An Abstract Semantics and Concrete Language for Continuous Queries over Streams and Relations

Arvind Arasu Shivnath Babu Jennifer Widom

Stanford University
{arvinda,shivnath,widom}@cs.stanford.edu

Abstract

Despite the recent surge of research in query processing over data streams, little attention has been devoted to defining precise semantics for continuous queries over streams. We first present an abstract semantics based on several building blocks: formal definitions for streams and relations, mappings among them, and any relational query language. From these basics we define a precise interpretation for continuous queries over streams and relations.

We then propose a concrete language, *CQL* (for *Continuous Query Language*), which instantiates the abstract semantics using SQL as the relational query language and window specifications derived from SQL-99 to map from streams to relations. We identify some equivalences that can be used to rewrite CQL queries for optimization, and we discuss some additional implementation issues arising from the language and its semantics.

We have implemented a substantial fraction of CQL in a Data Stream Management System at Stanford, and we have developed a public repository of data stream applications that includes a wide variety of queries expressed in CQL.

1 Introduction

There has been a considerable surge of interest recently in query processing over *unbounded data streams*, e.g., [CCC+02, CF02, DGGR02, MF02, MW+03, VN02]. Because of the continuously arriving nature of the data, queries over data streams tend to be *continuous* [BW01, CDTW00, MSHR02, LPT99], rather than the traditional *one-time* queries over stored data sets. Many papers have included example continuous queries over data streams, expressed in some declarative language, e.g., [ABB+02, CF02, CDTW00, MSHR02]. However, these queries tend to be for illustrative purposes, and for complex queries the precise semantics may be left unclear. To the best of our knowledge no prior work has provided an exact semantics for general-purpose declarative continuous queries over streams and relations.

It may appear initially that the problem is not a difficult one: We take a relational query language, replace references to relations with references to streams, register the query with the stream processor, and wait for answers to arrive. For simple monotonic queries over complete stream histories indeed this approach is nearly sufficient. However, as queries get more complex—when we add aggregation, subqueries, windowing constructs, relations mixed with streams, etc.—the situation becomes much murkier. Even a simple query such as:

```
Select * From S[Rows 5], R
Where S.A = R.B
```

where S is a stream, R is a relation, and `[Rows 5]` specifies a *sliding window*, has no single obvious interpretation that we know of.

In this paper we initially define an *abstract semantics* based on "black box" components—any relational query language, any window specification language, and a set of relation-to-stream operators. We then define a *concrete language* that instantiates the black boxes in our abstract semantics, and that we are in the process of implementing in the *STREAM* prototype (*STanford stREeam datA Manager*), a general-purpose Data Stream Management System (DSMS) being developed at Stanford [MW+03]. Our concrete language also has been used to specify a wide variety of continuous queries in a public repository of data stream applications we are curating [SQR].

In defining our abstract semantics and concrete language we had several goals in mind:

1. We wanted to exploit well-understood relational semantics to the extent possible.
2. We wanted queries performing simple tasks to be easy and compact to write. Conversely, we wanted simple-appearing queries to do what one expects.
3. Since transformations are crucial for query optimization, we did not want to inhibit standard relational transformations with our new constructs and operators, or with our semantic interpretation. Furthermore, we wanted to enable new transformations specific to streams.

We believe these goals have been achieved to a large extent. To summarize the contributions of this paper:

- We formalize streams, updateable relations, and their interrelationship (Section 4).
- We define an abstract semantics for continuous queries constructed from three building blocks: any relational query language to operate on relations, any window specification language to convert streams to relations, and a set of three operators that convert relations to streams (Section 5).
- We propose a concrete language, *CQL* (for *Continuous Query Language*), which instantiates the abstract semantics using SQL as its relational query language and a window specification language derived from SQL-99. We define syntactic shortcuts and defaults in CQL for convenient and intuitive query formulation, and compare expressiveness against related query languages (Section 6).
- We briefly consider two issues that arise when supporting CQL in a DSMS: exploiting CQL equivalences for query-rewrite optimization, and dealing with time-related issues such as input data streams arriving at the DSMS out of order (Section 7).

2 Related Work

A comprehensive description of work related to data streams and continuous queries is given in [BBD+02]. Here we focus on work related to languages and semantics for continuous queries.

Continuous queries have been used either explicitly or implicitly for quite some time. *Materialized views* [GM95] are a form of continuous query, since a view is continuously updated to reflect changes to its base relations. Reference [JMS95] extends materialized views to include *chronicles*, which essentially are continuous data streams. Operators are defined over chronicles and relations to produce other chronicles, and also to transform chronicles to materialized views. The operators are constrained to ensure that the resulting materialized views can be maintained incrementally without referencing entire chronicle histories.

