

Condensative Stream Query Language for Data Streams

Lisha Ma¹

Werner Nutt²

Hamish Taylor¹

¹School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, UK

²Faculty of Computer Science
Free University of Bozen-Bolzano, Italy

Abstract

In contrast to traditional database queries, a query on stream data is continuous in that it is periodically evaluated over fractions (sliding windows) of the data stream. This introduces challenges beyond those encountered when processing traditional queries. Over a traditional DBMS (Database Management System), the answer to an aggregate query is usually much smaller than the answer to a similar non-aggregate query making query processing *condensative*. Current proposals for declarative query languages over data streams do not support such condensative processing. Nor is it yet well understood what query constructs and what semantics should be adopted for continuous query languages.

In order to make existing stream query languages more expressive, a novel stream query language CSQL (Condensative Stream Query Language) are proposed over a sequence-based stream model (Ma & Nutt 2005). It is shown that the sequence model supports a precise tuple-based semantics that is lacking in previous time-based models, and thereby provides a formal semantics to understand and reason about continuous queries. CSQL supports sliding window operators found in previous languages and processes a declarative semantics that allows one to specify and reason about the different meanings of the frequency by which a query returns answer tuples, which are beyond previous query languages over streams. In addition, a novel condensative stream algebra is defined by extending an existing stream algebra with a new frequency operator, to capture the condensative property. It is shown that a condensative stream algebra enables the generation of efficient continuous query plans, and can be used to validate query optimisation. Finally, it is shown via an experimental study that the proposed operators are effective and efficient in practice.

Keywords: Data Stream, Stream Query Language, Window Aggregation, Sequence Model, Stream Algebra, Condensative Queries.

1 Introduction

Stream data can be characterised as an unbounded list of data elements that arrive in an order and at a rate over which a Data Stream Management System (DSMS) has no control. These can be found in a large variety of applications as diverse as stock ticker pro-

cessing, network traffic analysis, intrusion detection, sensor monitoring, web tracking and personalisation.

Compared to a traditional database architecture in which data is persistent and queries can be considered to be dynamic, in these new applications data can be considered active whereas queries are persistent or long-standing. As a result, traditional relational database technology designed to support applications over relatively static data has not proved entirely suitable for data stream applications.

A new research area, *data stream processing*, has generated considerable interest from both industry and research community seeking to bridge the gap between current technology and the needs of these emerged applications. This has resulted in more challenging requirements being generated for the data stream field with increasing interaction among different academic fields and a growing demand for information sharing to address many related research problems in semantics, query processing and runtime management. Among all the interesting issues, one major challenge is the development of techniques for providing continuously updating answers to aggregate queries over potentially unbounded streams. A general approach to addressing this challenge is by means of both window queries and a centralised query processing approach. Window queries add window clauses to continuous queries and allow aggregate queries to be evaluated over a segment of the input data stream rather than over the entire stream. There has been a surge of interest in the research issues involved in developing algorithms for windowed aggregate queries (Arasu & Widom 2004b, Chandrasekaran & Franklin 2002, Cranor, Johnson, Spataschek & Shkapenyuk 2003, Dobra, Garofalakis, Gehrke & Rastogi 2002, Dobra et al. 2002, Gilbert, Kotidis, Muthukrishnan & Strauss 2001, Li, Maier, Tufte, Papadimos & Tucker 2005).. Driven by different purposes, a number of Data Stream Management Systems (DSMSs) have been developed (Babcock, Babu, Datar, Motwani & Widom 2002, Abadi, Carney, Çetintemel, Cherniack, Convey, Lee, Stonebraker, Tatbul & Zdonik 2003, Yao & Gehrke 2003, Cranor et al. 2003, Chen, DeWitt, Tian & Wang 2000, Chandrasekaran, Cooper, Deshpande, Franklin, Hellerstein, Hong, Krishnamurthy, Madden, Raman, Reiss & Shah 2003) and several stream query languages (Arasu, Babcock, Babu, McAlister & Widom 2004, Dobra et al. 2002, Hammad, Franklin, Aref & Elmagarmid 2003, Arasu & Widom 2004a) have been recently proposed.

However, current techniques have two serious limitations. First, most current stream languages do not have the necessary language constructs to support *condensative* query processing over streams in the manner of traditional DBMSs (Database Management Systems). Aggregate query processing is called *condensative* in a traditional DBMS since the answer

to an aggregate query is usually much smaller than the answer to its non-aggregate counterpart. It is crucial that a declarative stream language supports condensative query processing over distributed data streams due to the large amount of data that is usually involved and the limited storage capacity that is usually available, such that aggregation will *not* be performed over a window where queries return new results whenever the window content changes. Second, the focus of previous work has mainly been on query execution while fundamental questions in connection with data models and formal semantics for queries have not yet been thoroughly addressed. The lack of this makes it difficult to reason about aggregate queries and compare different languages within a uniform semantics. Moreover, it makes it difficult to define a clear and sensible semantics for distributed data stream processing, as is usually the case when dealing with data streams.

Prior Work. Prior work in this area has been focused on operations and system architectures for data stream processing, to augment existing technologies and build new systems for stream-based applications, e.g., (Babcock et al. 2002, Abadi et al. 2003, Yao & Gehrke 2003, Cranor et al. 2003, Chen et al. 2000, Sullivan 1996, Chandrasekaran et al. 2003). Among the few established works on studying stream query languages, CQL (Continuous Query Language) (Arasu & Widom 2004a) is one of the most powerful relation-based languages. It is used in the STREAM system, and is proposed with full semantics over a time-based model. CQL provides advanced windowing capabilities, and it is even possible to PARTITION a window on an attribute and specify the width of a window (e.g. ROWS 100 or RANGE 100 MINUTES). However as the order of tuples is not uniquely defined in a time-based model, there is no clear semantics for continuous queries involving tuple-based sliding windows or moving aggregation. Another expressive language that supports condensation is StreaQuel, which is implemented in TelegraphCQ (Chandrasekaran et al. 2003). In StreaQuel, each query definition is followed by a for-loop construct that specifies (1) the set of windows over which the query is to be executed, and (2) how often the query should be run. Recently, Li et al. (Li et al. 2005) have proposed a similar, but more declarative way to define windowed aggregate queries. Their window definition has three parameters: RANGE specifies the window size, SLIDE the window movement, and WATTR the granularity, that is, whether RANGE and SLIDE are defined in terms of timestamps or sequence numbers of tuples. All these patterns were defined with respect to window identifiers. Such semantics define a function to identify uniquely each window extent for a given window aggregate query; also, they require an inverse function that, for each tuple, it determines the extents of the window to which the tuple belongs. Window identifier semantics was implemented in an extended version of the Niagara Query Engine (Chen et al. 2000) for evaluating aggregate window queries over data streams. In all three languages frequency and window length have to be defined in terms of the same granularity. Besides frequency can not exist independently without a window expression and has to be defined in a fixed place within a query due to the limited semantics to interpret different meanings of frequency, e.g., the frequency over the input stream cannot be differentiated from the frequency over the result stream.

