Condensative Stream Query Language
for Data Streams

Lisha Ma and Hamish Taylor
School of Mathematical and Computer Sciences
Heriot-Watt University, Edinburgh, EH14 4AS, UK
Email: lm2, hamish@macs.hw.ac.uk
August 14, 2006

Abstract

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. Therefore, we call such query processing condensative. Current proposals for declarative query languages over data streams do not support such condensative processing.

In order to make existing stream query languages more expressive, we propose a new data stream model, referred to as the sequence model, and a novel stream query language CSQL (Condensative Stream Query Language). We show that the sequence model supports precise tuple-based semantics that is lack in previous time-based model. Moreover CSQL 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 the existing stream algebra with a new operator, frequency operator, to capture the condensative property. We show that a condensative stream algebra enables the generation of efficient continuous query plans, and can be used to validate query optimisation.

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

1 Introduction

A data source which is constantly publishing readings does so in the format of stream data. These can be found in many areas such as fanatical
applications, network monitoring, sensor network, etc. Different from traditional data sets presented by a DBMS, stream data arrive in multiple, rapid and constantly changing way. This creates more difficulties in managing such data by using conventional DBMS. Data streams have received increased attention in database research. Recently more challenging requirements are generated for the data stream field with the development of interaction among different academic fields and the increasing demand of information sharing, which consequentially lead to many related research problems in semantics, query processing, runtime management, etc. One major challenge is the development of techniques for providing continuously updating answers to aggregate queries over potentially unbounded streams. A general approach for addressing this challenge is by means of windows queries, which add window clauses to continuous queries and thus allow aggregate queries to be evaluated over a segment of the input data stream rather than over the entire stream. There has been a great deal on research in developing algorithms for windowed aggregate queries (Arasu & Widom 2004b, Chandrasekaran & Franklin 2002, Cranor, Johnson, Spataschek & Shkapenyuk 2003, Dobra, Garofalakis, Gehrke & Rastogi 2002a, Dobra et al. 2002a, Gilbert, Kotidis, Muthukrishnan & Strauss 2001, Li, Maier, K.Tufte, Papadimos & Tucker 2005). Driven by different purposes, a number of Distributed Stream Management Systems (DSMSs) have been developed (Babcock, Babu, Datar, Motwani & Widom 2002a, Abadi, Carney, Çetintemel, Cheriack, 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, Reiss & Shah 2003) and several stream query languages (Arasu, Babcock, Babu, McAlister & Widom 2004, Dobra, Garofalakis, Gehrke & Rastogi 2002b, Hammad, Franklin, Aref & Elmagarmid 2003, Arasu & Widom 2004a) have been recently proposed.

However, current techniques are limited in two crucial aspects. Firstly, most query languages over streams do not have the necessary constructs to support condensative query processing over streams, as is the case in traditional DBMSs. We call aggregate query processing in a traditional DBMS condensative, since the answer to an aggregate query is usually much smaller than the answer to its non-aggregate counterpart. Having a declarative stream language that supports condensative query processing is crucial since data streams are potentially unbounded, while main memory and secondary storage have fixed limits; new query results should therefore be generated whenever the window contents change. Secondly, the focus of previous work has mainly been on query evaluation while fundamental questions in con-
connection with data models and formal semantics for queries have not yet been thoroughly addressed. This lack makes it difficult to reason about aggregate queries and compare different languages in a uniform semantics. The situation is aggravated when one moves to the realm of distributed computation, as is usually the case when dealing with data streams.

**Prior Work.** CQL (continuous Query Language) (Arasu & Widom 2004a) is one of the most powerful relation-based language that is used in the STREAM system, which is proposed with full semantics over 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 uniquely identify 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 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 (Babcock, Datar & Motwani 2004, Gehrke, Korn & Srivastava...
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 we term condensative query evaluation: the user should be able to merely declare how frequently sampling should take place and the system should perform aggregation based on the user’s instruction.