Continuous queries were introduced explicitly for the first time in *Tapestry* [Bar99, TGNO92] with a SQL-based language called *TQL*. Conceptually, a TQL query is executed once every time instant as a one-time SQL query over the snapshot of the database at that instant, and the results of all the one-time queries are merged using set union. Several systems use continuous queries for information dissemination, e.g., [CDTW00, LPT99, NACP01]. The semantics of continuous queries in these systems is also based on periodic execution of one-time queries as in Tapestry. In Section 6.4 we show how Tapestry queries and materialized views over relations and chronicles can be expressed in CQL.

The abstract semantics and concrete language proposed in this paper are more general than any of the languages above, incorporating window specifications, constructs for freely mixing and mapping streams and relations, and the full power of any relational query language. Recent work in the *TelegraphCQ* system [C+03] proposes a declarative language for continuous queries with a particular focus on expressive windowing constructs. The *TelegraphCQ* language is discussed again briefly in Section 7.1. The *ATLaS* [WZL02] SQL extension provides language constructs for expressing incremental computation of aggregates over windows on streams, but in the context of simple SPJ queries.

Several systems support procedural continuous queries, as opposed to the declarative approach in this paper. The *event-condition-action* rules of active database systems, closely related to SQL *triggers*, fall into this category [PD99]. The *Aurora* system [CCC+02] is based on users directly creating a network of stream operators. A large number of operator types are available, from simple stream filters to complex windowing and aggregation operators. The *Tribeca* stream-processing system for network traffic analysis [Sul96] supports windows, a set of operators adapted from relational algebra, and a simple language for composing query plans from them. Tribeca does not support joins across streams. Both Aurora and Tribeca are compared against CQL in more detail in Section 6.4.

Since stream tuples have timestamps and therefore ordering, our semantics and query language are related to *temporal* [OS95] and *sequence* [SLR95] query languages. In most respects the temporal or ordering constructs in those languages subsume the corresponding features in ours, making our language less expressive but easier to implement efficiently. Also note that the semantics of temporal and sequence languages is for one-time, not continuous, queries.

3 Running Example

We introduce a running example based on a fabricated online auction application originally proposed as part of a benchmark for data stream systems [TTPM02]. This particular running example has been selected (and purposely kept simple) not for its realism but in order to easily illustrate various aspects of our semantics and query language. The interested reader is referred to the public query repository [SQR] for more complex and realistic stream applications, including a large number and variety of queries expressed in the language proposed in this paper.

The online auction application consists of users, the transactions of the users, and continuous queries that users or administrators of the system register for various monitoring purposes. Before submitting any transaction to the system a user registers by providing a name and current state of residence. Registered users can later deregister. Three kinds of transactions are available in the system: users can place an item for auction and specify a starting price for the auction, they can close an auction they previously started, and they can bid for currently active auctions placed by other users by specifying a bid price. From the point of view of the auction system, all user registrations, deregistrations, and transactions are provided in the form of a

continuous unbounded data stream.

Users also can register various monitoring queries in the system. For example, a user might request to be notified about any auction placed by a user from California within a specified price range. The auction system itself can run continuous queries for administrative purposes, such as: (1) Whenever an auction is closed, generate an entry with the closing price of the auction based on bid history. (2) Maintain the current set of active auctions and currently highest bid for them (to support "ad-hoc" queries from potential future bidders). (3) Maintain the current top 100 "hot items," *i.e.*, 100 items with the most number of bids in the last hour.

We will formalize details of the running example as the paper progresses.

4 Streams and Relations

In this section we define a formal model of streams, relations, and mappings between them, which we will use as the basis for our abstract semantics in Section 5. (In Section 7.1 we justify our decision to have both streams and relations instead of, for example, simply streams). For now let us assume any global, discrete, ordered time domain $\mathcal{T}$, such as the nonnegative numbers. A *time instant* (or simply *instant*) is any value from $\mathcal{T}$. We discuss time in much more detail in Section 7.3. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of attributes.

Definition 4.1 (**Stream**) A stream S is a bag of *elements* $\langle s, \tau \rangle$, where s is a tuple belonging to the schema of the stream and $\tau \in \mathcal{T}$ is the *timestamp* of the element. □

Definition 4.2 (**Relation**) A relation R is a mapping from $\mathcal{T}$ to a finite but unbounded bag of tuples, where each tuple belongs to the schema of the relation. □

A stream element $\langle s, \tau \rangle \in S$ indicates that tuple s arrives on stream S at time τ, and note that the timestamp is not part of the schema of the stream. For a given time instant $\tau \in \mathcal{T}$ there could be zero, one, or multiple elements with timestamp τ in a stream S. However we require that there be a finite (but unbounded) number of elements with a given timestamp. In addition to the source data streams that arrive at a DSMS, streams may result from queries or subqueries as described in Section 5. We use the terms *base stream* and *derived stream* to distinguish between source streams and streams resulting from queries or subqueries. For a stream S, we use the phrase *elements of S up to τ* to denote the elements of S with timestamp $\leq \tau$.

