$d' = \frac{1}{\sum_{i=1}^{n} \frac{1}{d_i}}, l' = \frac{\sum_{i=1}^{n} \frac{l_i}{d_i}}{\sum_{i=1}^{n} \frac{1}{d_i}} \text{, and } g' = g. \tag{3}$$

DISCUSSION. For each input stream, the average input stream rate r_i per time unit corresponds to $\frac{1}{d_i}$. As the union merges all input streams, the rate r' of the output stream is the sum of the input rates, $r' = \sum_{i=1}^{n} r_i = \sum_{i=1}^{n} \frac{1}{d_i}$. Due to the reciprocal relation between the average stream rate and the average time distance between two successive starting instants, it follows that $d' = \frac{1}{r'} = \frac{1}{\sum_{i=1}^{n} \frac{1}{d_i}}$.

The average length of time intervals in the output stream l' is the weighted average over the l_i, $i \in \{1, \ldots, n\}$, where the weight is the stream rate $r_i = \frac{1}{d_i}$. Intuitively, r_i expresses how many time intervals with average length l_i start at each single time instant. Thus, the average output length l' results from the sum of the weighted l_i divided by the sum of the weights.

□

COST FORMULA 4 (CARTESIAN PRODUCT $\times$). For the Cartesian Product of two streams,

$$d' = \frac{d_1 d_2}{l_1 + l_2}, l' = \frac{l_1 l_2}{l_1 + l_2}, \text{ and } g' = g. \tag{4}$$

DISCUSSION. For each input stream, the Cartesian Product maintains a separate status structure that stores all elements as long as they can have a temporal overlap with future incoming elements. Hence, the elements of input stream i have to be kept for l_i time instants on average. According to the input stream rates $\frac{1}{d_i}$, the size of status structure i results from $\frac{l_i}{d_i}$. Due to the symmetric implementation, the second status structure is probed whenever an element from the first input stream arrives, and vice versa. Since all elements of the probed status structure qualify in a Cartesian Product, the number of results produced on average at a single time instant is $\frac{1}{d'} = \frac{1}{d_1}\frac{l_2}{d_2} + \frac{1}{d_2}\frac{l_1}{d_1}$.

The time interval assigned to a result of the Cartesian Product is the intersection of the time intervals of its two constituents. Hence, the average length l' of time intervals in the output stream corresponds to the average overlap of time intervals from the input streams. Let I_1 and I_2 be two time intervals where the size of I_1 is l_1 and the size of I_2 is l_2. Without loss of generality, we assume that $l_1 \geq l_2$. Furthermore, we assume that each overlap combination of the two time intervals is equiprobable. All possible combinations of overlap can be produced by shifting I_1 over I_2. Figure 5 shows the degree of overlap on the y-axis and the relative position of I_1 on the x-axis. The resulting figure has the shape of a trapezoid. The average overlap l' is the point at the y-axis where the surface area A_R of the rectangle defined by width a and height l' is equal to the surface area of the trapezoid A_T. That means $l' = \frac{A_R}{a} = \frac{A_T}{a} = \frac{\frac{1}{2}(l_1+l_2+l_1-l_2)l_2}{l_1+l_2} = \frac{l_1 l_2}{l_1+l_2}$.

A theta-join can be expressed as a composition of Cartesian Product and selection.

□

COST FORMULA 5 (SCALAR AGGREGATION α). For the aggregation, $g' = g$, and

$$\begin{array}{ll} d' = l' = \frac{g}{1-(1-\frac{1}{d})^{2g}} & \text{, if } d < l \\ d' = d, l' = l & \text{, otherwise} \end{array} . \tag{5}$$

DISCUSSION. For $d \geq l$, each incoming element usually results in a new aggregate (see Algorithm 1 lines 8 and 9). The average size of the state is 1 because elements in the state prior to the insertion of the new aggregate are removed due to temporal expiration (see lines 12 and 13). Thus, $d' = d$ and $l' = l$.

For the case $d < l$, the aggregation behaves similarly on average, namely an incoming element causes the removal of an aggregate from the state as well. However, the average size of the state can exceed 1 because aggregates are split at incoming start or end timestamps (see line 4). The number of splits determines the parameters d' and l'. This split frequency can be computed analogously to Cardenas's formula [12] as follows. $\frac{X}{d}$ start and end timestamps fall into a sufficiently large timespan of duration X on average. Thus, $2 \cdot \frac{X}{d}$ timestamps are distributed throughout X on average. According to the time granularity $\mathbb{G}$, the probability that a timestamp matches with an arbitrary but single time instant in G is $\frac{g}{X}$. The probability that all $2 \cdot \frac{X}{d}$ timestamps do not hit this time instant in G is $(1 - \frac{g}{X})^{\frac{2X}{d}}$. Consequently, the probability of the complementary event, namely that a start or end timestamp hits that time instant, is $1 - (1 - \frac{g}{X})^{\frac{2X}{d}}$. As X contains $\frac{X}{g}$ time instants belonging to G, the expected number of time instants matched by at least one start or end timestamp is $(1 - (1 - \frac{g}{X})^{\frac{2X}{d}}) \cdot \frac{X}{g}$. In order to retrieve the average time distance d', we have to compute $\frac{X}{(1-(1-\frac{g}{X})^{\frac{2X}{d}}) \cdot \frac{X}{g}}$. Replacing the auxiliary variable X by the concrete value $g \cdot d$ results in $d' = \frac{g}{1-(1-\frac{1}{d})^{2g}}$. As the state is a list of aggregates having adjacent time intervals, $l' = d'$.

The estimation above becomes imprecise if the implicit assumption of uniformly distributed timestamps in X is violated. The extreme case is an aggregation with the constraints that (1) d results from a constant start timestamp distance, (2) l results from a constant time interval length, and (3) l is a multiple of d. To overcome this deficiency, we propose to estimate $d' = l' = d$. In all other cases, the formula above provides acceptable results as confirmed by our experiments.

□

COST FORMULA 6 (GROUP-BY WITH AGGREGATION γ). Let $n \in \mathbb{N}, n > 0$ be the number of groups. For group-by with aggregation, $g' = g$,

$$\begin{array}{ll} d' = \frac{g}{n(1-(1-\frac{1}{n \cdot d})^{2g})}, \text{ and } l' = \frac{g}{1-(1-\frac{1}{n \cdot d})^{2g}} & \text{, if } n \cdot d < l \\ d' = d, l' = l & \text{, otherwise} \end{array} . \tag{6}$$

DISCUSSION. Group-By with aggregation can be considered as a combination of the following three steps. The first step is grouping. We assume the input stream to be equally split into n groups by a grouping function. Hence, grouping produces n streams, denoted by $S_1, \ldots, S_n$, where each stream contains the elements of a group. The second step is the scalar aggregation which is applied to each stream S_i independently. The third step is the union of all aggregation output streams which provides the final output stream.