A host of research exists on tackling aggregate query evaluation over data streams. Widely differing approaches employing, e.g., hashing, sampling, sketches and wavelets, just to name a few, have been

proposed in the literature (Golab & Özsu 2003, Babcock et al. 2002, Carney, Çetintemel, Cherniack, Conway, Lee, Seidman, Stonebraker, Tatbul & Zdonik 2002, Yao & Gehrke 2003, Arasu & Widom 2004b, Chandrasekaran & Franklin 2002, Cranor et al. 2003, Dobra et al. 2002, Gilbert et al. 2001, Li et al. 2005). All those approaches, however, are workload-driven, as opposed to being user-driven. In more detail, it is desirable to give the user considerable freedom in how windows are defined and handled by the system. In other words, the user should be in a position to declare window semantics at the language level, rather than rely on the system to deduce tradeoffs between accuracy and resource consumption, as is usually the case. This is what is termed condensative query evaluation: the user should be able to declare how frequently sampling should take place and the system should perform aggregation based on the user's instructions.

This paper introduces a Condensative Stream Query language (CSQL) over the sequence model. CSQL aims to extend the expressiveness of stream query languages along the dimension of answer frequency, which is a live issue for continuous queries and for which no analogy exist in classical databases. It is shown that CSQL supports sliding window operators found in previous languages, and a frequency operator. It is also shown that CSQL is capable of expressing useful sample queries such as queries with user-defined sampling, mixed jumping windows, and nested aggregation, which are beyond previous query languages over streams. More specifically, the main contributions are as follows.

1. Sampling and jumping windows are incorporated in a declarative fashion into a novel Condensative Stream Query Language (CSQL), along with its formal semantics and syntax over the sequence model (Ma & Nutt 2005). It is shown that CSQL can specify window queries found in (Arasu & Widom 2004a, Carney et al. 2002, Chandrasekaran et al. 2003, Chen et al. 2000, Yao & Gehrke 2003); also it supports (1) a way to express frequency requirements on both base and derived streams (2) user-defined group-based sampling (3) a mixed jumping window over streams (4) various forms of aggregation, e.g., nested aggregation. None of them can be realised in previous stream query languages.
2. A *condensative stream algebra* is introduced extending the existing stream algebra with a new kind of operator, called a frequency operator as well as showing its concrete semantics. Approaches to optimisation for the condensative model such as splitting, interleaving and compositional operations are discussed. Furthermore as an independent operator, the frequency operator can be easily pushed down in a stream algebra to avoid unnecessary computation based on *local* and *non-local* semantics. This allows the possibility of splitting aggregate query processing techniques into two levels, namely, tuple sampling and aggregation evaluation, which provides a flexible mechanism to interact with different advanced aggregate operators.
3. Finally, these ideas have been implemented in a prototype query engine. In order to demonstrate the efficiency gained by a pushed down frequency operator for a jumping window query over an ordered sequence stream, it is compared with the window Id approach presented in (Li et al. 2005). The experimental results show that a pushed down frequency operator is effective and outperforms the window Id approach and that a con-

densative stream algebra can be reasonably optimised for continuous queries with a frequency-based equivalence. It is also shown how to evaluate “mixed jumping window” queries, which cannot be handled by existing approaches.

Organisation. The remainder of the paper is organised as follows. Section 2 reviews sequence databases and the time-based data model. Sections 3, 4 and 5 introduce the semantics of the CSQL language. Language syntax is presented in section 6. Algebraic rules for the operators and example CSQL queries are given in section 7 and 8, respectively. The main algorithms for CSQL are given in section 9, while experimental results are presented in section 10. Section 11 concludes the paper and discusses future work.

2 Background

In this section two previous data models are reviewed, from which the proposed model draws some features.

2.1 Sequence Database Model

The SEQ sequence model and algebra were introduced by Seshadri et al. in (Seshadri, Livny & Ramakrishnan 1995). They define a sequence as an ordering function from the integers (or another ordered domain such as calendar dates) to the items in the sequence. The SEQ model separates the data from the ordering information and can deal with different types of sequence data by supporting an expressive range of sequence queries.

Some operators, such as selection, projection, various set operations, and aggregation (including moving windows) are carried over from the relational model. A number of operators for manipulating sequences have also been developed. The SEQ model has been implemented in SRQL (Sorted Relational Query Language) (Ramakrishnan, Donjerkovic, Ranganathan, Beyer & Krishnaprasad 1998), in which sequences are implemented as logically or physically sorted multi-sets (relations) and the language attempts to exploit the sort order.

2.2 Time-based Stream Model

In a data stream model, data items appear in a time-varying, continuously arriving, and append-only format.

A formal time-based stream model has been defined in (Arasu & Widom 2004a) and a declarative Continuous Query Language (CQL), including a formal semantics, has been defined. CQL has been implemented in the *STREAM* system at Stanford. The core of the model is as follows: Let \mathcal{D}_r be the set of all tuples that satisfy the schema r . Let \mathcal{T} be the set of all timestamps. Then a *stream* s with schema r is a subset

$$s \subseteq \mathcal{D}_r \times \mathcal{T},$$

such that for every $\tau \in \mathcal{T}$ the bag $\{\{e \mid \langle e, \tau \rangle \in s\}\}$ is finite. $\mathcal{P}(\mathcal{D}_r)$ denotes the set of all subsets of \mathcal{D}_r . A *time-dependent relation* R for the schema r is a mapping:

$$R: \mathcal{T} \rightarrow \mathcal{P}(\mathcal{D}_r),$$

such that each set $R(\tau)$ is finite. With these definitions, a stream can be transformed into a time-varying relation and vice versa.

3 Sequence Model

In a time-based model the order of tuples is not uniquely defined. This drawback leads to ambiguous semantics for continuous queries involving tuple-based sliding windows or moving aggregation. To address this issue, features are drawn from sequence databases and is constructed a sequence dependent model for streams. In this section the model and its formal semantics are introduced. As will be seen in the next two sections, the model is more expressive than the existing time-based model in (Arasu & Widom 2004a). The section concludes by describing the common properties that need to be shared by different data stream models, and that the new model aims to encode. An abstract *relational stream* is required to have the following characteristics:

- A stream consists of tuples;
- A stream has a relational schema and all its tuples comply with that schema;
- A stream develops over time. Therefore it is assumed that there is a set \mathcal{T} to represent the time domain, such as wall-clock time or the natural numbers. A *timestamp* is any value from \mathcal{T} . Timestamps are linearly ordered.