A relation R defines an unordered bag of tuples at any time instant τ, denoted $R(\tau)$. Note the difference between this definition for relation and the standard one: In the usual relational model a relation is simply a set (or bag) of tuples, with no notion of time as far as the semantics of relational query languages are concerned. We use the term *instantaneous relation* to denote relations in the traditional static bag-of-tuples sense, and *relation* to denote time-varying bag of tuples as in Definition 4.2. In addition to the stored relations in a DSMS, relations may result from queries or subqueries as described in Section 5. We use the terms *base relation* and *derived relation* to distinguish between stored relations and relations resulting from queries or subqueries.

We emphasize that the timestamp τ of a stream element $\langle s, \tau \rangle$ refers to logical time as determined by application semantics. Time domain $\mathcal{T}$ need not relate to any notion of actual clock; τ certainly need not (although could) be the physical time of arrival of the element at the system. Similarly, τ in the instantaneous relation $R(\tau)$ refers to logical time from domain $\mathcal{T}$ and not physical time. Section 7.3 discusses the relationship between stream timestamps and physical time in greater detail.

4.1 Modeling the Running Example

We now formally model the running example introduced in Section 3, drawn from [TTPM02]. The input to the online auction system consists of the following five streams:

- `Register(user_id,name,state)`: An element on this stream denotes the registration of a user identified by `user_id`. A user may register more than once in order to change his state, but we assume in our example queries that users do not change their name (i.e., there is a functional dependency `user_id` → `name`).
- `Deregister(user_id)`: An element on this stream denotes the deregistration of a user identified by `user_id`. We assume that a user deregisters at most once, i.e., `user_id` is a key for the `Deregister` stream.
- `Open(item_id,seller_id,start_price)`: An element on this stream denotes the start of an auction on `item_id` by user `seller_id` at a starting price of `start_price`. We assume that `item_id` is a key for this stream.
- `Close(item_id)`: An element on this stream denotes the closing of the auction on `item_id`, and `item_id` is again a key.
- `Bid(item_id,bidder_id,bid_price)`: An element on this stream denotes user `bidder_id` registering a bid of price `bid_price` for the open auction on `item_id`. We assume bid prices for each item strictly increase over time and are all greater than the starting price.

The time domain for the running example is *Datetime*, and the timestamp of stream elements correspond to the logical time of occurrence of registration, deregistration or a transaction. In addition to these five streams, we have a derived relation `User(user_id, name, state)` containing the currently registered users. This relation is derived from streams `Register` and `Deregister`, as will be shown in Query 6.4 of Section 6.3.

4.2 Mapping Operators

We consider three classes of operators over streams and relations: *stream-to-relation* operators, *relation-to-relation* operators, and *relation-to-stream* operators. We decided not to introduce *stream-to-stream* operators, instead requiring those operators to be composed from the other three types. One rationale for this decision is goal #1 from Section 1—exploiting relational semantics whenever possible—but other aspects of the decision are discussed in Section 7.1, after our semantics and language are formalized.

1. A *stream-to-relation operator* takes a stream S as input and produces a relation R with the same schema as S as output. At any instant τ, $R(\tau)$ should be computable from the elements of S up to τ.
2. A *relation-to-relation operator* takes one or more relations $R_1, \ldots, R_n$ as input and produces a relation R as output. At any instant τ, $R(\tau)$ should be computable from the states of the input instantaneous relations at τ, i.e., $R_1(\tau), \ldots, R_n(\tau)$.
3. A *relation-to-stream operator* takes a relation R as input and produces a stream S with the same schema as R as output. At any instant τ the elements of S with timestamp τ should be computable from $R(\tau')$ for $\tau' \leq \tau$. In fact for the three operators we introduce in Section 5.1 only $R(\tau)$ and $R(\tau - 1)$ are needed, where $\tau - 1$ generically denotes the time instant preceding τ in our discrete time domain $\mathcal{T}$.

Example 4.1 Consider the `Bid` stream from Section 4.1. Suppose we take a "sliding window" over this stream that contains the bids over the last ten minutes. This windowing operator is an example of a stream-to-relation operator: the output relation R at time τ contains all the bids in stream `Bid` with timestamp between τ and $\tau - 10$ `Minute`. (Note that the subtraction is not integer subtraction but subtraction in the *Datetime* domain.)

