Continuous Queries over Data Streams
– Semantics and Implementation

Jürgen Krämer

Philipps-Universität Marburg
kraemerj@informatik.uni-marburg.de

Abstract: Recent technological advances have pushed the emergence of a new class of data-intensive applications that require continuous processing over sequences of transient data, called data streams, in near real-time. Examples of such applications range from business activity monitoring and online analysis of sensor data to trend detection in stock ticker data. This work presents a solid and powerful foundation for processing continuous queries over data streams. We (i) define a sound semantics for continuous sliding window queries, (ii) introduce a unique stream algebra implemented with efficient online algorithms, and (iii) sketch the adaptive runtime environment. Instead of adapting view maintenance techniques, our work carries over and enhances findings from temporal databases to meet the challenging requirements of the data stream computation model.

1 Introduction

Traditional database systems have proven to be well-suited to the organization, storage, and retrieval of finite datasets. In recent years, however, new data-intensive applications have emerged that need to process data which is continuously arriving at the system in the form of potentially unbounded, time-varying sequences of data items, termed data streams. Examples belonging to this new class of stream-oriented applications can be found in a diversity of application domains including sensor monitoring, finance, transaction log analysis, and network security [SCZ05, GÖ03b, BBD+02]:

- **Sensor networks** are used in a variety of applications for monitoring physical or environmental conditions, for instance, in traffic management, location tracking, supply chain management based on the upcoming RFID technology, medical monitoring, and manufacturing processes. Queries may involve the detection of unusual and often complex conditions, even across multiple streams, with the aim to activate alarms or trigger actions.

- **Network traffic management** involves online monitoring and analysis of network packets to find out information on traffic flow patterns for routing system analysis, bandwidth usage statistics, and network security. In particular, intrusion detection requires real-time responses, for instance, to prevent denial-of-service attacks.
• **Electronic trading** often relies on the online analysis of financial data obtained from stock tickers and news feeds. Specific goals include discovering correlations, identifying trends, and forecasting stock prices. More current results maximize arbitrage profits.

• **Transaction logging** is performed in many applications, generating huge volumes of data, for example, web server logs and click streams, telephone call records, user account logging, or event logs in online auction systems such as eBay. Queries over these streams of log entries could be used to initiate immediate actions on specific customer behavior, detect suspicious access patterns that could indicate fraud or attacks, or identify performance bottlenecks with the aim to improve service reliability.

In recent years a dramatic escalation in feed volumes has been observable due to steady technological progress. For example, every day WalMart records 20 million sales transactions, AT&T generates 275 million call records, and eBay logs 10 million bids. Market data feeds can even generate multiple thousands of messages per second. While storing these massive data sets is possible to a certain extent, extracting valuable information from the resultant histories is often extremely expensive because the overwhelming data volumes accumulated over months or even years cannot be searched and analyzed in acceptable time. As a consequence, the archived data is often discarded after the retention time has elapsed, without any prior analysis but to free space for new data. Even in applications with less restrictive time requirements it is thus advisable to pre-process data streams in order to extract relevant information and reduce data volumes.

**Data Stream Management Systems (DSMSs)**  Traditional database systems are not designed to support high-volume, low-latency stream processing because queries are expected to run continuously and return new answers as new data arrives, without the need to store data persistently [BBD+02, GO03a]. For this reason, novel techniques and systems dedicated to the challenging requirements in stream processing have been developed.

In our work, we restrict the evaluation of continuous queries to *sliding windows* over the potentially unbounded data streams to keep resource requirements in bounds. This technique has the advantage that it emphasizes new data, which in the majority of real-world applications is considered more important than older data.

Rather than focusing on isolated research problems such as scheduling strategies or individual stream operators, our main objective has been to develop a general-purpose query engine for data streams. As part of our research, we have spent significant time and effort in the implementation of our Java library for data stream management called PIPES (*Public Infrastructure for Processing and Exploring Streams*) to demonstrate the feasibility and practical applicability of our methods. With PIPES, we pursue a library approach rather than building a monolithic system because we are convinced that it is almost infeasible to develop a general-purpose and still highly efficient DSMS for the plethora of streaming applications [KS04]. PIPES provides powerful and generic building blocks for the construction of a fully functional DSMS which can be tailored to the specific needs of an
application domain. The following sections sketch its query language, semantic foundations, algorithms and implementation concepts, and its adaptive runtime environment (see Fig. 1).