3.1 Relational Schema

A relation schema has the form:

$$r \langle a_1: T_1, \dots, a_k: T_k \rangle,$$

where r is a relation symbol, a_1, \dots, a_k are attributes, and T_1, \dots, T_k are types as in SQL. A timestamp is not included as a default attribute in the schema in case there is a need to separate various different timestamps associated with a tuple such as the tuple’s birth time or its arrival time.

3.2 Time Domain

There is no restriction on whether the time domain has to use wall clock time or the natural numbers. However it is still required that the general properties of the time domain be defined. A time domain should be *ordered*. Let $R \subseteq X \times X$ be an ordering (i.e, R is reflexive, antisymmetric and transitive). For every ordering R the strict version R' of R is defined by

$$xR'y \text{ iff } xRy \text{ and } x \neq y.$$

A binary relation is

Linear: if for all $x, y \in X$ where $x \neq y$ either $xR'y$ or $yR'x$

Dense: if for all $x, y \in X$ where $xR'y$, there is a $z \in X$ such that $xR'z$ and $zR'y$

Discrete: if for every two elements $x, y \in X$ where $xR'y$, there are only finitely many elements z between them, i.e, there are only finitely many z such that $xR'z$ and $zR'y$

If a linear ordering R' is discrete, then for every element $x \in X$, either at least one element y is such that $xR'y$, or there is no element y such that $xR'y$. A time ordering is required to have following properties:

1. Any two distinct timestamps must be comparable. This means, the ordering should be *linear*.
2. The ordering should not be *dense*, but *discrete*.

A time domain with these properties is essentially identical with the integers or an interval of the integers. A window of length n consists of the starting point plus the next $(n - 1)$ elements. If the time domain has a first element and is not bounded, then it can be represented by the natural numbers.

3.3 Local and Non-Local Semantics

A stream operator is a function Ω that takes a stream s as input and outputs a stream Ωs . Stream operators that apply to a stream can be *local* or *non-local*. Suppose there is an operator \mathcal{Q} that transfers a data stream s from a sequence model to a time based model $\mathcal{Q}s$, and $\mathcal{Q}s(t)$ represents a bag of the tuples that have timestamp t .

Definition 1 Ω is local if:

$$\mathcal{Q}s_1(t) = \mathcal{Q}s_2(t) \text{ implies } (\Omega(\mathcal{Q}s_1))(t) = (\Omega(\mathcal{Q}s_2))(t),$$

otherwise it is non-local.

Most relational operators are local such as selection σ , and most stream operators are non-local such as sliding window operators \mathcal{W} .

4 Stream Operators

It is shown that queries expressible in a time-based model can also be specified in this model. Furthermore, as will be seen in section 8, the model and operators are capable of expressing queries found in practice that are beyond previous models and languages. Next, some typical stream operators are introduced in this model.

4.1 Selection Operator

First the conditions for a selection operator are defined. A *term* is either an attribute name or a value constant. An *atomic condition* is an expression of the form

$$t_1 \rho t_2,$$

where t_1, t_2 are terms and ρ is a comparison like “<”, “≤”, “=”, “≥”, or “>”. Arbitrary conditions can be built up from atomic conditions using the boolean connectives “¬”, “∨”, or “∧”. Conditions are denoted by the letter C .

For every condition C a selection operator σ_C is defined. Intuitively, $\sigma_C(s)$ is the subsequence of tuples with index j of stream s , where $j \in \mathbb{N}$. $\sigma_C(s)$ is defined for an arbitrary stream s recursively by saying what it is the tuple $\sigma_C(s)(j)$ for an arbitrary number j . The set of indices I_1 is defined as

$$I_1 = \left\{ k \in \mathbb{N} \mid s(k) \text{ satisfies } C \right\}.$$

If $I_1 \neq \emptyset$, then let $n_1 = \min I_1$ and define $\sigma_C(s)(1) := s(n_1)$, otherwise let $\sigma_C(s) = \perp$. Now, suppose n_j is defined for some $j \in \mathbb{N}$. Then let

$$I_{j+1} = \left\{ k \in \mathbb{N} \mid s(k) \text{ satisfies } C \text{ and } k > n_j \right\}.$$

Again, if $I_{j+1} \neq \emptyset$, then let $n_{j+1} = \min I_{j+1}$ and define $\sigma_C(s)(j+1) := s(n_{j+1})$, otherwise let $\sigma_C(s)(j+1)$ be undefined. Also, $\sigma_C(s)(j+1)$ is undefined if n_j is undefined.

4.2 Sliding Window Operators

W_t will be used to denote a time-based window, and W_n will be used to denote a tuple-based sliding window.

Time-based Sliding Window

A time-based sliding window W_t is bounded by its temporal size t even though it is not known exactly how many tuples there are within the window size. However it slides whenever the time slot increases. The sliding rate will depend on the time granularity. $s^\tau(k)$ is introduced to denote the timestamp for tuple $s(k)$. More formally, the output stream $W_t s$ is defined as a sequence of sets $W_t s(j)$ for a given j in stream s . $W_t s(j)$ is not defined if $s(j)$ is not defined, otherwise

$$W_t s(j) = \left\{ s(k) \mid s^\tau(k) + t \geq s^\tau(j) \text{ and } k \leq j \right\}.$$

Tuple-based Sliding Window

A tuple-based sliding window will slide whenever a new tuple arrives. So, for every $n \in \mathbb{N}$, a tuple-based sliding window W_n over stream s produces a sequence of sets

$$W_n s(j) = \left\{ s(k) \mid k \geq \max\{0, j - n\} \text{ and } k \leq j \right\}.$$

4.3 Frequency Operator

The frequency operator F will pick the stream tuple based on a defined frequency. Depending on how the frequency is set, different types of frequency operators can be defined. Basically, the parameters can be set either by a physical bound (tuple-based) or a logical bound (time-based). In order to separate the different bounds, F_n and F_t are used to denote a tuple-based frequency operator and a time-based frequency operator respectively.

Tuple-based Frequency Operator

For every natural number $n \in \mathbb{N}$ a tuple-based frequency operator F_n selects every n -th tuple of a stream. Formally:

$$F_n s(j) = s(n \times j).$$

Time-based Frequency Operator

For every time instance t a time-based frequency operator F_t selects tuples with timestamp $j \times t$ as a stream $F_t s$, where $j \in \mathbb{N}$.