In this paper, we introduce the semantics of the CSQL language as well as a novel condensative stream algebra. We illustrate the expressive ability of our language by some example queries and show how it allows us to overcome some limitations of existing query languages. 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. More specifically, our main contributions are as follows.

1. We present a formal semantics that models time-varying stream data as a function over an ordered sequence domain; this is called a sequence model. Our semantics draws features from existing work on sequence databases (Seshadri, Livny & Ramakrishnan 1995) and the time-based approach of modeling stream data (Arasu & Widom 2004a), supports a precise semantics for a tuple-based operators that can not be captured by existing time series model.

2. We incorporate sampling and jumping windows in a declarative fashion into a novel condensative stream query language CSQL. We show that our query language can specify window queries found in (Arasu & Widom 2004a, Carney, Çetintemel, Cherniack, Convey, Lee, Seidman, Stonebraker, Tatbul & Zdonik 2002, Chandrasekaran et al. 2003, Chen et al. 2000, Yao & Gehrke. 2003); also it provides functionality to support (a) a declarable way to express frequency need either on base stream or derived stream (b) a mixed jumping window over streams (c) nested aggregation. None of them can be realised in existing steam query languages.

3. We introduce a condensative stream algebra by extending the existing stream algebra with a new kind of operator, called the frequency operator as well as its concrete semantics. We also introduce the relevant
optimisation approach for our condensative model such as splitting, inter-leaving and compositive. Furthermore as an independent operator, the frequency operator can be easily pushed down in a stream algebra to avoid unnecessary computation. This allows us to split aggregate query processing technique into two levels, namely, tuple sampling and aggregation evaluation, which provides a flexible mechanism to interact with different advanced aggregate operators.

Finally, based on these we have implemented our conceptual operators in a prototype query engine. In order to demonstrate the efficiency gained by pushed down frequency operator for a jumping window query over an ordered sequence stream, we compare it with the window Id approach presented in (Li et al. 2005). Our experimental result shows that a pushed down frequency operator is effective and it outperforms window Id approach, besides a condensative stream algebra can be reasonably optimised for continuous queries with a frequency-based equivalence. We also show 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 formal semantics of our language. Language syntax is presented in section 6. Example queries are given in section 7 and 8. The main algorithms for CSQL are given in section 9, while experimental results is presented in section 10. Section 11 discusses related work and section 12 concludes the paper.

2 Background

In this section we briefly introduce stream data and semantics of continuous queries, besides we review two previous data models, from which our model draws some features.

2.1 Stream Data

Recently, a new class of applications on stream data (Chandrasekaran et al. 2003, Cranor et al. 2003, Dobra et al. 2002a) have been addressed in the database community. Mobile-phone call records, stock ticker data, web usage logs, and network monitoring data are all examples of stream data. By its nature, stream data changes constantly and quickly during the query
execution period, and normally it is impossible or quite expensive to store it, and to operate on it many times. It is different from traditional data sets that are stored in a database, which can be queried several times without any change or with small updates. Due to the rapid and continuous change of stream data, it is often not adequate to handle continuous queries with traditional DBMSs (Database Management Systems). Requirements of new technologies for stream processing arise such as stream query languages, windowed queries, approximative answers, load shedding, stream operator scheduling, stream mining, etc. (Arasu & Widom 2004, Babcock et al. 2004).

2.2 Semantics of Continuous Queries

The first continuous queries were introduced in the Tapestry system (Terry, Goldberg, Nichols & Oki 1992) with a SQL-based language called TQL. The Tapestry system allows users to pose a permanent query over a database of e-mails and bulletin board messages without joins and time, and returns the relevant answers continuously whenever a new incoming tuple passes the filter. The continuous query semantics is implemented by executing a one-time query periodically. The frequency for query answering pushes the information constantly, for example answering the query once every minute. The basic algorithm is shown below.

```
FOREVER DO
    Execute Query Q
    RETURN results to user
    Sleep for some period of time
ENDLOOP
```

Compared with conventional one-time queries that are evaluated once over static data sets stored by a DBMS, continuous queries are evaluated continuously. Answers are also streamed continuously, and do not stop until input ceases.