2 Query Formulation

Today, many applications dealing with continuous queries over data streams rely on hand-coded queries to directly process elements at delivery. Custom coding typically entails high development and maintenance costs and it is almost impossible to optimize queries automatically. PIPES overcomes these drawbacks by offering two ways to express complex application logic in an easy and comprehensible manner: (i) via a graphical user interface that enables users to specify data flow by combining operators from a stream algebra in a procedural manner, or (ii) via a declarative query language inheriting the basic syntax from the prevalent SQL standard.

As SQL is meant for one-time queries over relations and not for continuous queries over data streams, we basically enhanced the FROM clause with a WINDOW specification to express sliding window queries. Our grammar reuses and extends a subset of the window constructs definable in SQL:2003 for OLAP functions. Our query language is at least as expressive as the Continuous Query Language (CQL) [ABW06], while staying closer to SQL:2003 standards (see [Krät07] for a formal proof). It even allows users to define and include derived streams as well as complex nested queries with aggregates or quantifiers.

Example: Pair trading strategy – Notify me if the average ratio of the prices of stock AG1 and AG2 in the last 10 minutes deviates from the common ratio of 5 to 3 by more than 10%.
3 Query Semantics

Query formulation only makes sense if the query semantics is precisely defined. To guarantee predictable and repeatable query results, continuous query semantics should be defined independent of how the system operates internally. Unlike traditional one-time queries, continuous queries produce output over time. Therefore, it is important to incorporate the notion of time into the semantics. Since queries expressed in our query language are translated into a query plan composed of operators from our stream algebra, the semantics of a query results from the semantics of the contributing operators.

Logical Stream Algebra In analogy to traditional database systems, we distinguish between a logical and physical operator algebra. Our logical algebra precisely defines the semantics of each operation in a direct way by modeling data streams as temporal multisets.

In order to preserve the well-understood semantics of the relational algebra to the extent possible, our algebra contains a stream-counterpart for every operator in the extended relational algebra, except for sorting. Each of these, so-called standard operators is snapshot-reducible. We denote a stream operation $op_S$ with inputs $S_1, \ldots, S_n$ as snapshot-reducible to its relational counterpart $op_R$ if, for every time instant $t$, the snapshot at $t$ of the results of $op_S$ is equal to the results of applying $op_R$ to the snapshots of $S_1, \ldots, S_n$ 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 this instant. Fig. 2 illustrates the temporal concept of snapshot reducibility [SJS01].

In addition, our algebra provides novel window operators to define the scope of operations. Rather than integrating windows directly into operators, we separated the functionalities to avoid redundancy and facilitate the exchange of window types. The combination of window operators with our stream variants of the relational operators creates their windowed counterparts. The input of operators can be the output of an arbitrary operator plan. Overall, our logical algebra assigns an exact meaning to any continuous query at any point in time. As our logical algebra reveals the temporal properties of stream operations, it represents a valuable tool for exploring and validating equivalences.
Plan Generation

Parsing and translating a query from its textual representation into a logical plan closely resembles query plan construction in conventional database systems. Whenever the parser identifies a window expression following a stream reference, the corresponding window operator is installed in the query plan downstream of the node standing for the stream reference.

Example: Fig. 3 demonstrates the query plan for the pair trading example query, using the following operators: time-based window \( \omega_{\text{time}} \), Cartesian product \( \times \), map \( \mu \), scalar aggregation \( \alpha \), and filter \( \sigma \).

Query Optimization

We denote two query plans equivalent if they generate snapshot-equivalent results, i.e., if for any time instant \( t \) their snapshots are equal. Due to defining our standard operations in semantic compliance with [SJS01], the well-known transformation rules such as join reordering and selection push-down carry over from conventional and temporal databases to stream processing. Here, semantic compliance means that our stream operators generate snapshot-multiset-equivalent results to the temporal operators presented in [SJS01]. The conventional transformation rules are applicable to any snapshot-reducible subplans of the query graph, i.e., all contiguous plans composed of standard operators. For a snapshot-reducible subplan, any optimizations with conventional transformation rules lead to an equivalent plan.