If there is no tuple with timestamp $j \times t$, then the last tuple is output within that time slot. $F_t s(j)$ is a subsequence of tuples with order j over order n_j of stream s , where $j \in \mathbb{N}$. Then if $s(n_j) \neq \emptyset$ let

$$n_j = \max \left\{ k \in \mathbb{N} \mid (j - 1) \times t \leq s^\tau(k) \leq j \times t \right\},$$

otherwise it is undefined. Now, $F_t s(j) = s(n_j)$, for all $j \in \mathbb{N}$ if n_j is defined, otherwise $F_t s(j) = \perp$.

4.4 Jumping Windows

Sometimes, the window needs to jump rather than slide. This can be achieved by applying a frequency operator to a sequence of sets instead of posing a frequency to a sequence of tuples. Such a kind of window is called a jumping window. Depending on how the frequency length is defined. Jumping windows are categorised into two different types: tuple-based jumping windows and time-based jumping windows.

Tuple-based Jumping Window

For every number $n \in \mathbb{N}$, and a sequence of sets Ws that are produced by the sliding window operators W_n or W_t , a tuple-based jumping window $F_n(Ws)$ selects every n -th set of Ws as follows:

$$(F_n(Ws))(j) = Ws(n \times j).$$

Time-based Jumping Window

For a sequence of sets Ws that are produced by a sliding window operator W applied to the stream s , A time-based jumping window $F_t(Ws)$ is obtained by selecting a subsequence $Ws(n_j)$ ($j \in \mathbb{N}$) of $Ws(n)$. Intuitively, $F_t(Ws)(j)$ is the first window that contains an element with a timestamp that is greater or equal to $t \times j$. Formally, for an arbitrary stream s and a window operator W , $F_t(Ws)$ is defined recursively by saying what it is the set $F_t(Ws)(j)$ for an arbitrary number j . The set of indices I_j is defined as

$$I_j = \left\{ i \in \mathbb{N} \mid \exists k. s(k) \in Ws(i) \wedge s^\tau(k) \geq j \times t \right\}.$$

If $I_j \neq \emptyset$, then $n_j := \min I_j$, and $F_t(Ws)(j) := Ws(n_j)$, otherwise let $F_t(Ws)(j) = \perp$.

Time-based Jumping Operator

For every time instance t , and a stream Ws that is produced by applying a sliding window operator W to a stream s . A time-based jumping window $F_t(Ws)$ is obtained by selecting a bag of tuples for every multiple of t from stream Ws . $(F_t(Ws))(\tau)$ is defined for an arbitrary time t , where $t \in \mathcal{T}$ as:

$$(F_t(Ws))(\tau) = \begin{cases} Ws(\tau) & \text{if } \tau = j \times t \text{ for some } j \in \mathbb{N} \\ \emptyset & \text{otherwise} \end{cases}$$

5 Condensative Stream Queries

A condensative stream query Q , in essence, is a traditional SPJ query augmented with frequency predicates (not necessarily have to be an aggregate query). Conceptually such queries have the ‘‘canonical’’ form of Eq. 1 in terms of relational algebra:

$$Q = \pi_* \mathcal{F}_{(p_1, \dots, p_n)} \sigma_{\mathcal{B}(c_1, \dots, c_m)} (R_1 \times \dots \times R_h) \quad (1)$$

That is, upon the product of the base relations $(R_1 \times \dots \times R_h)$, two types of operations performed with projected attributes (as indicated) are returned as the results.

Filtering: a Boolean function $\sigma_{\mathcal{B}(c_1, \dots, c_m)}$ filters the results by the selection operator $\sigma_{\mathcal{B}}$ (e.g., $\mathcal{B} = c_1 \wedge c_2 \wedge c_3$), and

Frequency: a Frequency function $\mathcal{F}(p_1, \dots, p_n)$ picks up the results from the base relations.

The goal is to support such condensative stream queries efficiently. Condensative stream models boolean filtering, i.e., $\sigma_{\mathcal{B}(c_1, \dots, c_m)}$ as a *first-class* construct in query processing. With algebraic support for optimisation, Boolean filtering is virtually never processed in the canonical form (of Eq. 1). Consider, for instance, $\mathcal{B} = c_1 \wedge c_2$ for c_1 as a selection over R and c_2 as a join condition over $R \times S$. The algebra framework supports *splitting* of selections (e.g., $\sigma_{c_1 \wedge c_2}(R \times S) \equiv \sigma_{c_1} \sigma_{c_2}(R \times S) \equiv \sigma_{c_1}(R \bowtie_{c_2} S)$) and *interleaving* them with other operators (e.g., $\sigma_{c_1}(R \bowtie_{c_2} S) \equiv \sigma_{c_1}(r) \bowtie_{c_2} S$). Their algebraic equivalences enable query optimisation to transform the

canonical form into efficient query plans by splitting and interleaving.

Such algebraic support, *splitting* and *interleaving* for optimisation, are completely inherited for frequency, i.e., $\mathcal{F}(p_1, \dots, p_2)$. Moreover, the support can be *compositional*. Suppose there is a frequency function $\mathcal{F} = p_1 \wedge p_2 \wedge p_3$, for p_1, p_2, p_3 as a frequency condition over R_1, R_2, R_3 respectively. p_1, p_2, p_3 are either all time-based or all tuple-based. Suppose $p_3 \bmod p_2 = 0, p_3 \bmod p_1 = 0, p_2 \bmod p_1 = 0$, then the frequency functions are *compositional* (e.g., $\mathcal{F}_{p_1} \mathcal{F}_{p_2} \mathcal{F}_{p_3}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1} \mathcal{F}_{p_2} \mathcal{F}_{p_2}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1} \mathcal{F}_{p_1} \mathcal{F}_{p_1}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1}(R_1) \times (R_2 \times_{\mathcal{F}_{p_1}} R_3)$). When queries are nested, frequency functions can be compositional even when the frequencies involved do not have the same granularity. (e.g, for a self-join query $(R_1 \times_{\mathcal{F}_{p_1}} R_2) \times_{\mathcal{F}_{p_2}} (R_1 \times_{\mathcal{F}_{p_1}} R_2)$, the inner frequency condition p_1 has the priority to synchronise the outer frequency condition p_2 , $(R_1 \times_{\mathcal{F}_{p_1}} R_2) \times_{\mathcal{F}_{p_2}} (R_1 \times_{\mathcal{F}_{p_1}} R_2) \equiv (R_1 \times_{\mathcal{F}_{p_1}} R_2) \times_{\mathcal{F}_{p_1}} (R_1 \times_{\mathcal{F}_{p_1}} R_2)$).

Finally relational algebra’s *pushing down* optimisation is extended into a stream algebra. The operators which can be used in a stream algebra are categorised by local and non-local semantics defined in definition 1 as such semantics assist the pushing down optimisation approach. An operator can be easily pushed down if it is a local operator such as a time-based frequency operator or a selection operator, otherwise not, i.e., suppose there is a time based frequency function \mathcal{F} , then $\mathcal{F}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}(R_1) \times \mathcal{F}(R_2) \times \mathcal{F}(R_3)$.