Now consider the operator `Avg(bid_price)` over the relation R output from the window operator. This aggregation is an example of a relation-to-relation operator: at time τ it takes instantaneous relation $R(\tau)$ as input and outputs an instantaneous relation with one single-attribute tuple containing the average price of the bids in $R(\tau)$—that is, the average price of bids in the last ten minutes on stream `Bid`. Finally, a simple relation-to-stream operator might stream the average price resulting from operator `Avg(bid_price)` every time the average price changes. □

5 Abstract Semantics

In this section we present an abstract semantics for continuous queries using our model of streams, relations, and mappings among them from Section 4. We assume that a query is constructed from the following three building blocks:

1. Any relational query language, which we can view abstractly as a set of relation-to-relation operators as defined in Section 4.

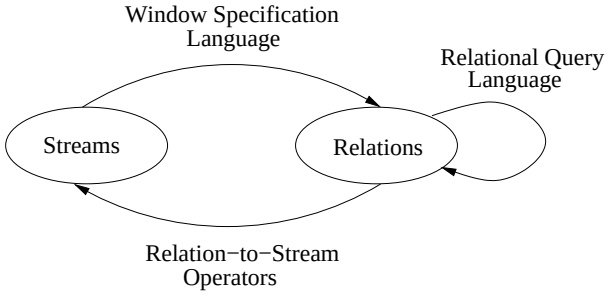

Figure 1: Mappings used in abstract semantics.

2. A *window specification language* used to extract tuples from streams, which we can view as a set of stream-to-relation operators as defined in Section 4. In theory these operators need not have anything to do with "windows," but in practice windowing is the most common way of producing bounded sets of tuples from unbounded streams [BBD+02].[1]
3. Three relation-to-stream operators: `Istream`, `Dstream`, and `Rstream`, defined shortly in Section 5.1. Here we could certainly be more abstract—any relation-to-stream language could be used—but the three operators we define capture the required expressiveness for all queries we have considered [SQR].

The interaction among these three building blocks is depicted in Figure 1. With these building blocks, our abstract semantics is straightforward: A well-formed continuous query is simply assembled in a type-consistent way from streams, relations, and the operators in Figure 1. Specifically, consider a time instant τ. If a derived relation R is the output of a window operator over a stream S, then $R(\tau)$ is computed by applying the window semantics on the elements of S up to τ. If a derived relation R is the output of a relational query, then $R(\tau)$ is computed by applying the semantics of the relational query on the input relations at time τ. Finally, any derived stream is the output of an `Istream`, `Dstream`, or `Rstream` operator over a relation R and is computed as described next in Section 5.1. The final result of the continuous query can be a relation or a stream.

In the remainder of this section we define our three relation-to-stream operators, then provide a "concrete" example of our abstract semantics. Further examples are provided later in the paper in the context of our concrete language.

5.1 Relation-to-Stream Operators

As mentioned earlier, our abstract semantics could assume a "black box" language for mapping relations to streams, as we have done for streams-to-relations. However, at this

[1] Certain types of sampling [Vit85] provide another way of doing so, and do fit into our framework. However, our current plan is to implement sampling as a separate operator in our query language as outlined in [MW+03].

point we will formalize three operators that we have found to be particularly useful, and, so far, sufficient [SQR]. Note that operators $\cup$, $\times$, and $-$ below are assumed to be the bag versions.

- `Istream` (for "insert stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Using 0 to denote the earliest instant in time domain $\mathcal{T}$, formally we have (assuming $R(-1) = \phi$ for notational simplicity):

$$\texttt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$

- Analogously, `Dstream` (for "delete stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau - 1) - R(\tau)$. Formally:

$$\texttt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$

- `Rstream` (for "relation stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in R at time τ. Formally:

$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

With some basic windowing and relational constructs such as those defined for CQL in Section 6, `Rstream` subsumes the combination of `Istream` and `Dstream`. For many queries `Istream` is more natural to use than `Rstream`, and `Dstream` is a natural counterpart to `Istream`. Hence we decided to introduce all three operators in keeping with goal #3 from Section 1—simple queries should be easy and compact to write. (The astute reader may note that `Dstream` does not appear again in this paper, however it does occasionally have its uses, as seen in [SQR].)

5.2 Example

Using our abstract semantics let us revisit the example continuous query in Section 1 (expressed there as SQL), to understand its behavior in detail and to illustrate the semantics of our relation-to-stream operators. Using relational algebra as our relational query language and $S[n]$ to denote an n-tuple sliding window over stream S, the query can be rewritten as:

$$S[5] \bowtie_{S.A=R.B} R$$

At any time instant τ, $S[5]$ produces an instantaneous relation containing the last five tuples in S up to τ. That relation is joined with $R(\tau)$, producing the final query result which in this case is a relation. The result relation may change whenever a new tuple arrives in S or R is updated.