2.3 Sequence Database Model

The SEQ sequence model and algebra were introduced by Seshadri et al. in (Seshadri et al. 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.

2.4 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 $D_r$ be the set of all tuples that satisfy the schema $r$. Let $T$ be the set of all timestamps. Then a stream $s$ with schema $r$ is a subset $s \subseteq D_r \times T$, such that for every $\tau \in T$ the bag $\{ e \mid \langle e, \tau \rangle \in s \}$ is finite. With $P(D_r)$ we denote the set of all subsets of $D_r$. A time-dependent relation $R$ for the schema $r$ is a mapping:

$$R : T \rightarrow P(D_r),$$

such that each set $R(\tau)$ is finite. With these definitions, we can transform a stream 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, we drew features from sequence databases and decided to construct a sequence dependent model for streams. In this section we introduce our model as well as a formal semantics. As will be seen in the next two sections, our model is more expressive than the existing time-based model in (Arasu & Widom 2004a). We conclude by describing the common properties that need to be shared by different data stream models, and that we want to encode...
into our new model. We consider that an abstract relational stream should 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:

$$r(a_1 : T_1, \ldots, a_k : T_k),$$

where $r$ is a relation symbol, $a_1, \ldots, a_k$ are attributes, and $T_1, \ldots, T_k$ are types as in SQL. 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 a 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

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

A binary relation is

- **Linear:** if for all $x, y \in X$ where $x \neq y$ either $x R' y$ or $y R' x$
- **Dense:** if for all $x, y \in X$ where $x R' y$, there is a $z \in X$ such that $x R' z$ and $z R' y$
- **Discrete:** if for every two elements $x, y \in X$ where $x R' y$, there are only finitely many elements $z$ between them, i.e, there are only finitely many $z$ such that $x R' z$ and $z R' y$
If a linear ordering $R'$ is discrete, then for every element $x \in X$, either at least one element $y$ is such that $x R' y$, or there is no element $y$ such that $x R' y$. We require that a time ordering should 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. This allows us also to define the length of a sliding window. A window of length $n$ consists of the starting point plus the next $(n - 1)$ elements. If it is decided that a 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 $\Omega$ that takes a stream $s$ as input and outputs a stream $\Omega s$. We categorise the stream operators that apply to a stream as local or non-local. Suppose there is an operator $Q$ that transfers a data stream $s$ from a sequence model to a time based model $Qs$, and $Qs(t)$ represents a bag of the tuples that have timestamp $t$.

**Definition 1** $\Omega$ is local if:

$$Qs_1(t) = Qs_2(t) \text{ implies } (\Omega(Qs_1))(t) = (\Omega(Qs_2))(t),$$

otherwise it is non-local.

Most relational operators are local such as selection $\sigma$, and most stream operators are non-local such as sliding window operators $W$.

4 Stream Operators

We show that queries expressible in a time-based model can also be specified in our model. Furthermore, as will be seen in section 8, our model and operators are capable of expressing queries found in practice that are beyond previous models and languages. Next, we introduce some typical stream operators in our model.
4.1 Selection Operator

We first define the conditions for a selection operator. 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 \( \rho \) is a comparison like “\(<\)”, “\(\leq\)”, “\(=\)”, “\(\geq\)”, or “\(>\)”. Arbitrary conditions can be built up from atomic conditions using the boolean connectives “\(\neg\)”, “\(\lor\)”, or “\(\land\)”. Conditions are denoted by the letter \( C \).