6 Syntax of CSQL

CSQL is a stream language that adds additional language patterns to SQL to support a stream processing ability. The core syntax of CSQL can be described with a context-free grammar.

```

string: represents for any valid string
number: represents any valid number
asterisk: represents *
<Query> → <Select><From> | <Select><From><Where> |
          <Select><From><Where><GroupBy>
<Name> → string | <Name>.<Name> | asterisk |
          <Name>AS<Name>
<Attribute List> → <Name> | <Name>(<Name>)*
<Granularity> → Milliseconds|Seconds|Minutes|Hours|Tuples|
                Millisecond|Second|Minute|Hour|Tuple
<Length> → number
<Frequency> → [<Fre>Partitioned By <Attribute List>]
              [<Fre>]
<Fre> → Frequency<Length><Granularity>
<Range> → Range<Length><Granularity>
<Compare> → > | < | = | = | <<
<Clause> → <Name><Compare><Name> |
           <Name><Compare>number
<op> → and|or
<Condition> → <Clause> | <Clause> (<op><Clause>)*
<term> → COUNT|SUM|AVG|MAX|MIN
<Aggregation> → <term> (<Name>)|
                <Aggregation> (<Aggregation>)*
<Select> → SELECT <Select Term> | <Select><Frequency>
<Select Term> → <Aggregation> | <Attribute List> |
                <Select Term> (<Aggregation>)*
<From> → FROM <From Term>
<LeftBracket> → (
<RightBracket> → )
<From Term> → <Name> | <Name><Frequency> |
              <Name> [<Range>] |
              <Name> [<Frequency>, <Range>] |
              <LeftBracket><Query><RightBracket>AS
              <From Term> |
              <From Term> (<From Term>)*
<Where> → WHERE <Condition>
<GroupBy> → GROUP BY <Attribute List>

```

7 Example Scenario and Queries

In practice there has been an increasing need for aggregate queries. To illustrate this, consider a scenario of a tracing system to study the behavior of wild animals, which collects distributed sensor measurements. One of the sensors records the pulse of an animal. Upon every heart beat of an animal it will send out a tuple with a timestamp and the animal’s ID. The schema of the relation for these measurements has the form

Pulse(Id, Timestamp)

The other type of sensors report on an animal’s blood pressure and body temperature regularly, for example every (full) second. It has a core relation

BodyCondition(Id, Species, BTemp, BloodP, Timestamp)

In these two relations: Id is the unique number of each animal, Species represents the type of animal, Timestamp represents the timestamp, BloodP is the blood pressure of the animal, and BTemp is the animal’s temperature. For ease of presentation, it is assumed that tuples arrive in the order of their timestamp attribute. Here are four queries with requirements on how often to evaluate them.

1. **Simple sampling query:** For every 100 tuples, report all horses’ body condition records.
2. **Latest result query:** Report the latest results of measurement on blood pressure and body temperature for each animal at the rate of one reading every minute, and evaluate the query for every 100 arriving tuples.
3. **Aggregate query:** For each animal, what is the pulse rate per minute? It is supposed that the user wants to know the result for every 10 tuples.
4. **Nested aggregate query:** For each animal, what is the average pulse rate per hour? It is supposed that the user wants to make use of the answers to the first query and expects a result every minute.

8 CSQL vs. Condensative Stream Algebra

The Condensative Stream Query Language (CSQL) is similar to SQL but extended with operators such as those discussed in Section 4, as well as a mechanism for directly submitting plans in a stream algebra that underlies the language.

In the CSQL language, a frequency operator can be expressed by adding to a range variable of a stream, say S , the expression [Frequency F], where F denotes an interval length. The length is either defined in terms of a number of tuples as [Frequency n Tuples] (“every n tuples”) or in terms of a time period, e.g. as [Frequency t Minutes] (“every t minutes”). The operator picks tuples based on the predefined frequency length from the base stream. For group-based sampling [Frequency F Partitioned By A_1, \dots, A_k] is used. The operator will partition S into different substreams based on the grouping attributes A_1, \dots, A_k , then for each substream the operator picks tuples based on the predefined frequency. The frequency is separated for an input stream and a result stream by putting the frequency expression in either the FROM clause or the SELECT clause.

The frequency operator can be combined with sliding window operators when the window needs to move much faster. Such a kind of window is called a *jumping window*. When Frequency = 1 Tuple, it is equivalent to a normal sliding window. Depending on how

the frequency length is defined, *tuple-based* and *time-based* jumping windows can be distinguished. Instead of computing the answer whenever a new tuple arrives, the frequency operator requires a computation only after an interval of the frequency length. This means that, the operator will take a “nap” between any two computations. A jumping window has two parameters:

The window size W . All of the tuples that arrive from the start during a period of W , or within W tuples have to be stored for a computation.

The length of the “nap” F . A new window is only output after the nap is over.

A jumping window is always defined by a sliding window operator, followed by a frequency operator expression, such as [Range W , Frequency F]. The semantics supports the mixing of tuple-based frequencies with time-based window bounds and vice versa. Such windows are called “mixed jumping windows”.

To enable frequency-based query processing and optimisation, the relational algebra (Kießling 2002) is extended into a stream algebra (Babcock et al. 2002) by substituting relations for streams, where the operators, and algebraic laws “respect” and take advantage of the “compositional” property introduced in section 5. Such a stream algebra is extended by adding the new operator frequency operator \mathcal{F} and the sliding window operator \mathcal{W} , and so generalise the existing relational operators (e.g., $\pi, \mathcal{A}, \sigma, \mathcal{G}$ in figure 1) to be “frequency-based”. CSQL is shown to be declarative by expressing the frequency in different places within a query. It is also shown how those differences affect the stream algebra in figure 1 and 2 respectively. Consider the first query.

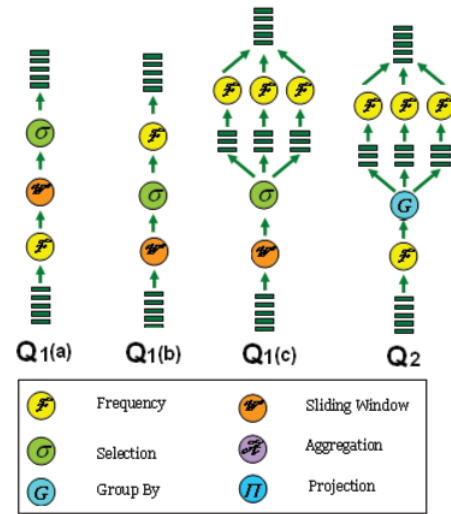


Figure 1: Stream Algebra for Example Queries 1 & 2

Query 1

“For every 100 tuples, report all horses’ body condition records.”. The query can be interpreted as for every 100 tuples over the base stream:

$Q_1(a)$:

```
SELECT *
FROM   BodyCondition AS B
WHERE  [Range 1 Minute, Frequency 100 Tuples]
        B.Species = 'horse'
```

It can also be understood as for every 100 tuples over an answer stream (a stream full of animal’s pulse rate per minute).