Now suppose we add an outermost `Istream` operator to this query, converting the relational result into a stream. With `Istream` semantics, a new element $\langle u, \tau \rangle$ is streamed whenever tuple u is inserted into $S[5] \bowtie R$ at time τ as the result of a stream arrival or relation update. Note however that u must not be present in the join result at time $\tau - 1$, which creates an unusual but noteworthy subtlety in `Istream` semantics: If S has no key, and it contains two identical tuples s_i and s_{i+5} that are five tuples apart and that join with a tuple $r \in R$, then the `Istream` operator will not produce tuple (s_{i+5}, r) in the result stream when s_{i+5} arrives on S since the relational output of the join is unchanged. Replacing the outermost `Istream` operator with `Rstream` will produce element $\langle (s_{i+5}, r), \tau \rangle$ in the result stream, however now the entire join result will be streamed at each instant of time.

Fortunately, anomalous queries such as the example above are unusual and primarily of theoretical interest. For example, no such queries have arisen in [SQR], and we were unable to construct a realistic concrete example for this paper.

6 Concrete Query Language

Our concrete language *CQL* (standing for *Continuous Query Language* and pronounced "C-Q-L" or "sequel," depending on taste) uses SQL as its relational query language. In theory it supports all SQL constructs, but our current implementation eliminates many of the more esoteric ones. CQL contains three syntactic extensions to SQL:

1. Anywhere a relation may be referenced in SQL, a stream may be referenced in CQL.
2. In CQL every reference to a base stream, and every subquery producing a stream, must be followed immediately by a window specification. Our window specification language is derived from SQL-99 and defined in Section 6.1. Syntactic shortcuts based on default windows are defined in Section 6.2.1.
3. In CQL any reference to a relation, or any subquery producing a relation, may be converted into a stream by applying any of the operators `Istream`, `Dstream`, or `Rstream` defined in Section 5.1 to the entire `Select` list. Default conversions are applied in certain cases; see Section 6.2.2.

After defining our window specification language in Section 6.1, we introduce CQL's syntactic shortcuts and defaults in Section 6.2. We provide detailed examples of CQL queries in Section 6.3, and in Section 6.4 we compare CQL against related query languages.

Although we do not specify them explicitly as part of our language, incorporating user-defined functions, user-defined aggregates, or user-defined window operators poses no problem in CQL, at least from the semantic perspective.

6.1 Window Specification Language

Currently CQL supports only *sliding* windows, and it supports three types: time-based, tuple-based, and partitioned. Other types of sliding windows, *fixed* windows [Sul96], *tumbling* windows [CCC+02], *value-based* windows [SLR95], or any other windowing construct can be incorporated into CQL easily—new syntax must be added, but the semantics of incorporating a new window

type relies solely on the semantics of the window operator itself, thanks to our "building-blocks" approach.

In the definitions below note that a stream S may be a base stream or a derived stream produced by a subquery.

6.1.1 Time-Based Windows

A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following S in the query with `[Range T]`.[2] Intuitively, this window defines its output relation over time by sliding an interval of size T time units over S. More formally, the output relation R of "`S[Range T]`" is defined as:

$$R(\tau) = \{s \mid \langle s, \tau' \rangle \in S \ \wedge \ (\tau' \leq \tau) \ \wedge \ (\tau' \geq \max\{\tau - T, 0\})\}$$

Two important special cases are $T = 0$ and $T = \infty$. When $T = 0$, $R(\tau)$ consists of tuples obtained from elements of S with timestamp τ. In CQL we introduce the syntax "`S[Now]`" for this special case. When $T = \infty$, $R(\tau)$ consists of tuples obtained from all elements of S up to τ and uses the SQL-99 syntax "`S[Range Unbounded]`."

6.1.2 Tuple-Based Windows

A tuple-based sliding window on a stream S takes a positive integer N as a parameter and is specified by following S in the query with `[Rows N]`. Intuitively, this window defines its output relation by sliding a window of size N tuples over S. More formally, for the output relation R of "`S[Rows N]`," $R(\tau)$ consists of tuples obtained from the N elements with the largest timestamps in S no greater than τ (or all elements if the length of S up to τ is $\leq N$). Suppose we specify a sliding window of N tuples and at some point there are several tuples with the Nth most recent timestamp (while to keep things clear we assume the other $N - 1$ more recent timestamps are unique). Then we must "break the tie" in some fashion to generate exactly N tuples in the window. We assume such ties are broken arbitrarily. Thus, tuple-based sliding windows may be nondeterministic—and therefore may not be appropriate—when timestamps are not unique. The special case of $N = \infty$ is specified by `[Rows Unbounded]`, and is equivalent to `[Range Unbounded]`.