Based on the assumption that grouping distributes the elements of S uniformly among the groups $S_1, \ldots, S_n$, $d_i = n \cdot d$ and $l_i = l$ for each stream S_i. Applying the cost formula for the aggregation (α) to each of these stream characteristics delivers the input stream characteristics for the union formula ($\cup$) whose result, in turn, corresponds to the final formulas for d' and l' listed above.

□

COST FORMULA 7 (WINDOW ω). Let $w \in T$ be the window size where $w \gg g$. For the window operator,

$$d' = \frac{g \cdot d}{l}, l' = \left\lfloor \frac{w+1}{g} \right\rfloor g \text{ , and } g' = g. \tag{7}$$

DISCUSSION. Let $(e, [t_S, t_E))$ be an incoming element of the window operator. According to the definition, the window operator extends the validity of an incoming element for each time instant $t \in [t_S, t_E)$ at time granularity $\mathbb{G}$ by w time units. Note that the timestamps of the resulting time intervals have to be in G to preserve the given time granularity. For that reason, new time intervals are generated by incrementing t_S stepwise by g time units as long as $t_S + k \cdot g \leq t_E, k \in \mathbb{N}_0$. The end timestamp of each time interval is set relatively to its start timestamp by adding $\left\lfloor \frac{w+1}{g} \right\rfloor g$. Hence, the window operator produces a sequence of elements as output, $(e, [t_S, t_S + \left\lfloor \frac{w+1}{g} \right\rfloor g)), \ldots, (e, [t_E, t_E + \left\lfloor \frac{w+1}{g} \right\rfloor g))$. As a consequence, $\frac{l}{d}$ elements overlap in the output stream at an arbitrary time instant on average. Because the window operator addresses time instants at granularity $\mathbb{G}$ and $\frac{l}{d}$ time intervals start at each time instant in G, the time distance d' between two successive starting instants in the output stream is $g/(\frac{l}{d})$. The length l' of the associated time intervals is $\left\lfloor \frac{w+1}{g} \right\rfloor g$ due to the rounding required to be compatible with the time granularity $\mathbb{G}$.

□

COST FORMULA 8 (GRANULARITY Γ). Let $\hat{g}$ be the time distance of the new time granularity $\hat{\mathbb{G}}$. Let $\hat{\mathbb{G}}$ be coarser than the time granularity $\mathbb{G}$ of the input stream which means that $\hat{g}$

is greater than g. For the granularity operator,

$$\begin{array}{ll} d' = d, l' = l - \hat{g}, \text{ and } g' = \hat{g} & \text{, if } l \geq 2\hat{g} \\ d' = \frac{d\hat{g}}{l-\hat{g}}, l' = \hat{g}, \text{ and } g' = \hat{g} & \text{, if } \hat{g} < l < 2\hat{g} \\ d' = \infty, l' = 0, \text{ and } g' = \hat{g} & \text{, otherwise} \end{array} . \quad (8)$$

DISCUSSION. The first case is the common one in practice, namely $l \geq 2 \cdot \hat{g}$. In order to determine l', we have to estimate the extent of l that is cut off by adjusting a time interval of average length l to the new granularity. For that reason, we have to consider the different overlaps between incoming time intervals and the inherent time intervals of length $\hat{g}$ given by the new granularity $\hat{\mathbb{G}}$. As we assume the starting instants to occur uniformly distributed within time distance $\hat{g}$, the average amount an incoming element is shortened turns out to be $\frac{\hat{g}}{2}$. The same argumentation can be applied for the ending instants. Consequently, a time interval with average length l is cut off $\frac{\hat{g}}{2}$ at its start as well as its end. Hence, $l' = l - \hat{g}$. Shortening each incoming time interval by $\frac{\hat{g}}{2}$ on average means that the time distance between two successive time intervals in the output is still d.

In the second case, the length l is between $\hat{g}$ and $2\hat{g}$. $\frac{l-\hat{g}}{\hat{g}}$ corresponds to the probability that an incoming time interval spans two successive time instants of the new granularity $\hat{\mathbb{G}}$ after rounding. For this case, output is produced with $l' = \hat{g}$. Otherwise, the incoming element is dropped because the start and end timestamp would be equal after rounding. Dropping elements causes the time distance d' to be larger than d, here, by a factor of $1/\frac{l-\hat{g}}{\hat{g}}$. The smaller l the larger the probability that time intervals are dropped.

In the last case, the output stream converges to the empty stream because the new granularity is larger than the average length of incoming time intervals. We cover this by setting $d' = \infty$ and $l' = 0$.

□

3.4 Estimation of Resource Allocation

Depending on the previous estimations for output stream characteristics, this section reveals the way we use our model parameters to estimate resource usage in terms of memory and processing costs. Our estimations refer to the resources required at steady state, i. e., after initializing an operator according to the sliding windows.

- M_{op} denotes the memory costs of an operator, i. e., the memory allocated for the elements comprising its state.
- C_{op} denotes the processing costs of an operator per unit in application time.

Based on our cost formulas, the resource utilization of a query plan is computed by summing up the individual costs of the involved operators.

3.4.1 Implementation Notes

Temporal Ordering In PIPES, a heap is used to restore the temporal ordering of the output stream of operators with multiple inputs, such as the join and union [19]. This approach does not

enforce a synchronized temporal processing of input streams as required in many other stream systems, e. g., [3]. As PIPES applies *heartbeats* [22] to cope with application time skew between streams and latency in streams, the heap size is kept small. This fact was also confirmed by our experiments. As a consequence, the memory and processing costs spent on heap maintenance are negligible. Although it would be possible to derive formulas to estimate an upper bound for the heap size based on application time skew and latency in the input streams, this is beyond the scope of this paper.

Temporal Expiration Stateful operators have to adhere our temporal semantics. This means, they additionally have to evaluate temporal predicates for probing and reorganization. A unique feature of our cost model is that these costs are considered. In order to reorganize efficiently, i. e., to purge the state of expired elements, all elements in a status structure are linked according to end timestamps. Because a separate heap is used for this purpose, maintenance costs are logarithmic in the size of the status structure.

3.5 Projection, Selection, Union

Projection, selection and union are stateless. Hence, no memory is allocated, i. e., $M_\sigma = M_\pi = M_\cup = 0$. The processing costs per application time unit are as follows:

- $C_\sigma = \frac{1}{d} \cdot C_P$ where C_P is the cost of a single evaluation of the selection predicate.
- $C_\pi = \frac{1}{d} \cdot C_F$ where C_F is the cost of a single evaluation of the projection function.
- $C_\cup = \frac{1}{d'} \cdot C_T$ where C_T denotes the costs to append a single element to the output stream.