$Q_1(b)$:

```

SELECT * [Frequency 100 Tuples]
FROM BodyCondition AS B [Range 1 Minute]
WHERE B.Species = 'horse'

```

Thirdly it can be understood as for every 100 tuples over a substream w.r.t to each animal.

$Q_1(c)$:

```

SELECT *
      [Frequency 100 Tuples Partitioned By B.Id ]
FROM BodyCondition B [Range 1 Minute]
WHERE B.Species = 'horse'

```

This shows how CSQL supports different meanings of frequency. Figure 1 and 2 give more intuitive interpretation for these three queries using stream algebra.

Query 2

Report the latest results of measurement on blood pressure and body temperature for each animal at the rate of one reading every minute, and evaluate the query for every 100 arriving tuples.

```

SELECT * [Frequency 1 Minute Partitioned By B.Id]
FROM BodyCondition B
      [Frequency 100 Tuples Partitioned By B.Id]

```

This query will take all the last 100 tuples, and then group the stream into substreams with equality of grouping attribute “Id”. Finally it returns the relation that contains all the latest results from each substream within a minute.

Query 3

“For each animal, what is the pulse rate per minute?”, and supposing the user wants to know the result for every 10 tuples.

```

SELECT P.Id, COUNT(*)
      [Frequency 10 Tuples Partitioned By P.Id]
FROM Pulse P
      [Range 1 Minute, Frequency 1 Tuple]
GROUP BY P.Id

```

This mixed jumping window query will evaluate the query with sliding semantics $\text{Frequency} = 1 \text{ Tuple}$. It will count the last minute’s worth of *Pulse* tuples for each animal after receiving every incoming tuple. It can be easily optimised by using the default jumping semantics $\text{Range} = \text{Frequency}$ without losing the precision of the result. Then the frequency operator will evaluate the relational query over the window, at the end of the nap period. The frequency operator sitting in the *SELECT* clause will sample the resulting substream for each animal picking one tuple out of every ten tuples.

Query 4

“For each animal, what is the average pulse rate per hour?”, and supposing that the user wants to make use of the answers to the first query and expects a result every minute.

```

SELECT PR.Id, AVG(PR.Rate)
      [Frequency 1 Minute Partitioned By PR.Id]
FROM (SELECT P.Id, COUNT(*)
      [Frequency 10 Tuples Partitioned By P.Id]
      AS Rate

```

```

FROM Pulse P [Range 1 Minute,
              Frequency 1 Tuple]
GROUP BY P.Id )
AS PulseRate PR
 [Range 1 Hour, Frequency 1 Tuple]
GROUP BY PR.Id

```

This query contains two nested frequencies, but only the outer query determines how often the inner query is evaluated. The inner query or a similar query can also be registered with a frequency that is an integer fraction of 10 tuples as an independent view, then this existing inner query can be used to answer a new query.

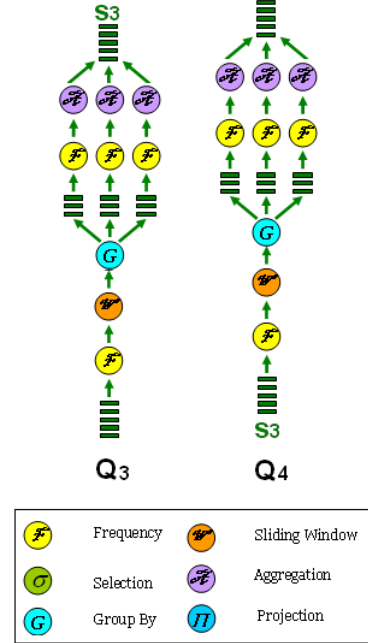


Figure 2: Stream Algebra for Example Queries 3 & 4

9 Frequency Algorithms

Some aggregation algorithms are provided from the implementation for the CSQL language below. The algorithms are categorised based on the frequency declaration. The main optimisation in the algorithm is to find the right window for each tuple (one tuple can belong to more than one window). As the window can be constructed incrementally within a sequence model, any time-based declaration (window or frequency) may lead to the next tuple *jumping* over some empty windows. A variable *jump* is therefore defined to calculate the starting bound of a target window. Time-based declarations also require two pointers to mark the current insertion and deletion points. Based on different updating requirements (lazy or eager), the old tuple can be either deleted whenever the new tuple is inserted or the old tuple deleted when the timestamp changes. Note that a different buffer is used for different window, e.g., a circular buffer with fixed size w_n when the window length is in sequence number and a linked list when the window length is in the time range w_t .

Algorithm 1 Non-grouping Aggregation with Time-based Frequency

Require: frequency length is in time range, non-grouping aggregate query

Input: tuple, query starting time q , current loop index i , frequency length f_t , window length w_t

Output: answer over windowed stream W_s

if w_t is in time range & $w_t \leq f_t$ **then**

$jump_{to} = f_t * i - w_t + q + 1$;
 { $jump_{to}$: sliding window bound}

while $t_\tau \geq jump_{to} + w_t$ **do**

$i++$;
 $jump_{to} = f_t * i - w_t + q + 1$;
 { t_τ : tuple's timestamp}

end while

while $t_\tau < jump_{to}$ **do**

 discard current tuple;
 get next tuple;

end while

while $t_\tau < jump_{to} + w_t$ **do**

 insert tuple into the buffer;
 get next tuple;

end while

 perform moving aggregation over the buffer;
 $i++$;

else if w_t is in time range & $w_t > f_t$ **then**

while $t_\tau > f_t * i + q$ **do**

$i++$;
 $jump_{to} = f_t * i - w_t + q + 1$;

end while

while $t_\tau \leq f_t * i + q$ & $t_\tau \geq f_t * i - w_t + q$ **do**

 insert tuple into the buffer;
 $\tau_O = \tau_{head}$;
 { τ_{head} : timestamp of first tuple in buffer}
 $\tau_N = \tau_{tail}$;
 { τ_{tail} : timestamp of last tuple in buffer}
 get next tuple;

end while

 perform aggregation over the buffer;
 update the buffer by condition: $\tau_N - \tau_O \leq w_t$;
 $i++$;

end if

Algorithm 2 Non-grouping Aggregation with Mixed-jumping Window

Require: frequency length is in time range, non-grouping aggregate query, w_t is in sequence number