In [Krä07], we enrich this extensive set of algebraic equivalences with novel rules for the window constructs. Besides these logical optimizations, we investigate several physical optimizations dealing with expiration patterns, stream ordering, and time granularity. We identify vital properties of subplans that enable the system to apply these optimizations. Altogether, our approach paves the way for powerful query optimizations. We refer the interested reader to [KS09] for further information.

4 Stream Algorithms

The data stream model introduces new challenges for the implementation of queries. First, algorithms no longer have random access to their input, only sequential access. Second, algorithms that need to store some state information from stream elements that have been seen previously, e.g., the join and aggregation, must be computable within a limited amount of space, while the streams themselves are unbounded. Third, some implementations of relational operators are blocking. Since streams may be infinite, blocking operators cannot be applied in the stream computation model because they would produce no output. Fourth, algorithms should process incoming elements on-the-fly and generate
output continuously over time. This implies that amortized processing time per stream element should be kept small.

**Physical Stream Algebra**  Our *physical stream algebra* reveals the implementation of continuous queries. It consists of push-based, nonblocking, stream-to-stream algorithms that produce results. Although results are semantically equivalent to the logical algebra, our physical algebra considers streams as sequences of tuples tagged with time intervals non-decreasingly ordered by start timestamps. Window operators set the lifespan of tuples according to the type and size of the window. Hence, the time interval of a stream element indicates the validity of the tuple according to the specified windows. Placing window operators upstream of stateful operators in a query plan avoids blocking and restricts resource requirements because tuples expire from the state.

While stateless operators such as selection or map do not consider the associated time intervals and thus can deal with potentially infinite windows well, time intervals affect stateful operators as follows. Elements are relevant to a stateful operator as long as their time interval may overlap with the time interval of any future stream element. This also means that a stateful operator can purge those elements from its state whose time interval cannot intersect with the time interval of any incoming stream element in the future. For operator state maintenance, our algorithms make use of generic sweepline data structures that allow for efficient probing and eviction of stream elements. See [Krä07, KS09] for details.

**Example:** The physical *join* operator has to satisfy the following two conditions to be snapshot-reducible: (a) for two contributing stream elements, the join predicate has to be satisfied for their tuples and (b) their time intervals have to intersect. The time interval assigned to the join result corresponds to the interval intersection. Our stream algorithm adapts the ripple join technique [HH99] to push-based query processing. The join state consists of two sweepline data structures, one for each input, and a min-priority queue at the output. For each incoming element, (i) expired elements are purged from the opposite data structure, (ii) the incoming element is inserted into the own data structure, (iii) join results are determined by probing the opposite data structure and inserted into the priority queue, and (iv) the minimum of the priority queue is extracted and appended to the output stream as long as the start timestamp ordering requirement of the output stream cannot be violated.

**Positive-Negative Approach**  To the best of our knowledge, our time-interval approach (TIA) is unique in the stream community. Related work embracing similar semantics implements the positive-negative approach (PNA) which propagates different types of elements, namely positive and negative ones tagged with a timestamp, through an operator plan to control expiration [ABW06, GHM+07]. The window operators are stateful and generate the positive and negative elements according to their semantics. As a consequence, stream rates are doubled which means that twice as many elements as in the time-interval approach have to be processed by operators in general.

Our implementation comparison and experimental studies conclude that TIA has definite
advantages for time-based sliding window queries, which is the most common type of query in streaming applications. Because TIA does not have to handle negative elements, TIA roughly outperforms PNA by a factor of two. Due to the stateless time-based window operator, TIA needs to allocate less memory than PNA. For queries composed of stateless operators, join, and duplicate elimination, results for TIA are available at the same time as the positive result is propagated for PNA. For grouping, aggregation, and difference, however, TIA has a slightly delayed response time, because operators cannot emit results as long as a temporal overlap with future elements is possible. The latency advantage for PNA also applies to queries with count-based or partitioned windows. However, it is questionable to which extent these windows are used in practice because of their often ambiguous semantics. See [Krä07] for details.

The fundamental difference between PNA and TIA is the way windowing constructs are implemented. TIA uses time intervals to model validity like in temporal databases, whereas PNA originates from the maintenance of materialized views.