Note that we do not explicitly consider the costs to append an element to the output stream in our operators because they are generally negligible. However, the factor for these costs can easily be determined by the output stream rate $\frac{1}{d'}$ as shown for the union.

3.6 Window and Granularity

3.6.1 Window

Let C_S be the cost of setting a timestamp. We distinguish between the following window types:

- For a *time-based sliding window*, $M_\omega = 0$ and $C_\omega = \frac{1}{d} \cdot C_S$. The end timestamp is set according to the window size for each incoming element.
- For a *tuple-based sliding window*, $M_\omega = c \cdot s$ and $C_\omega = \frac{1}{d} \cdot (2C_Q + C_S)$ where c denotes the number of elements in the window, s the size of an element in bytes, and C_Q the average costs to enqueue and dequeue an element, respectively. Because the end timestamp of an incoming element is set to the start timestamp of the c-th future element, a FIFO queue of size c is needed in the implementation.

Table 1: Variables specific to join costs

C_B	cost of a single hash bucket operation: insertion and removal
C_H	costs for a call of the hash function
C_P	costs for a call of the join predicate (including check for temporal overlap)
C_F	costs for building a join result (concatenation and projection)
C_E	costs for a call of the expiration predicate
C_R	costs to restructure the heap caused by insertion or removal of its top
$\sigma_{\bowtie}$	join selectivity factor
s_i	size of an element from input stream S_i in bytes
$\#B_i$	number of hash buckets in status structure i

3.6.2 Granularity

The granularity operator rounds the start and end timestamps of each incoming element according to the new time granularity. Temporal ordering is preserved under the assumption that the time granularity is only changed from a coarser to a finer granularity, at time instants belonging to the coarser granularity. Hence, the memory cost is $M_\Gamma = 0$, and the processing cost is $C_\Gamma = \frac{1}{d} \cdot 2C_R$ where C_R denotes the cost of rounding a timestamp.

3.6.3 Join

Without loss of generality, we only consider the symmetric hash join (SHJ) here (see [17] for implementation details). For each input, a hash table is maintained as status structure. The symmetric nested-loops join can be viewed as a special SHJ where each hash table only contains a single bucket and the costs to evaluate the hash function are set to 0. Table 1 shows the cost variables.

Memory Usage Temporal expiration is implemented input-triggered. Each incoming element from the second input stream triggers the reorganization of the first status structure. Directly after a reorganization, the average size of the first status structure is $\left\lceil \frac{l_1}{d_1} \right\rceil$ elements because only elements that have a temporal overlap with the incoming element are kept (see Section 2.4.4). Until the next reorganization takes place, the memory increases by $\frac{d_2}{d_1}$ elements. Thus, the average size of the first status structure is $\left\lceil \frac{l_1}{d_1} \right\rceil + \frac{d_2}{2d_1}$ elements. Due to symmetry arguments, the average size of the second status structure is $\left\lceil \frac{l_2}{d_2} \right\rceil + \frac{d_1}{2d_2}$. Consequently, we can estimate the total join memory allocation by

$$M_{\bowtie} \approx \left(\left\lceil \frac{l_1}{d_1} \right\rceil + \frac{d_2}{2d_1} \right) s_1 + \left(\left\lceil \frac{l_2}{d_2} \right\rceil + \frac{d_1}{2d_2} \right) s_2. \tag{9}$$

Processing Costs We estimate the processing costs of the join by summing up costs for insertion, probing, result generation and reorganization:

$$C_{\bowtie} \approx C_{insert} + C_{probe} + C_{result} + C_{reorg}. \tag{10}$$

Insertion Costs (C_{insert}): For each incoming element, (i) the hash function is called, (ii) the element is inserted into a bucket, and (iii) the temporal linkage is updated. As mentioned above, the elements are additionally linked by end timestamps to provide an efficient reorganization. This causes cost C_R because the element is inserted into a heap. As $\frac{1}{d_1} + \frac{1}{d_2}$ elements arrive per application time unit,

$$C_{insert} := \left(\frac{1}{d_1} + \frac{1}{d_2}\right) \cdot (C_H + C_B + C_R). \tag{11}$$

Probing Costs (C_{probe}): Each incoming element is probed against the opposite status structure. This includes the evaluation of the hash function and the probing of all elements in the corresponding hash bucket. As we assume the elements to be uniformly distributed among all buckets, a bucket of status structure i contains $\frac{1}{\#B_i}\frac{l_i}{d_i}$ elements on average, $i \in \{1, 2\}$.

$$C_{probe} := \frac{1}{d_1 d_2} \cdot \left(\frac{l_2}{\#B_2} + \frac{l_1}{\#B_1}\right) \cdot C_P + \left(\frac{1}{d_1} + \frac{1}{d_2}\right) \cdot C_H \tag{12}$$

Result Generation Costs (C_{result}): These costs result from the number of join results produced per application time unit multiplied with the costs for composing a result tuple.

$$C_{result} := \sigma_{\bowtie} \cdot \frac{l_1 + l_2}{d_1 d_2} \cdot C_F \tag{13}$$

Reorganization Costs (C_{reorg}): For each incoming element, a reorganization of the opposite status structure is triggered. While using the temporal linkage, at least the top element of the heap is compared with the incoming element for a temporal overlap. For a unit in application time, this causes cost of $\left(\frac{1}{d_1} + \frac{1}{d_2}\right) \cdot C_E$. For the case that an element expired, the status structure together with the heap has to be reorganized. Because the heap contains a reference to the element in the status structure, this can be done with constant costs C_B, even for the nested-loops case. However, removing and replacing the top element of the heap has logarithmic costs C_R in the size of the status structure. Because each elements in the status will expire due to the sliding windows specified, cost of $\left(\frac{1}{d_1} + \frac{1}{d_2}\right) \cdot (C_E + C_B + C_R)$ arise from reorganization. Altogether, the reorganization costs per application time unit are as follows:

$$C_{reorg} := \left(\frac{1}{d_1} + \frac{1}{d_2}\right) \cdot (2C_E + C_B + C_R). \tag{14}$$

3.6.4 Aggregation

The cost analysis is provided for the algorithm given in Section 2. Table 2 explains the cost variables. The status structure for the aggregation is a linked list ordered by start timestamps. An invariant of the algorithm is that elements in the state are disjoint with regard to their time intervals.