Input: tuple, query starting time q , current loop index i , frequency length f_t , window length w_t

Output: answer over windowed stream W_s

while $t_\tau > f_t * i + q$ **do**

$i++$;

end while

while $t_\tau \leq f_t * i + q$ **do**

 insert tuple into the buffer;
 get next tuple;

end while

perform moving aggregation over the buffer;
 $i++$;

Algorithm 3 Aggregation with Partitioned Tuple-based Frequency

Require: partitioned tuple-based frequency

Input: tuple, query starting time q , current inner loop index i for each group, frequency length f_n , window length w_n or w_t

Output: answer over windowed stream W

if window length is in sequence number w_n & $w_n \leq f_n$ **then**

$jump_{to} = f_n * i - w_n + 1$;

while $t_n \leq jump_{to}$ **do**

 discard current tuple;

end while

while $t_n \leq f_n * i$ **do**

 insert tuple into the buffer;
 get next tuple;

end while

 perform aggregation over the buffer;
 $i++$;

else if window length is in sequence number w_n & $w_n > f_n$ **then**

while $t_n \leq f_n * i$ **do**

 insert tuple into the buffer;
 get next tuple;

end while

 perform aggregation over the buffer;
 $i++$;

else if window length is in time range w_t **then**

while $t_n \leq f_n * i$ **do**

 insert tuple into the buffer;

$\tau_O = \tau_{head}$;

$\tau_N = \tau_{tail}$;

 update the buffer by condition: $\tau_N - \tau_O \leq w_t$;
 get next tuple;

end while

 perform aggregation over the buffer;
 $i++$;

end if

10 Experimental Study

To evaluate the effectiveness of proposed semantics, the conceptual operators were implemented in a prototype query engine and a preliminary experimental study is conducted. The framework was implemented in Java and experiments were executed on a Pentium IV 2.4Ghz with 512M of RAM. Wall clock timings are used and execution time is calculated by measuring the average cost of 10000 answer tuples. Streaming behavior was simulated by using a pull-based execution model: the more effective the algorithm, the more tuples it is able to process. Since a frequency operator typically spends its time sampling and aggregating, there is a clear division of work. The interest is in showing how it is possible to optimize the sampling cost in such an environment, as the aim is to treat the efficiency of the aggregation algorithm as an orthogonal issue. Therefore, the same aggregation was used in all experiments, and the execution time is calculated as the sum of scanning the input stream and producing the aggregate. As a result, any performance gain observed will be due to the efficiency of the sampling methodology, which is directly tied to how well the semantics of the operators can be implemented.

Experiments are divided into two parts. Firstly, the performance of the pushed down frequency operators are evaluated in contrast to the window identifier approach for a tuple-based jumping window (AVG) query. Note that one tuple may be in the contents of multiple windows.

The efficiency of evaluating queries without or with a GROUP BY-clause is shown in figures 3 and 4 re-

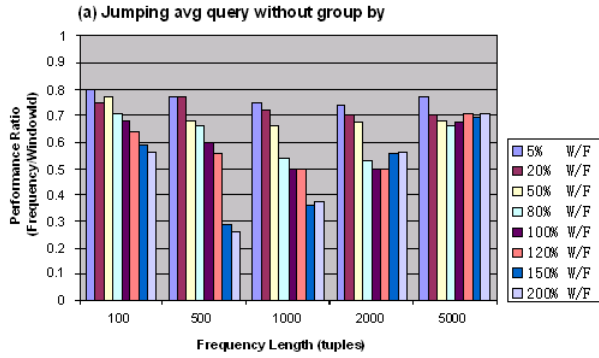


Figure 3: Cost Ratio of Frequency vs. Window Id Approach (a).

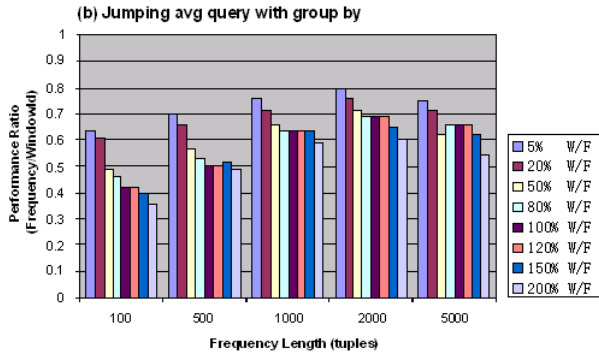


Figure 4: Cost Ratio of Frequency vs. Window Id Approach (b).

spectively. The horizontal axis is the frequency length measured in tuples. The vertical axis is the execution time using a pushed down frequency operator expressed as a fraction of the execution time of the window identifier approach. The window length is represented as a percentage of the frequency length. For example a 30% W/F ratio for a frequency length F of 1000 tuples will evaluate the query over a window length bounded by 300 tuples. As an independent operator, the frequency operator can be easily pushed down in a query plan to avoid unnecessary computation. This allows aggregate query processing to be split in two levels: (1) tuple sampling, and (2) aggregation evaluation; this modelling provides a flexible mechanism to interact with different advanced aggregate operators. Our experiment showed that pushing down the frequency operator is an effective technique and it significantly outperforms the window identifier approach. Secondly, the efficiency of processing

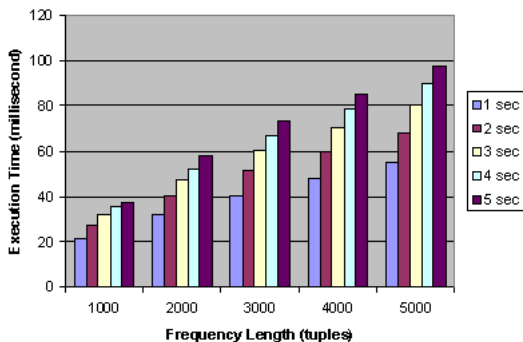


Figure 5: Performance of Mixed Jumping Window.

mixed jumping window queries was evaluated. That cannot be handled by existing approaches. Figure 5 shows the upper bound performance time of a mixed

jumping window (AVG) query that has a frequency length specified on a tuple basis and a window length specified on a time basis. The horizontal axis is the frequency length (measured in tuples) where the window length is measured in seconds. The vertical axis is the total execution time (per answer tuple appearing in the average) measured in milliseconds. Note that performance can be further improved if a more efficient aggregation algorithm is employed.

11 Conclusion

This paper has studied stream queries from a theoretical angle. More specifically, it has incorporated sampling and jumping in a declarative fashion in a query language, CSQL. Furthermore a formal semantics has been introduced on both new data model and along with a novel frequency operator for extending stream query languages with more expressibility, allowing e.g., for user-defined sampling and condensative query processing.