We define for every condition \( C \) a selection operator \( \sigma_C \). Intuitively, \( \sigma_C(s) \) is the subsequence of tuples with index \( j \) of stream \( s \), where \( j \in \mathbb{N} \). We then define \( \sigma_C(s) \) for an arbitrary stream \( s \) recursively by saying what \( \sigma_C(s)(j) \) for an arbitrary number \( j \). We first define the set of indices \( I_1 \) as

\[ I_1 = \{ k \in \mathbb{N} \mid s(k) \text{ satisfies } C \}. \]

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

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

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

We will use \( W_t \) to denote a time-based window, and use \( W_n \) 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 we do not know 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. We also introduce \( s^t(k) \) to denote the timestamp for tuple \( s(k) \). More formally, we define the output stream \( W_t s \) as a sequence of sets \( W_t s(j) \) for a given \( j \) in stream \( s \). We say \( W_t s(j) \) is not defined if \( s(j) \) is not defined, otherwise we have

\[ W_t s(j) = \{ s(k) \mid s^t(k) + t \geq s^t(j) \text{ and } k \leq j \}. \]
**Tuple-based Sliding Window**

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

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

### 4.3 Frequency Operator

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

**Tuple-based Frequency Operator**

For every natural number \( n \in \mathbb{N} \) we have a tuple-based frequency operator \( F_n \), which 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 \( t \), we have a time-based frequency operator \( F_t \). Conceptually it 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 we will output the last tuple within that time slot. We say \( F_t s(j) \) 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

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

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

### 4.4 Jumping Windows

Sometimes, we want our window 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. We call such a kind of window a jumping window. Depending on how we define the frequency length, we categorise jumping windows 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 $W_S$ that are produced by the sliding window operators $W_n$ or $W_t$, we can define a tuple-based jumping window $F_n(W_S)$, which selects every $n$-th set of $W_S$ as follows:

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

**Time-based Jumping Window**

For a sequence of sets $W_S$ that are produced by a sliding window operator $W$ applied to the stream $s$, we obtain a time-based jumping window $F_t(W_S)$ by selecting a subsequence $W_S(n_j)$ ($j \in \mathbb{N}$) of $W_S(n)$. Intuitively, $F_t(W_S)(j)$ 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 $W$, we define $F_t(W_S)$ recursively by saying what it is the set $F_t(W_S)(j)$ for an arbitrary number $j$. We first define the set of indices $I_j$ as

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

If $I_j \neq \emptyset$, then we define $n_j := \min I_j$, and $F_t(W_S)(j) := W_S(n_j)$, otherwise let $F_t(W_S)(j) = \bot$.

**Time-based Jumping Operator**

For every time instance $t$, and a stream $W_S$ that is produced by applying a sliding window operator $W$ to a stream $s$, we obtain a time-based jumping window $F_t(W_S)$ by selecting a bag of tuples for every multiple of $t$ from stream $W_S$. We then define $(F_t(W_S))(\tau)$ for an arbitrary time $t$, where $t \in T$ as:

$$(F_t(W_S))(\tau) = \left\{ \begin{array}{ll} W_S(\tau) & \text{if } \tau = j \times t \text{ for some } j \in \mathbb{N} \\ \emptyset & \text{otherwise} \end{array} \right.$$
5 Condensative Stream Queries

A condensative stream query $Q$, in essence, is a traditional SPJ query (Turner & Lowden 1985) augmented with frequency predicates. Conceptually such queries have the “canonical” form of Eq. [1] in terms of relational algebra:

$$Q = \pi_c \mathcal{F}(p_1,\ldots,p_n) \mathcal{B}(c_1,\ldots,c_m)(R_1 \times \ldots \times R_h) \tag{1}$$

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

**Filtering:** a Boolean function $\sigma_{B(c_1,\ldots,c_m)}$ filters the results by the selection operator $\sigma_B$ (e.g., $B = c_1 \land c_2 \land c_3$ for example 1), and