6.1.3 Partitioned Windows

A partitioned sliding window on a stream S takes a positive integer N and a subset $\{A_1, \ldots, A_k\}$ of S's attributes as parameters. It is specified by following S in the query with `[Partition By` $A_1, \ldots, A_k$ `Rows N]`. Intuitively, this window logically partitions S into different substreams based on equality of attributes $A_1, \ldots, A_k$ (similar to SQL `Group By`), computes a tuple-based sliding window of size N independently on each substream, then takes the union of these windows to produce the output relation. More formally, a tuple s with values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$ occurs in output instantaneous relation $R(\tau)$ *iff* there exists an element $\langle s, \tau' \rangle \in S$ such that $\tau' \leq \tau$ is among the N largest timestamps among elements whose tuples have values $a_1, \ldots, a_k$ for attributes $A_1, \ldots, A_k$. Note that analogous time-based partitioned windows would provide no additional expressiveness over nonpartitioned time-based windows.

[2] In all three of our window types we dropped the keyword `Preceding` appearing in the SQL-99 syntax and in our earlier specification [MW+03]—we only have "preceding" windows for now so the keyword is superfluous. Also we do not specify a syntax, type, or restrictions for time-interval T at this point. Currently our system is restricted use the *Datetime* type for timestamps and the SQL-99 standard for time intervals. Examples are given in Section 6.3 and time is discussed further in Section 7.3.

6.2 Syntactic Shortcuts and Defaults

In keeping with goal #3 in Section 1, we permit some syntactic "shortcuts" in CQL that result in the application of certain defaults. Of course there may be cases where the default behavior is not what the author intended, so we assume that when queries are registered the system informs the author of the defaults applied and offers the opportunity to edit the expanded query. There are two classes of shortcuts: omitting window specifications (Section 6.2.1) and omitting relation-to-stream operators (Section 6.2.2).

6.2.1 Default Windows

When a base stream or a stream derived from a subquery is referenced in a CQL query and is not followed by a window specification, an `Unbounded` window is applied by default. (Recall from the beginning of this section that every reference to a stream within a query must be followed immediately by a window specification.) While the default `Unbounded` window usually produces appropriate behavior, there are cases where a `Now` window is more appropriate, e.g., when a stream is joined with a relation; see Query 6.5 in Section 6.3 for an example. Also it is important to realize that `Unbounded` windows often may be replaced by `Now` windows as a query rewrite optimization; see Section 7.2.1.

6.2.2 Default Relation-to-Stream Operators

There are two cases in which it seems natural for authors to omit an intended `Istream` operator from a CQL query:

1. On the outermost query, even when *streamed results* rather than *stored results* are desired [MW+03].
2. On an inner subquery, even though a window is specified on the subquery result.

For the first case we add an `Istream` operator by default whenever the query produces a relation that is *monotonic*. A relation R is monotonic *iff* $R(\tau_1) \subseteq R(\tau_2)$ whenever $\tau_1 \leq \tau_2$. A conservative monotonicity test can be performed statically. For example, a base relation is monotonic if it is known to be append-only, "`S[Range Unbounded]`" is monotonic for any stream S, and the join of two monotonic relations also is monotonic. If the result of a CQL query is a monotonic relation then it makes

intuitive sense to convert the relation into a stream using Istream. If it is not monotonic, the author might intend Istream, Dstream, or Rstream, so we do not add a relation-to-stream operator by default.

For the second case we add an Istream operator by default whenever the subquery is monotonic. If it is not, then the intended meaning of a window specification on the subquery result is somewhat ambiguous, so a semantic (type) error is generated, and the author must add an explicit relation-to-stream operator.

6.3 Example Queries

We present several example queries to illustrate the syntax and semantics of CQL. All queries are based on the online auction schema introduced in Section 3 and formalized in Section 4.1.

Query 6.1: Stream Filter

Select auctions where the starting price exceeds 100 and produce the result as a stream.

```
Select * From Open Where start_price > 100
```

This query relies on two CQL defaults. Since the Open stream is referenced without a window specification, an Unbounded window is applied by default. At time τ, the relational result of the unbounded window contains tuples from all elements of Open up to τ, and the output relation of the entire query contains the subset of those tuples that satisfy the filter predicate. Since the output relation is monotonic, a default Istream operator is applied, converting the output relation into a stream consisting of each element of Open that satisfies the filter predicate.

As will be discussed in Section 7.2.1, the Unbounded window in this query can be replaced by a Now window, which obviously suggests a much more efficient implementation. The final expanded and rewritten query is:

```
Select Istream(*) From Open[Now]
Where start_price > 100
```

Query 6.2: Sliding-Window Aggregate

Maintain a running count of the number of bids in the last hour on items with item_id *in the range 100 to 200.*

```
Select Count(*) From Bid[Range 1 Hour]
Where item_id >= 100 and item_id <= 200
```

The Bid stream has an explicit window specification and the result of the query is a nonmonotonic singleton relation, so no defaults are applied. If the author adds an Istream operator, then the result will instead stream a new value each time the count changes. If the count should be streamed at each time instant regardless of whether its value has changed, then an Rstream operator should be used instead of Istream.