In the future there is the prospect of handling more complex queries, which require the automatic construction of distributed query plans, based on techniques for answering continuous queries using continuous views. A key operation in creating such plans is to determine whether a query is contained in another query. While this problem has been thoroughly investigated for queries over static databases, it is still open for continuous queries. The algebraic perspective of the model is also an interesting issue to be investigated in the future by defining extensions of relational operators to handle data stream constructs, and to study further the resulting "stream algebra" and other properties of these extensions.

Furthermore, little work considers data stream processing over different forms of stream data such as tuple-like data or token-like data. Though a data stream is commonly recognised as continuously arriving data usually at a high rate, an individual data stream item can appear in various forms such as tuple-like data (relational tuples or instances of objects) or token-like data (XML files), or numerical data (treated as a special case of tuple-like data). In relation-based models (e.g. STREAM (Babcock et al. 2002)), items are transient tuples stored in virtual relations, possibly horizontally partitioned across remote nodes. In object-based models (e.g. COUGAR (Yao & Gehrke 2003) and Tribeca (Sullivan 1996)), sources and item types are modelled as (hierarchical) data types with associated methods. Semi-structured data models for data streams have just recently been introduced (Koch, Scherzinger, Schweikardt & Stegmaier 2004, Florescu, Hillery, Kossmann, Lucas, Riccardi, Westmann, Carey & Sundararajan 2004). In an XML context, a "tuple" is often specifically referred to as a set of cells, where each cell contains a set of XML nodes (e.g. XML trees). Each data stream item is a token such as a start tag, an end tag or a PCDATA item. The work published here relates well to previous work in selectively extracting data for XML files (Fan & Ma 2006), which can lead to a promising bridge effort between querying relational data streams and XML streams.

Finally, as CSQL is a declarative stream language which is able to describe how to transform streams into smaller result streams, it will be useful for queries in large distributed applications. Such a foundation is surely key to develop a general-purpose well-understood distributed query processor for distributed data streams.

References

- Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N. & Zdonik, S. B. (2003), 'Aurora: a new model and architecture for data stream management.', *VLDB Journal* **12**(2), 120–139.
- Arasu, A., Babcock, B., Babu, S., McAlister, J. & Widom, J. (2004), 'Characterizing memory requirements for queries over continuous data streams.', *ACM Transactions on Database Systems* **29**(1), 162–194. ACM Press.
- Arasu, A. & Widom, J. (2004a), 'A denotational semantics for continuous queries over streams and relations.', *SIGMOD Record (ACM Special Interest Group on Management of Data)* **32**(2), 5–14. ACM Press.
- Arasu, A. & Widom, J. (2004b), Resource sharing in continuous sliding-window aggregates., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 336–347.
- Babcock, B., Babu, S., Datar, M., Motwani, R. & Widom, J. (2002), Models and issues in data stream systems., in 'Proceedings of ACM Symposium on Principles of Database Systems (PODS)', ACM Press, pp. 1–16.
- Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N. & Zdonik, S. B. (2002), Monitoring streams - a new class of data management applications., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 215–226.
- Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M. J., Hellerstein, J. M., Hong, W., Krishnamurthy, S., Madden, S., Raman, V., Reiss, F. & Shah, M. A. (2003), TelegraphCQ: Continuous dataflow processing for an uncertain world., in 'Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)', ACM Press, pp. 269–280.
- Chandrasekaran, S. & Franklin, M. J. (2002), Streaming queries over streaming data., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 203–214.
- Chen, J., DeWitt, D., Tian, F. & Wang, Y. (2000), NiagaraCQ: A scalable continuous query system for internet databases., in 'Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)', ACM Press, pp. 379–390.
- Cranor, C., Johnson, T., Spataschek, O. & Shkapenyuk, V. (2003), Gigascope: A stream database for network applications., in 'Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)', ACM Press, pp. 647–651.
- Dobra, A., Garofalakis, M. N., Gehrke, J. & Rastogi, R. (2002), Processing complex aggregate queries over data streams., in 'Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)', ACM Press, pp. 61–72.
- Fan, W. & Ma, L. (2006), Selectively storing XML data in relations., in 'Proceedings of International Conference on Database and Expert Systems Applications (DEXA)', Lecture Notes in Computer Science, Springer Verlag, pp. 518–526.
- Florescu, D., Hillery, C., Kossmann, D., Lucas, P., Riccardi, F., Westmann, T., Carey, M. J. & Sundararajan, A. (2004), 'The BEA streaming xquery processor.', *The VLDB Journal* **13**(3), 294–315.
- Gilbert, A. C., Kotidis, Y., Muthukrishnan, S. & Strauss, M. (2001), Surfing wavelets on streams: One-pass summaries for approximate aggregate queries., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 79–88.
- Golab, L. & Özsu, M. T. (2003), 'Issues in data stream management.', *SIGMOD Record (ACM Special Interest Group on Management of Data)* **32**(2), 5–14. ACM Press.
- Hammad, M. A., Franklin, M. J., Aref & Elmagarmid, A. K. (2003), Scheduling for shared window joins over data streams., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 297–308.
- Kießling, W. (2002), Foundations of preferences in database systems., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 311–322.
- Koch, C., Scherzinger, S., Schweikardt, N. & Stegmaier, B. (2004), FluXQuery: An optimizing XQuery processor for streaming XML data., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, pp. 1309–1312.
- Li, J., Maier, D., Tufte, K., Papadimos, V. & Tucker, P. A. (2005), Semantics and evaluation techniques for window aggregates in data streams., in 'Proceedings of ACM SIGMOD Conference on Management of Data (SIGMOD)', ACM Press, pp. 311–322.
- Ma, L. & Nutt, W. (2005), Semantics of stream operators for condensatively querying data streams., in 'IEEE International Conference on e-Business Engineering (ICEBE)', IEEE Computer Society, pp. 518–526.
- Ramakrishnan, R., Donjerkovic, D., Ranganathan, A., Beyer, K. S. & Krishnaprasad, M. (1998), SRQL: Sorted relational query language., in 'Proceedings of International Conference on Scientific and Statistical Database Management (SSDBM)', IEEE Computer Society, pp. 84–95.
- Seshadri, P., Livny, M. & Ramakrishnan, R. (1995), SEQ: A model for sequence databases., in 'Proceedings of IEEE International Conference on Data Engineering (ICDE)', IEEE Computer Society, pp. 232–239.
- Sullivan, M. (1996), Tribeca: A stream database manager for network traffic analysis., in 'Proceedings of International Conference on Very Large Databases (VLDB)', Morgan Kaufmann, p. 594.
- Yao, Y. & Gehrke, J. (2003), Query processing in sensor networks., in 'Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)', ACM Press, pp. 233–244.