5 Adaptive Query Execution

Adaptive Resource Management  Due to the long-running nature of queries, DSMSs need to control their resources adaptively since stream characteristics and query workload may vary over time. Our approach to adaptive resource management adjusts window sizes and time granularities to keep system resource usage within bounds [CKSV08]. Extensions to our query language allow the user to specify quality of service (QoS) ranges to restrict window sizes and time granularities. For time-based sliding window queries it is apparent that there is a direct correlation between window sizes and the memory usage of stateful operators. Because stateful operators have to keep all valid elements in their state, the state becomes larger the longer the validity is, and smaller the shorter the validity is (see Fig. 4(a)). The changes also affect CPU load, because the smaller the state the faster becomes element retrieval and eviction.
An advantage of this technique 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 just the basic idea of sliding windows [GO03b]. Our second QoS constraint refers to 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. Increasing the granule size associated with a stream in our physical algebra affects any downstream operators, particularly scalar aggregation and grouping with aggregation. The effects include savings in memory allocation and computational costs but also reductions to stream rates. Decreasing the granule size inverts the effects (see Fig. 4(b)).

Based on current runtime statistics, the resource manager has to adjust the window sizes and time granularities to keep the overall resource utilization within bounds. For this purpose, its objective function is to maximize overall QoS while keeping the individual settings for window and granule size within the pre-defined QoS ranges.

In contrast to random load shedding [TCZ+03], the main advantage of these methods is that the system can give reliable guarantees on query answers. Because query results are exact with regard to the selected QoS settings, adaptations do not collide with query optimization at runtime and consequently increase the flexibility and scalability of a DSMS.

Cost Model Our extensive cost model for the physical algebra allows for the estimation of resource consumption at operator-, query-, and system-level based on stream characteristics [CKSV08]. The stream characteristics involve the average inter-arrival time, the average validity of tuples, and the granule size. Based on our cost functions for the individual physical operators, the average memory costs, i.e., the amount of memory allocated for state maintenance, and the average processing costs per unit in application time can be estimated under acceptable system load at steady state. The resource utilization of a query plan is computed by summing up the individual costs of the involved operators.

Such a cost model is crucial to any adaptive runtime component in a DSMS because it can be used to quantify the effects of changes to query plans on the resource usage in advance. On the one hand, our resource manager employs the cost model to estimate the effects of adaptation techniques on query plans in order to determine to which extent window and granularity sizes need to be adjusted to keep resource usage in bounds. On the other hand, our query optimizer uses the cost model to select the best execution plan, namely the plan with the lowest estimated cost, from a set of semantically equivalent plans. Furthermore, the cost model enables the system to decide whether sufficient resources are available to accept a new query or not.

Our extensive experimental studies on real-world and synthetic data streams prove the accuracy of our cost model and the effectiveness of our adaptation techniques (see Fig. 4), even for experiments on larger settings where thousands of continuous queries were processed concurrently over several hours (see [CKSV08]).

Plan Migration PIPES is able to exploit query optimization with the aim to save system resources and improve scalability. Besides possibilities to optimize queries prior to execution and to share the execution of queries with common subexpressions, it provides
methods for dynamic query optimization [YKPS07, KYC+06]. GenMig, our novel plan migration strategy, performs a gradual transition from the old plan, which has become inefficient over time due to changes in the data characteristics, to the new, more efficient plan at runtime (see Fig. 5(a)).

The core idea of our approach is to define an appropriate split time representing an explicit point in application time that separates the processing of the old and new plan. Stream elements delivered by the input streams are assigned to the plans as follows. For all tuples valid at time instants prior to the split time results are computed by the old plan, otherwise by the new plan. The migration is finished as soon as the time progression in any input stream reached the split time.

Because GenMig treats the old and new plans as snapshot-equivalent black boxes, the complete set of transformation rules holding in our stream algebra becomes applicable for optimization purposes. This is a major advance over the existing parallel-track technique [ZRH04], which is limited in its scope to plans with stateless operators and joins. Furthermore, our experiments confirm that GenMig exhibits a smoother output rate, lower memory and CPU overhead and finishes migration earlier (see Fig. 5(b)).

6 Conclusions

Our work tackles one of the most challenging issues in continuous query processing, namely, the development of a coherent and efficient stream processing engine whose powerful processing primitives derive from suitable extensions of the relational algebra. In addition to our theoretical results that establish a semantically sound and powerful foundation for data stream management, we have also pointed out the practicability of our work by successfully applying PIPES to various application domains including traffic management, online auctions, sleep medicine, and factory automation. As the data stream model finds widespread use in modern applications, stream processing engines like PIPES are expected to gain considerable popularity.
References