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

Our goal is to support such condensative stream queries efficiently. Condensative stream models Boolean filtering, i.e., $\sigma_{B(c_1,\ldots,c_m)}$ 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, $B = c_1 \land c_2$ 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., $\sigma_{c_1 \land c_2}(R \times S) \equiv \sigma_{c_1} \sigma_{c_2}(R \times S) \equiv \sigma_{c_1}(R \bowtie_{c_2} S)$) 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 thus 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., $\mathcal{F}(p_1,\ldots,p_2)$. Moreover, the support can be compositive. Suppose we have a frequency function $\mathcal{F} = p_1 \land p_2 \land p_3$, 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 we have: $p_3 \mod p_2 = 0, p_3 \mod p_1 = 0, p_2 \mod p_1 = 0$, then the frequency functions are compositive (e.g., $\mathcal{F}_{p_1} \mathcal{F}_{p_2} \mathcal{F}_{p_3}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1} \mathcal{F}_{p_2} \mathcal{F}_{p_3}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1} \mathcal{F}_{p_2}(R_1 \times R_2 \times R_3) \equiv \mathcal{F}_{p_1}(R_1) \times (R_2 \times R_3)$. When queries are nested, frequency functions can be compositive 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 $p_2$, $(R_1 \times \mathcal{F}_{p_1} R_2) \times \mathcal{F}_{p_2} (R_1 \times \mathcal{F}_{p_1} R_2) \equiv (R_1 \times \mathcal{F}_{p_1} R_2) \times \mathcal{F}_{p_2} (R_1 \times \mathcal{F}_{p_1} R_2)$).
Finally we extend relational algebra’s *pushing down* optimisation into a stream algebra. We category the operators in a stream algebra by local and non-local semantics defined in definition [1] as such semantics assists our 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 we have a time based frequency function $F$, then we have $F(R_1 \times R_2 \times R_3) \equiv F(R_1) \times F(R_2) \times F(R_3)$.

### 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 *

```plaintext
<Query> → <Select><From> | <Select><From><Where> | <Select><From><Where><GroupBy>
<Name> → string | <Name>.<Name> | asterisk
<Attribute List> → <Name> | <Name>(<Name>)*
<Granularity> → Milliseconds|Seconds|Minutes|Hours|Tuples
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>)
<Select> → SELECT <SelectTerm> | <Select><Frequency>
<SelectTerm> → <Aggregation> | <Attribute List> | <SelectTerm>(<SelectTerm>)*
<From> → FROM <FromTerm>
<LeftBracket> → ( <RightBracket> → )
<FromTerm> → <Name> | <Name><Frequency> |
```


7 Example Scenario and Queries

Consider a tracing system to study the behaviour 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

\[
\text{Pulse}(\text{Id}, \text{Timestamp})
\]

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

\[
\text{BodyCondition}(\text{Id}, \text{Species}, \text{BTemp}, \text{BloodP}, \text{Timestamp})
\]

In these two relations: \text{Id} is the unique number of each animal, \text{Species} represents the type of animal, \text{Timestamp} represents the timestamp, \text{BloodP} is the blood pressure of the animal, and \text{BTemp} is the animal’s temperature. For ease of presentation, we assume 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 10 minutes, and evaluate the query for every 100 arriving tuples.

3. **Aggregate query:** For each animal, what is the pulse rate per minute? We suppose 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? We suppose 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**

We introduce a declarative language *CSQL* for continuous queries, similar to SQL but extended with operators such as those discussed in Section 4, as well as a mechanism for directly submitting plans in the query algebra that underlies our language.

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

We can combine the frequency operator with sliding window operators when we want our window to move much faster. We call such a kind of window a *jumping window*. When $\text{Frequency} = 1 \text{ Tuple}$, it is equivalent to a normal sliding window. Depending on how the frequency length is defined, we distinguish between *tuple-based* and *time-based* jumping windows. 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 $[\text{Range } W, \text{Frequency } F]$. 
Our 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, we extend relational algebra (Kießling 2002) into stream algebra (Babcock, Babu, Datar, Motwani & Widom 2002b) by substituting relations for streams, where the operators, and algebraic laws “respect” and take advantage of the “compositive” property introduced in section 5. We extend such a stream algebra by adding the new operator frequency operator $F$ and the sliding window operator $W$, and so generalise the existing relational operators (e.g., $\pi, \mathcal{A}, \sigma, \mathcal{G}$ in figure 1) to be “frequency-based”. We show how declarative CSQL is by expressing the frequency in different places within a query. We also show 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:

$Q_1(a)$:

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

It also can 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'
```

Finally 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'
```

From the above comparison, we show how CSQL supports different meanings of frequency. Figure ?? 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 Frequency = 1 Tuple. It will count the last minute’s worth of Pulse tuples for each animal after every coming tuples. It can be easily optimised by using the default jumping semantics Range = 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 result substream for each animal with 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. We can also register the inner query or a similar query with a frequency that is an integer fraction of 10 tuples as an independent view, then we can use this existing inner query
to answer a new query.

Figure 2: Stream Algebra for Example Queries 3 & 4

9 Frequency Algorithms

We provide some aggregation algorithms from the implementation for the CSQL language below. The algorithms are categorised based on the frequency declaration. The main optimisation approach in the algorithm is to find the right window for each tuple (one tuple can belong to more than one window). As we construct the window 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 jumpto 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 point. Based on different updating requirements (lazy or eager), we can either delete the old tuple whenever we insert the new tuple or delete the old tuple when the timestamp changes. To note that we use different buffer for different window, e.g., a circular buffer with fixed size $w_n$ when window length is in sequence number and a link list when window length is in 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 \( i \)

Input: frequency length \( f_t \), window length \( w_t \)

loop

if \( w_t \) is in time range & \( w_t \leq f_t \) then

\( \text{jump} = f_t \times i - w_t + q + 1 \) \( \{ \text{jump: sliding window bound} \} \)

\( \text{while } t_\tau \geq \text{jump} + w_t \) do

\( i++ \) \( \{ t_\tau: \text{tuple's timestamp} \} \)

\( \text{jump} = f_t \times i - w_t + q + 1 \)

end while

\( \text{while } t_\tau < \text{jump} \) do

discard current tuple, get next tuple

end while

\( \text{while } t_\tau < \text{jump} + w_t \) 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

\( \text{while } t_\tau > f_t \times i + q \) do

\( i++ \)

\( \text{jump} = f_t \times i - w_t + q + 1 \)

end while

\( \text{while } t_\tau \leq f_t \times i + q \& t_\tau \geq f_t \times i - w_t + q \) do

insert tuple into the buffer

\( \tau_O = \tau_{\text{head}} \)

\( \{ \tau_{\text{head}}: \text{timestamp of the first tuple in the buffer} \} \)

\( \tau_N = \tau_{\text{tail}} \)

\( \{ \tau_{\text{tail}}: \text{timestamp of the last tuple in the buffer} \} \)

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

end loop
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 \( i \)
Input: frequency length \( f_t \), window length \( w_t \)

loop
  while \( t_r > f_t \cdot i + q \) do
    i++
  end while
  while \( t_r \leq f_t \cdot i + q \) do
    insert tuple into the buffer
    get next tuple
  end while
  perform moving aggregation over the buffer
  i++
end loop

10 Experimental Study

To evaluate the effectiveness of our semantics, we implemented our conceptual operators in a prototype query engine and conducted a preliminary experimental study. Our framework was implemented in Java and our experiments were executed on a Pentium IV 2.4Ghz with 512M of physical memory. We report wall clock timings and calculate execution time 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. A frequency operator typically spends its time sampling and aggregating, so there is a clear division of work. We are interested in showing how it is possible to optimize the sampling cost in such an environment, as we want to treat the efficiency of the aggregation algorithm as an orthogonal issue. Therefore, we used the same aggregation in all experiments, and have calculated the execution time as the sum of scanning the input stream and producing the aggregate. As a result, any performance gain we observe 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, we evaluate the performance of the pushed down frequency operators in contrast to the window identifier approach for a tuple-based jumping window (AVG) query. Note
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