Memory Usage Aggregates with an end timestamp smaller or equal to the start timestamp of the incoming element are removed from the state (see lines 12 and 13). Prior to this reorganization step, the state is updated. Some of those new aggregates arise out of existing ones by

Table 2: Variables specific to aggregation costs

C_U	costs to update a single aggregate
C_I	costs to initialize a single aggregate
C_O	costs to check two elements for temporal overlap
C_L	cost of a single list operation: append and remove first element
s	size of a single aggregate in the state in bytes

splitting at the start timestamp of the incoming element (see Algorithm 1 line 4). The resulting aggregate with the time interval left of the split is removed from the state during reorganization. Therefore, we only have to consider the number of splits triggered by end timestamps when computing the size of the state. We deduce the average memory usage of the aggregation from Cardenas's formula [12], similarly to the discussion of Cost Formula 5 but only considering splits caused by end timestamps.

$$M_\alpha \approx \left\lceil \frac{l}{g} \cdot \left(1 - \left(1 - \frac{1}{d}\right)^g\right)\right\rceil \cdot s. \tag{15}$$

This formula holds for $d < l$. Otherwise, we estimate $M_\alpha \approx s$ as the state contains only a single element on average.

Processing Costs We estimate the processing costs of the aggregation as the sum of costs to update aggregates, create new aggregates, and to reorganize the state:

$$C_\alpha \approx C_{update} + C_{new} + C_{reorg}. \tag{16}$$

Update Costs (C_{update}): $\left\lceil \frac{l}{g} \cdot \left(1 - \left(1 - \frac{1}{d}\right)^g\right)\right\rceil$ end timestamps fall into a time interval of length l on average. In the average case, the time interval of an incoming element overlaps with that number of aggregates in the state which, in turn, have to be updated.

$$C_{update} := \begin{cases} \frac{l}{dg} \cdot \left(1 - (1 - \frac{1}{d})^g\right) \cdot (C_O + C_U) & , d < l \\ 0 & , d \geq l \end{cases} \tag{17}$$

Costs for New Aggregates (C_{new}): The ratio $\frac{d}{d'}$ corresponds to the number of new aggregates generated for an incoming element on average. A new aggregate is created whenever an existing aggregate is split (see line 4), or its start or end timestamp is larger than the end timestamp of the last aggregate in the list (see lines 9 and 11). The new aggregates are added to the status structure (list) which can be done with constant cost C_L during the sequential scan performed to update the aggregates. While the costs to update existing aggregates have been considered in C_{update}, the initialization costs for a new aggregate (lines 9 and 11) are included in C_{new}. Because both cases happen with the same probability, cost of $\frac{1}{2}\frac{d}{d'}(C_L) + \frac{1}{2}\frac{d}{d'}(C_L + C_I)$ arise from an incoming element on average. Multiplied with the number of elements arriving per application time unit $\frac{1}{d}$, we estimate

$$C_{new} := \begin{cases} \frac{1}{d'} \cdot C_L + \frac{1}{2d'} \cdot C_I & , d < l \\ \frac{1}{d} \cdot (C_L + C_I) & , d \geq l \end{cases}. \tag{18}$$

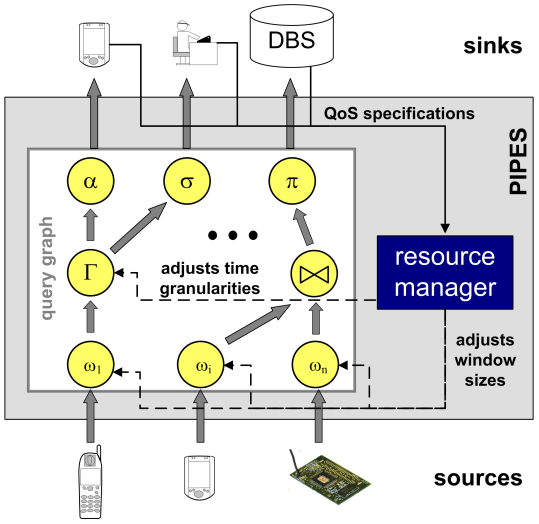

Figure 6: Adaptive resource management

For the case $d \geq l$, the average size of the state is 1. This implies that each incoming element produces a new aggregate which has to be initialized and appended to the list.

Reorganization Costs (C_{reorg}): Reorganization is a sequential scan of the list triggered for each incoming element (see lines 12 and 13). During traversal all aggregates having an end timestamp smaller than the start timestamp of the incoming element are removed and emitted as results. Since the reorganization is performed after updating the state, the traversal stops with a miss at the aggregate having the start timestamp of the incoming element. Those overlap misses cost $\frac{1}{d}C_O$ per application time unit. As each incoming element produces $\frac{d}{d'}$ aggregates on average, which are removed from the state sometime due to window constraints, cost of $\frac{1}{d}\frac{d}{d'}(C_O + C_L)$ arise from temporal overlap hits followed by list removals. Note that $d = d'$ for the case $d \geq l$.

$$C_{reorg} := \frac{1}{d'} \cdot (C_O + C_L) + \frac{1}{d} \cdot C_O \tag{19}$$

4 Adaptive Resource Management

This section outlines how to exploit our cost model for adaptive resource management. Our recent work [11] motivated that a DSMS can keep resource usage within bounds by adjusting window sizes and time granularities. Using our cost model it is now possible to quantify the impact of these adaptation methods on the resource usage of a query plan. In the following, we (i) clarify when a DSMS needs to apply the adaptation methods, (ii) describe their runtime behavior, and (iii) propose a non-trivial, adaptive strategy for resource management.

4.1 Runtime Adaptivity

In PIPES, the resource manager is the runtime component that controls resource utilization (see Figure 6). Based on our cost model, the resource manager is able to estimate the average

resource utilization of query plans with only low computational overhead. We distinguish between two cases that require adaptive resource management: changes in query workload and changes in stream characteristics. In both cases, the resource manager profits from our cost model because it can determine the degree of adjustments necessary to keep resource usage within bounds in advance. As a consequence, resource management is proactive, instead of being reactive without a cost model.

4.1.1 Changes in Query Workload

Whenever a new query is posed, the DSMS checks with our cost model if sufficient resources are available to evaluate the query (see Section 3.3.1). In particular, it considers adjustments to window sizes and time granularities to release resources in the running query graph. If enough resources can be allocated without violating QoS constraints, the query is accepted and integrated into the running query graph by the query optimizer. This includes updates to the QoS bounds at graph nodes and the assignment of memory to the newly installed stateful operators. Otherwise, the query is rejected. Whenever a query is removed and thus a significant amount of resources is released, our cost model can be used to determine strategies maximizing overall QoS by larger windows and finer time granularities.

4.1.2 Changes in Stream Characteristics

To keep up with changing stream rates, the resource manager monitors the start timestamp distance (parameter d) of all source streams. It compares the currently measured value of d with the one used for the latest estimation (see Section 3.2.2). Whenever the difference exceeds a certain threshold, a re-computation is performed to adapt cost estimations. Based on the new estimates, the resource manager can react appropriately to adjust overall resource utilization. As adjustments to window sizes and time granularities need a certain time to become fully effective, a DSMS should reserve a separate amount of resources to keep up with unpredictable behavior such as bursts in source streams. If an upper bound for the stream rate is known for an application, the resource manager can take advantage of this information to compute the resource utilization for the worst case, namely if bursts occur. Proactive resource management is feasible in those applications where forecasting techniques predict trends in stream rates or recurring stream patterns.