Query 6.3: Stream Subquery

Maintain a table of the currently open auctions.

```
Select * From Open
Where item_id Not In (Select * From Close)
```

Unbounded windows are applied by default on both Open and Close. The subquery in the Where clause returns a monotonic relation containing all closed auctions at any time, but a default Istream operator is not applied since there is no window specification following the subquery. The relational result of the entire query is not monotonic—auction tuples are deleted from the result when the auction is closed—and therefore an outermost Istream operator is not applied.

Query 6.4: Derived Relation

Compute the User *relation (described in Section 4.1) containing the currently registered users.*

```
Select user_id, name, state
From Register[Partition By user_id Rows 1]
Where user_id Not In
              (Select * From Deregister)
```

The partitioned window on the Register stream obtains the latest registration for each user, from which the Where clause filters out users who have already deregistered. In subsequent examples we refer to User in our queries as a normal relation. The query above definining relation User can be substituted syntactically into examples referencing User, although in a system we might instead choose to maintain User as a materialized view.

Query 6.5: Relation-Stream Join

Output as a stream the auctions started by users who were California residents when they started the auction.

```
Select Istream(Open.*) From Open[Now], User
Where seller_id = id and state = 'CA'
```

This query joins the Open stream with the User relation. The Now window on Open ensures that a stream tuple joins with the corresponding User tuple to find the state of residence at the time the auction starts. If we used an Unbounded window on Open instead of the Now window, then whenever a user moved into California all previous auctions started by that user would be generated in the result stream.

This query is an example where if a window specification were omitted the default Unbounded window would not provide the intended behavior. In general, if a stream is joined with a relation in order to add attributes to or filter the stream, then a Now window on the stream coupled with an Istream or Rstream operator usually provides the desired behavior. However, this rule is not so hard-and-fast that it would be appropriate to identify these cases and apply a default Now window.

Query 6.6: Windowed-Stream Join

Stream the `item_id`*'s for all auctions closed within 5 hours of their opening.*

```
Select Istream(Close.item_id)
From Close[Now], Open[Range 5 Hours]
Where Close.item_id = Open.item_id
```

This query streams any `item_id` from `Close` whose corresponding `Open` tuple arrived within the last 5 hours. Note that by our abstract semantics defined in Section 5, the timestamps on result stream elements correspond to the timestamps on the `Close` stream elements from which they are produced.

Query 6.7: Complex From Clause

Compute the closing price of each auction and stream the result.

We include the possibility of auctions with no bids, and solely for purposes of illustration we stream the starting price of an auction with no bids as its closing price. Recall that we assume bid prices are strictly increasing over time.

```
Select Istream(Close.item_id, P.price)
From Close[Now],
     ((Select item_id, bid_price as price
       From Bid)
      Union
      (Select item_id, start_price as price
       From Open))
      [Partition By item_id Rows 1] As P
Where Close.item_id = P.item_id
```

`Unbounded` windows are applied by default on the `Bid` and `Open` streams, however after we take their union the partitioned window extracts the latest element for each `item_id`. An `Istream` operator is (and needs to be) applied to the `Union` result by default, since the relational output of the `Union` subquery is monotonic and is followed by a window specification.[3]

Recall that we assume bids are not permitted once an auction closes, so new tuples are produced in the join only when they are produced on stream `Close`. The new join result tuple contains the latest transaction for the closed auction—either the latest bid or the opening of the auction—from which the closing price is extracted. Finally observe that based on application semantics the author could have used an `Unbounded` window on stream `Close` instead of a `Now` window and the query result would be equivalent, so the default window is acceptable for all three streams in this query.

[3]Technically we have not specified syntax in this paper for applying `Istream` to the result of a `Union` of two subqueries. In practice we allow `Istream`, `Dstream`, or `Rstream` to be placed outside of any query producing a relation, which for SPJ queries is equivalent to applying it to the `Select` list.

Monotonicity

Queries 6.1 and 6.7 exploited monotonicity, which was relatively straightforward to detect in both cases. We anticipate that that fairly simple (conservative) monotonicity tests can be used in general, since failing to detect monotonicity when it holds does not pose a real problem: For a monotonic subquery with a window operator, the author would be notified of the type inconsistency and would need to add the `Istream` operator explicitly. Similarly, if a default `Istream` is not applied to the outermost query the author would know that the query returns a relation and can add the `Istream` operator as desired.