Input: frequency length \( f_n \), window length \( w_n \) or \( w_t \)

loop

if window length is in sequence number \( w_n \) & \( w_n \leq f_n \) then
  sliding window bound \( \text{jump to } = f_n \times i - w_n + 1 \)
  while \( t_n \leq \text{jump to} \) do
    discard current tuple
  end while
  while \( t_n \leq f_n \times i \) 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 \( w_n \) & \( w_n \geq f_n \) then
  while \( t_n \leq f_n \times i \) 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 \( t_n \leq f_n \times i \) do
    insert tuple into the buffer
    \( \tau_O = \tau_{\text{head}} \)
    \( \tau_N = \tau_{\text{tail}} \)
    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

end loop
that we consider the case where one tuple may be in the contents of multiple windows.

The efficiency of evaluating queries without or with a \textit{GROUP BY}-clause is shown in Figures 3 and 4 respectively. The horizontal axis is the frequency length measured in tuples. The vertical axis is the performance ratio between the execution time using a pushed down frequency operator, over 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 us to split aggregate query processing in two levels: (i) tuple sampling, and (ii) aggregation evaluation; this modeling 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, we evaluate the efficiency of processing mixed jumping window queries, which 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) which the window length is mea-

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

Aggregation operators such as SUM, COUNT, MIN, MAX, and AVG are considered as blocking operators. Since continuous data streams may be infinite, the incorporation of blocking operators into a stream algebra poses problems. Windows queries accomplish the traditional blocking operator (aggregation) in an incremental form, and restricts it to an operator over a window. There has also been a good deal of research undertaken on studies of stream algorithms for different problems; examples include the cases where a window operator does not exist or is inefficient to evaluate with limited memory requirements such as counting, hashing, sampling, summaries, sketches, wavelets (Babcock et al. 2002b, Babcock et al. 2004, Gehrke et al. 2001, Gilbert et al. 2001, Manjhi, Nath & Gibbons 2005). However, the strong incentive behind traditional aggregation computation has not only given rise to research to support aggregation over streams, but also to how to deal with them effectively. We therefore take another important property of traditional aggregation processing into account, which is called

Figure 4: Cost Ratio of Frequency vs. Window Id Approach (b).
condensative in our paper. Such semantics will lead innovation on both
stream query language and query processing. We note that we also want
to realise user-level sampling rather than to imply a system-level trade-off
between accuracy and the amount of memory. Previous approaches have
been focusing on supporting aggregation over stream or optimising aggregate
evaluation.

12 Conclusion

We have studied stream queries from a theoretical angle. More specifi-
cally, we have incorporated sampling and jumping in a declarative fashion
to a query language, CSQL. We have also introduced a formal semantics
on both new data model and novel stream operators and proposed the fre-
cquency frequency operator for extending stream query languages with more
expressibility, allowing e.g., for user-defied sampling and condensative query
processing. Our new frequency operator can be combined with the existing
sliding window operators, which yields powerful jumping window operators,
that allow even for mixed windows that can not be handles by existing query
languages. Furthermore we have developed a simple yet effective optimization
technique to implement the semantics.

During the last three years, our group has participated in the R-GMA
project, which has developed a novel type of Grid information system (Ma,
Viglas, Li & Li 2005, Cooke, Gray, Ma, Nutt, Magowan, Oevers, Taylor,
Byrom, Field, Hicks, Leake, Soni, Wilson, Cordenonsi, Cornwall, Djaoui, Fisher, Podhorszki, Coghlan, Kenny & O’Callaghan 2003). Among the issues arising from this cooperation was one about devising an approach for integrating monitoring data that comes in a stream format. To date R-GMA is able to handle simple continuous selection queries within a view-based architecture. As on-going work, we are looking into ways to handle 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. From a purely theoretical perspective, perhaps the most interesting open question is that of defining extensions of relational operators to handle data stream constructs, and to further study the resulting ”stream algebra” and other properties of these extensions.

As CSQL 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 developing a general-purpose well-understood distributed query processor for distributed data streams.

References