4.2 Adaptation Delays

While stateless operators are not affected by our adaptation methods, stateful operators react to adaptation effects with a delay due to temporal expiration (see Section 2.4.4). Hence, it is important to know the point in time when those effects start becoming visible and when they are fully achieved.

- **Aggregation.** Let S be a the output stream of a granularity operator with characteristics (d, l, g). For an aggregation having S as input, the effect of changing the time granularity starts immediately and is fully accomplished after l time units for the case $d < l$. This duration corresponds to the application time required for purging the state of old elements,

i. e., elements being in the state prior to the granularity change. In the case $d \geq l$, the full effect is achieved immediately at arrival of the next incoming element since all elements in the state already expired.

- **Join.** For a join applied to two window operators over S_1 and S_2 with characteristics (d_1, l_1, g) and (d_2, l_2, g), respectively, a change of a window size is not visible instantaneously. Without loss of generality, let us assume the first window being changed at time instant t, i. e., l_1 is set to l_1' where l_1' corresponds to the new window size. The change starts to be observable at $t + min\{l_1', l_1\}$ and reaches its full impact at $t + max\{l_1', l_1\}$. Again, this inference directly results from the temporal expiration concept.

4.3 Strategies

In order to validate the suitability of our cost model for adaptive resource management, we implemented a preliminary resource manager which has to tackle the following optimization problem. Let C and M be the given system resource bounds for processing costs and memory usage, respectively. The resource manager has to adjust window sizes and time granularities within their QoS range with the objective function to maximize overall QoS, while keeping the estimated overall resource utilization within bounds.

Let $[min, max]$ denote the QoS range of a window or granularity operator. For the granularity, min denotes the coarsest and max the finest granularity [11]. The basic idea of our strategy is to adjust all windows sizes and time granularities to the same fraction $f \in \mathbb{R}|_{[0,1]}$ of their QoS range. This means to set them to the closest allowed value next to $min + f \cdot (max - min)$. Due to our definition of time granularity (see Section 2.1), the number of possible granularities within the QoS bounds is limited. In contrast, window sizes can be changed in a fine-grained manner. For that reason, our strategy is implemented in two steps, a rough approximation which sets the granularities followed by a fine tuning of the window sizes:

1. Let $p \in \mathbb{N}, p \geq 1$ be an approximation parameter. For $j = 0, 1, \ldots, p$, the resource manager computes the overall resource estimations C_j and M_j using our strategy above with $f = \frac{p-j}{p}$ until the constraints $C_j \leq C$ and $M_j \leq M$ are satisfied. The granularities are fixed to their values in this last step.

2. A fine tuning of factor f for the window sizes is achieved by a bisection strategy.

Recall that this preliminary resource manager relies on a positive correlation between the factor f and the overall resource utilization. This holds for query graphs with selection, projection, union, join, and aggregation.

4.4 Extensions and Future Work

More sophisticated strategies for adaptive resource management are beyond the scope of this paper, but represent an interesting direction for future work. Studying the use of forecasting and pattern recognition techniques with the aim to extract information about future changes in stream characteristics and query workload is a promising approach towards proactive resource

management. In this paper, however, we put emphasis on the development and usage of our cost model since an accurate cost model is a prerequisite for adaptive resource management and cost-based query optimization.

5 Experiments

In this section we discuss the most significant results obtained from our large number of experiments under various settings, including real-world as well as synthetic data sets. While all these experiments performed as expected with regard to our cost model, we could only select a few of them due to space constraints. Because we want to validate our cost estimations for numerous applications, we picked out a different data set for each experiment.

Our experimental results (i) show that adjustments to window sizes and time granularities are suitable for adaptive resource management, (ii) validate the accuracy of our cost model, (iii) illustrate the differences between window reduction and load shedding, and (iv) prove the scalability of both, the techniques and our cost model, based on our preliminary adaptive resource manager.

We used the PIPES stream infrastructure as platform [19, 10, 17]. The experiments were performed on a Windows XP workstation with 1 GB main memory and a 2.8 GHz CPU. We determined concrete values for the cost variables in our model and counted the corresponding events at runtime, e. g., calls of the hash function. For a better overview, we plotted the measured processing costs as a percentage of saturation, i. e., 100% processing costs correspond to the capacity the system becomes saturated. Some figures have two y-axes. The y1-axis depicts memory costs, whereas the y2-axis shows processing costs.

5.1 Adjustments to Window Size

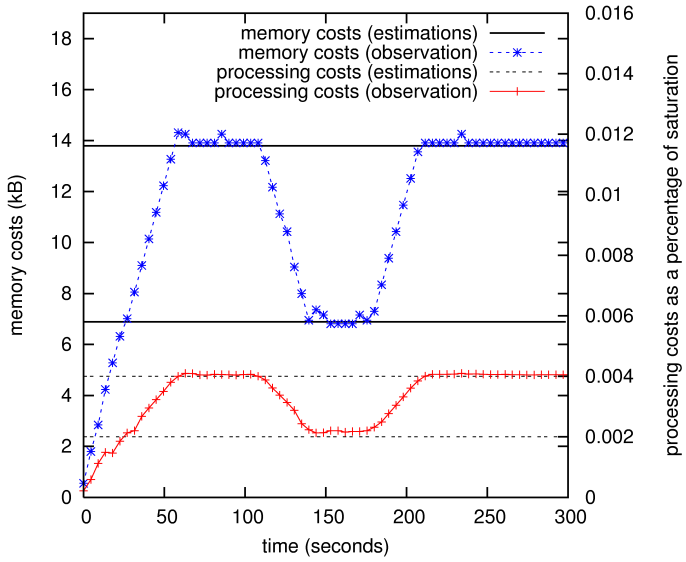

Figure 7: Changing the window size

In our first selected experiment, we point out that the resource usage of the temporal join operator is linear in the window size. This experiment relies on data streams obtained from a commercial system, called i-Plant, monitoring industrial production processes [10]. Two industrial sensors provide the input streams for our experiment. The first one measures the flow rate of a machine drying paper, while the second one measures the rate at which the paper is rolled up by the next machine. Differences in these rates indicate that the paper is at risk to tear. We computed a window similarity join to detect pairs of rates that differ by more than a given tolerance. Because we run this continuous query standalone, the utilized system resources were relatively low. Recall that a DSMS executes multiple thousands of such queries concurrently.

Figure 7 shows the effects of adjusting the window sizes together with their estimations obtained from our cost model. It starts with 60 seconds window size, and after 80 seconds the windows are reduced to 30 seconds. The resulting effect starts to be visible after the new window size ($80 + 30 = 110$) and reaches its full impact after the old window size ($80 + 60 = 140$), which agrees with our predictions. At 150 seconds, we set the window size back to 60 seconds. Then, the effects begin to be observable after the old window size ($150 + 30 = 180$) and are fully visible after the new window size ($150 + 60 = 210$).

5.2 Adjustments to Time Granularity

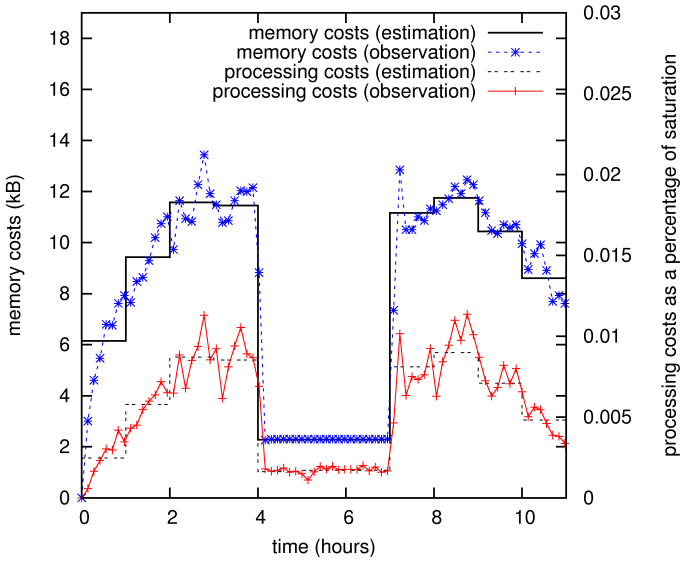

Figure 8: Changing the time granularity

Our second selected experiment demonstrates the impact of a time granularity adjustment on the resource usage of an aggregation. We used real-world data streams with traffic data collected in the Freeway Service Patrol Project (FSP) in 1993. We posed a continuous sliding window query with a COUNT aggregate, i. e., we computed the number of vehicles that passed a loop detector during the past 10 minutes. As the original granularity of the timestamps delivered from the sensor is relatively fine ($\frac{1}{60}$ seconds), almost every input element of the aggregation results in two output elements. During hour 4 to hour 7 of the experiment, we set the time granularity to

10 seconds. Since the average temporal distance between the event that two successive vehicles pass a sensor is lower than that granularity, we expected a significant decrease in the resource usage, which agrees with our observation shown in Figure 8. Moreover, our cost model gives a good approximation to the measured costs. Note that the estimation was updated hourly and thus this experiment also outlines the adaptivity potential of our approach. Having in mind that this real-world data set inherently contains correlation between values and time, the experiment additionally confirms that re-estimations compensate possible correlation effects.

5.3 Window Reduction vs. Load Shedding

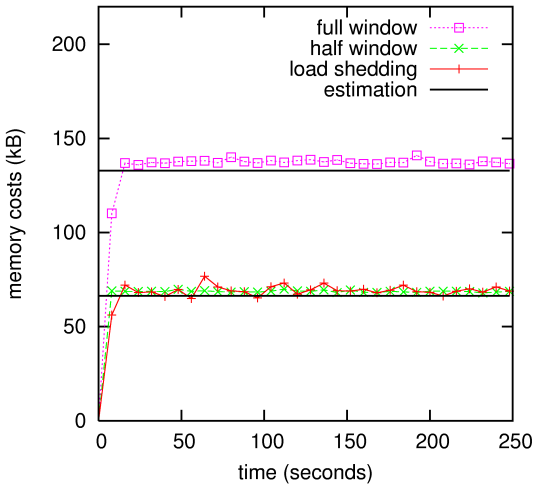

Figure 9: Savings in memory usage

The third experiment compares window reduction with random load shedding [24]. We computed an equi-join of two streams, each stream delivering random integers between 0 and 49 at a rate of 100 elements per second. Using a time-based sliding window of 10 seconds, the join needed about 140 kB memory as shown in Figure 9. We compare both techniques with the aim to achieve a memory saving of 50 per cent. For load shedding, this means to drop an element of the input stream with a probability of $\frac{1}{2}$. For window reduction, the window sizes have to be halved which means to assign validities of 5 seconds. As Figure 9 illustrates, both methods achieve their goal. With regard to processing costs, load shedding is more effective than window reduction – see Figure 10. While half a window size causes only half of the processing costs, load shedding reduces the processing costs by a factor of 4 because elements of both input streams are dropped with a probability of $\frac{1}{2}$.

Much more interesting is the number of results produced. In general, results of time-based sliding window joins are considered to be more important if the start timestamps of their constituents are sufficiently close to each other. We plotted the effect of both techniques on the number of results in Figure 11. For two joining elements, the x-axis shows the absolute value of the difference of their start timestamps. For sake of clarity, only each fifth value was plotted. In the case of load shedding, only about $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$ of the original join results was produced,

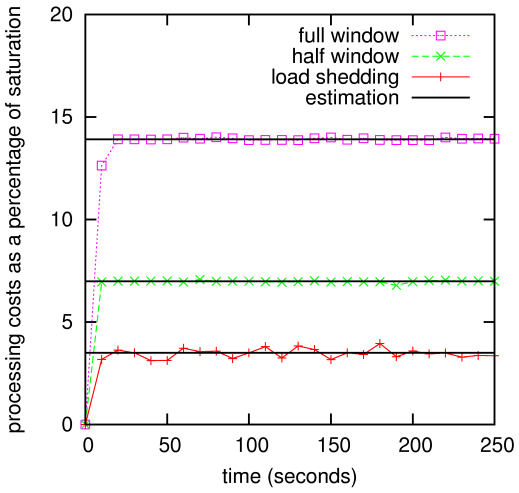

Figure 10: Savings in processing costs

whereas window reduction still preserved half of the results. A further advantage of window reduction is that it produces exact query results for elements whose start timestamps are close to each other (see semantic guarantees in [11]). This feature is important if query optimizations should be performed at runtime. For load shedding, query optimization becomes quite limited as, for example, the join over samples of its input does not return a sample of the join's output [15]. Hence, join reordering is not possible in general when random load shedding is applied. As a consequence, window reduction should be preferred to load shedding in those applications where exact results are important and resources need to be restricted, in particular memory usage.

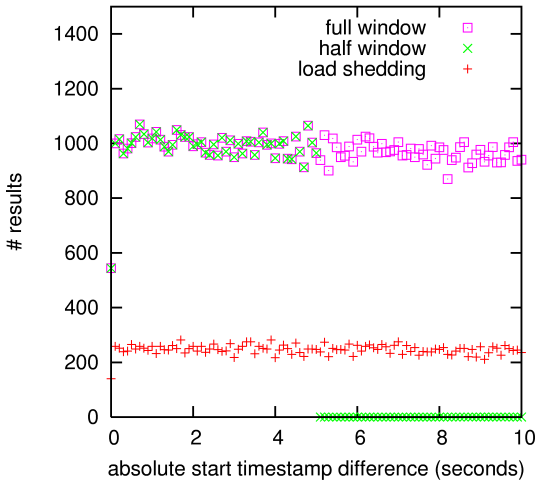

Figure 11: Output quality

5.4 Scalability

Our last experiment confirms the scalability of the techniques as well as the accuracy of our cost model. In addition, it proves that the techniques are suitable for adaptive resource management. We generated 5000 random continuous queries with time-based sliding windows. The query workload was randomly composed of the plans shown in Table 3 including aggregation, nested-loops (33% of all joins) and hash joins (67%). Join selectivities were chosen between 0.005 to 0.02. In addition, we uniformly picked QoS constraints for the window sizes with range from 1 to 30 minutes, and for the granularities with range from 1 to 10000 milliseconds. We used 50 synthetic data streams with Poisson-distributed stream arrival rates between 2 and 600 elements per minute, which implies that during the experiment 50 threads ran concurrently.

Table 3: Plan generation

plan types	percentage
$\omega(S_1) \bowtie \omega(S_2)$	30 %
$(\omega(S_1) \bowtie \omega(S_2)) \bowtie \omega(S_3)$	20 %
$((\omega(S_1) \bowtie \omega(S_2)) \bowtie \omega(S_3)) \bowtie \omega(S_4)$	10 %
$(\omega(S_1) \bowtie \omega(S_2)) \bowtie (\omega(S_3) \bowtie \omega(S_4))$	10 %
$\alpha(\Gamma(\omega(S)))$	10 %
$(\alpha(\Gamma(\omega(S_1)))) \bowtie \omega(S_2)$	10 %
$(\alpha(\Gamma(\omega(S_1)))) \bowtie (\alpha(\Gamma(\omega(S_2))))$	10 %

We implemented the preliminary adaptive resource manager with the bisection strategy explained in Section 4.3. During the experiment the resource manager had the objective function to adapt system memory usage to 100 MB. In order to demonstrate adaptivity as well as scalability, we steadily increased query workload. We started with 5000 queries. Since the operator states were empty at the beginning, it lasted about 30 minutes until the states became steady in size. Note that this time period corresponds to the maximum window size. After 1 hour, we started to pose 10 new random queries every 72 seconds for a duration of 30 minutes. Finally, we created a burst of 500 new queries after 2 hours total runtime. As shown in Figure 12, our preliminary resource manager succeeded in controlling the resource utilization dynamically. The periods of underestimation in Figure 12 result from adaptation delays (see Section 4.2). While the operators of a new query allocate resources immediately when the query is installed, the reaction of the resource manager in form of window reductions requires time to become fully effective. This can be avoided if the resource manager adapts resource utilization in advance, either by preventive strategies or by delaying the start of new queries. We prefer to keep a portion of the system resources as reserve which is also reasonable for the case of unpredictable stream behavior such as bursts.

We want to highlight the following points regarding scalability.

- The experiment ran for several hours with a workload of at least 5000 continuous queries.
- We had a passable memory usage around 100 MB.

- The processing costs were about 30% of the system saturation capacity. We chose these extent of resources because we did not want to have side effects caused by the our profiling overhead. Be aware that we exactly monitored all operator costs specified in our model like function and predicate calls.
- As our cost model only provides cost estimations instead of exact costs, and cost estimation runs bottom up in a query plan, the estimates get worse with the height. For that reason, we measured the average relative error in dependence of the plan height and observed a slight increase by at most 1% per level.

To summarize, both techniques turn out to be appropriate for adaptive resource management. The observed behavior is close to the one predicted by our cost model, which consequently provides a sufficient degree of accuracy.

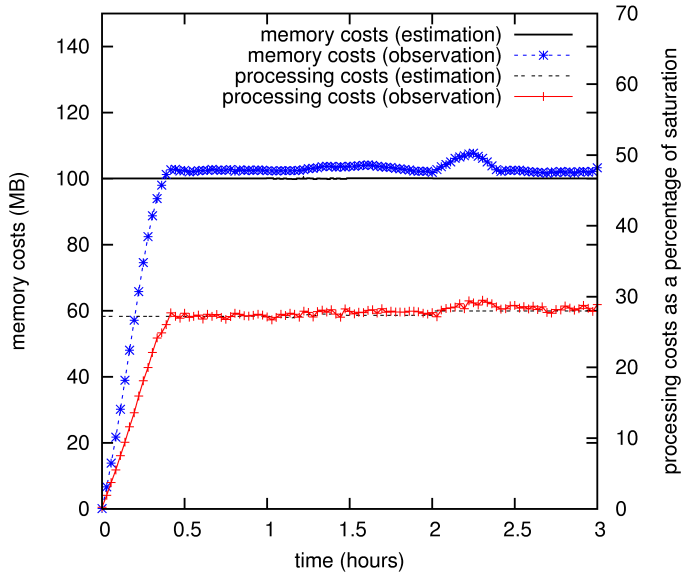

Figure 12: Scalability and adaptivity

6 Related Work

In the following, we concentrate on cost models for continuous queries to estimate resource requirements and methods for adaptive resource management. The interested reader is referred to [7] for a survey of work on adaptive query processing.

Adaptive query processing often relies on a cost model to choose the best query execution plan according to a heuristic method [23]. For data streams, the initial work on rate-based query optimization [25] introduces a cost model for SPJ queries with the main goal of maximizing the output rate of join trees. A different objective for sliding window multi-way joins is to minimize the number of intermediate tuples [16]. The cost model in [5] deals with the optimum placement of random load shedding operators to find an execution plan that minimizes resource usage when computational resources are insufficient. This problem is similar to ours but the

techniques applied are entirely different. Furthermore, our cost model is more extensive as it considers not only the memory usage of SPJ queries but also processing costs and contains additional important operators like aggregation. It is also more detailed because it shows the impact of time granularity on resource consumption as well as cost of temporal expiration in the case of time-based sliding windows. The cost model in [9] is used to answer in advance whether the QoS requirements of a query can be met or not. The QoS of running queries is guaranteed at operating system level. In contrast to [9], our approach is dynamic because it adapts to changing stream characteristics and query workload, but at the expense of weaker QoS guarantees. Additionally, the cost models assume different operator semantics.

k-constraints [8] are stream characteristics obtained from an online detection to automatically exploit conditions to reduce the runtime state. In [2], a broad range of continuous queries is categorized into two classes based on their asymptotic memory requirements and an algorithm is presented which determines whether or not a given query can be evaluated with bounded memory. Both papers neither address our techniques and QoS constraints nor give a cost model for estimating the resource usage of a query plan.

There has also been considerable research on approximate query processing over data streams in recent years. Load shedding [24] belongs to this class. Aurora [1] uses QoS information provided by external applications to control the degree of load to shed, usually CPU load. While standard load shedding randomly drops elements from a stream, semantic load shedding aims at maximizing QoS by dropping those elements that have a lower utility to the application. Compared to load shedding, our adaptation techniques preserve the generally accepted semantics of continuous sliding window queries [3, 19] since query results are exact for the chosen window sizes and time granularities [11]. This consequently enables query reoptimization at runtime [26]. Note that query optimization becomes quite limited for random load shedding as, for example, it is known from joins that the join over samples of its inputs does not return a sample of the join's output [15]. Another advantage of window reduction is that, even for small windows and high system load, it preserves those results computed from elements that are temporally close to each other, which is the basic idea of sliding windows. On the contrary, load shedding is very likely to drop some of those results. Work on approximate query processing is altogether complementary to this work which provides exact query results with regard to the chosen QoS.

The impact of operator scheduling on system resources at runtime is studied in [13, 6]. While the first paper [13] proposes scheduling strategies to meet QoS specifications, *Chain* scheduling [6] adaptively minimizes inter-operator queues in query graphs. These works are complementary to ours since they neither address the memory allocated for operator states, which is under normal system load the dominating part [8], nor analyze operator costs in detail.

7 Conclusions

Our recent work [11] motivated two widely applicable techniques for adaptive resource management in data stream systems, namely adjustments to window sizes and time granularities. We extended this work as follows. First, we not only investigated the impact of both techniques on

memory usage but also took processing costs into account. Second, we developed a solid cost model to estimate adaptation effects on the average resource utilization of a query plan. Third, we analyzed the runtime behavior of both methods and thus established a basis for cost-based adaptive resource management. The results obtained from our extensive experimental study prove the potential and scalability of both adaptation methods and validate the accuracy of our cost model. Finally, we want to point out that query optimizers can profit from our cost model as well, in particular against the background of our stream operator algebra which preserves the transformation rules known from the relational algebra.

Acknowledgments

This work has been supported by the German Research Foundation (DFG) under grant no. SE 553/4-3.

References

[1] D. J. Abadi and D. Carney et al. Aurora: A New Model and Architecture for Data Stream Management. *VLDB Journal*, 12(2):120–139, 2003.

[2] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Transactions on Database Systems (TODS)*, 29(1):162–194, 2004.

[3] A. Arasu, S. Babu, and J. Widom. An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations. In *Proc. of the Intl. Conf. on Data Base Programming Languages (DBPL)*, pages 1–19, 2003.

[4] A. Arasu, S. Babu, and J. Widom. The CQL Continuous Query Language: Semantic Foundations and Query Execution. Technical report, Stanford University, 2003.

[5] A. Ayad and J. F. Naughton. Static Optimization of Conjunctive Queries with Sliding Windows Over Infinite Streams. In *Proc. of the ACM SIGMOD*, pages 419–430, 2004.

[6] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator Scheduling for Memory Minimization in Data Stream Systems. In *Proc. of the ACM SIGMOD*, pages 253–264, 2003.

[7] S. Babu and P. Bizarro. Adaptive Query Processing in the Looking Glass. In *Proc. of the Conf. on Innovative Data Systems Research (CIDR)*, pages 238 – 249, 2005.

[8] S. Babu, U. Srivastava, and J. Widom. Exploiting k-Constraints to Reduce Memory Overhead in Continuous Queries over Data Streams. *ACM Transactions on Database Systems (TODS)*, 29(3):545–580, 2004.

[9] H. Berthold, S. Schmidt, W. Lehner, and C.-J. Hamann. Integrated Resource Management for Data Stream Systems. In *Proc. of ACM SAC*, pages 555–562, 2005.

[10] M. Cammert, C. Heinz, J. Krämer, T. Riemenschneider, M. Schwarzkopf, B. Seeger, and A. Zeiss. Stream Processing in Production-to-Business Software. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*, pages 168–169, 2006.

[11] M. Cammert, J. Krämer, B. Seeger, and S. Vaupel. An Approach to Adaptive Memory Management in Data Stream Systems. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*, pages 137–139, 2006.

[12] A. Cardenas. Analysis and Performance of Inverted Database Structures. *Communications of the ACM*, 18(5):253–263, 1975.

[13] D. Carney, U. Cetintemel, S. Zdonik, A. Rasin, M. Cerniak, and M. Stonebraker. Operator Scheduling in a Data Stream Manager. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 838–849, 2003.

[14] C. Chatfield. *The Analysis of Time Series: An Introduction.* Chapman & Hall, 1996.

[15] S. Chaudhuri, R. Motwani, and V. Narasayya. On Random Sampling over Joins. In *Proc. of the ACM SIGMOD*, pages 263–274, 1999.

[16] L. Golab and M. T. Öszu. Processing Sliding Window Multi-Joins in Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 500–511, 2003.

[17] J. Krämer and B. Seeger. A Temporal Foundation for Continuous Queries over Data Streams. Technical report, University of Marburg, 2004.

[18] J. Krämer and B. Seeger. PIPES - A Public Infrastructure for Processing and Exploring Streams. In *Proc. of the ACM SIGMOD*, pages 925–926, 2004.

[19] J. Krämer and B. Seeger. A Temporal Foundation for Continuous Queries over Data Streams. In *Proc. of the Int. Conf. on Management of Data (COMAD)*, pages 70–82, 2005.

[20] Y.-N. Law, H. Wang, and C. Zaniolo. Query Languages and Data Models for Database Sequences and Data Streams. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 492–503, 2004.

[21] G. Slivinskas, C. S. Jensen, and R. T. Snodgrass. Query Plans for Conventional and Temporal Queries Involving Duplicates and Ordering. In *Proc. of the IEEE Conference on Data Engineering (ICDE)*, pages 547–558, 2000.

[22] U. Srivastava and J. Widom. Flexible Time Management in Data Stream Systems. In *Symp. on Principles of Database Systems (PODS)*, pages 263–274, 2004.

[23] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 19–28, 2001.

[24] N. Tatbul, U. Cetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *Proc. of the Int. Conf. on Very Large Databases (VLDB)*, pages 309–320, 2003.

[25] S. D. Viglas and J. F. Naughton. Rate-based Query Optimization for Streaming Information Sources. In *Proc. of the ACM SIGMOD*, pages 37–48, 2002.

[26] Y. Zhu, E. A. Rundensteiner, and G. T. Heineman. Dynamic Plan Migration for Continuous Queries Over Data Streams. In *Proc. of the ACM SIGMOD*, pages 431–442, 2004.