6.4 Comparison with Other Languages

Now that we have presented our language we can provide a more detailed comparison against some of the related languages for continuous queries over streams and relations that were discussed briefly in Section 2. Specifically, we show that basic CQL (without user-defined functions, aggregates, or window operators) is strictly more expressive than *Tapestry* [TGNO92], *Tribeca* [Sul96], and materialized views over relations with or without *chronicles* [JMS95]. We also discuss *Aurora* [CCC+02], although it is difficult to compare CQL against Aurora because of Aurora's graphical, procedural nature. TelegraphCQ [C+03] is discussed in Section 7.1.

6.4.1 Views and Chronicles

Any conventional materialized view defined using a SQL query Q can be expressed in CQL using the same query Q with CQL semantics.

The *Chronicle Data Model* (*CDM*) [JMS95] defines chronicles, relations, and persistent views, which are equivalent to streams, base relations, and derived relations in our terminology. For consistency we use our terminology instead of theirs. CDM supports two classes of operators based on relational algebra, both of which can be expressed in CQL. The first class takes streams and (optionally) base relations as input and and produces streams as output. Each operator in this class can be expressed equivalently in CQL by applying a `Now` window on the input streams, translating the relational algebra operator to SQL, and applying an `Rstream` operator to produce a streamed result. For example, join query $S \bowtie_{S.A=S'.B} S'$ in CDM is equivalent to the CQL query:

```
Select Rstream(*) From S[Now], S'[Now]
Where S.A = S'.B
```

The second class of operators take a stream as input and produce a derived relation as output. These operators can be expressed in CQL by applying an `Unbounded` window on the input stream and translating the relational algebra operator to SQL.

The operators in CDM are strictly less expressive than CQL. CDM does not support sliding windows over streams, although it has implicit `Now` and `Unbounded`

windows as described above. Furthermore, CDM distinguishes between base relations, which can be joined with streams, and derived relations (persistent views), which cannot. These restrictions ensure that derived relations in CDM can be maintained incrementally in time logarithmic in the size of the derived relation. CQL queries, on the other hand, could require unbounded time and memory, as we have shown in [ABB+02] and addressed in [MW+03].

6.4.2 Tapestry

Tapestry queries [TGNO92] are expressed using SQL syntax. At time τ, the result of a Tapestry query Q contains the set of tuples logically obtained by executing Q as a relational SQL query at every instant $\tau' \leq \tau$ and taking the set-union of the results. This semantics for Q is equivalent to the CQL query:

```
Select Istream(Distinct *)
From (Istream(Q))[Range Unbounded]
```

Tapestry does not support sliding windows over streams or any relation-to-stream operators.

6.4.3 Tribeca

Tribeca is based on a set of stream-to-stream operators and we have shown that all of the Tribeca operators specified in [Sul96] can be expressed in CQL; details are omitted due to space limitations. Two of the more interesting operators are `demux` (demultiplex) and `mux` (multiplex). In a Tribeca query the `demux` operator is used to split a single stream into an arbitrary number of substreams, the substreams are processed separately using other (stream-to-stream) operators, then the resulting substreams are merged into a single result stream using the `mux` operator. This type of query is expressed in CQL using a combination of partitioned window and `Group By`.

Like chronicles and Tapestry, Tribeca is strictly less expressive than CQL. Tribeca queries take a single stream as input and produce a single stream as output, with no notion of relation. CQL queries can have multiple input streams and can freely mix streams and relations.

6.4.4 Aurora

Aurora queries are built from a set of eleven operator types. Operators are composed by users into a global query execution plan via a "boxes-and-arrows" graphical interface. It is somewhat difficult to compare the procedural query interface of Aurora against a declarative language like CQL, but we can draw some distinctions.

The aggregation operators of Aurora (*Tumble*, *Slide*, and *XSection*) are each defined from three user-defined functions, yielding nearly unlimited expressive power. The aggregation operators also have optional parameters related to system and application time (see Section 7). For example, these parameters can direct the operator to take certain action if no stream element has arrived for T seconds, making the semantics dependent on stream arrival rates and nondeterministic, an approach we have not considered to date in CQL.

All operators in Aurora are stream-to-stream, and Aurora does not explicitly support relations. Therefore, in order to express CQL queries involving derived relations and relation-to-relation operators, Aurora procedurally stores and manipulates state corresponding to a derived relation.

7 Discussion

7.1 Stream-Only Query Language

Our abstract semantics and therefore CQL distinguish two fundamental data types, namely, relations and streams. We can derive a stream-only language, L_s, from our language L as follows.

- Corresponding to each n-ary relation-to-relation operator O in L, there is an n-ary stream-to-stream operator O_s in L_s. The semantics of $O_s(S_1, \ldots, S_n)$ when expressed in L is $\texttt{Rstream}(O(S_1[\texttt{Now}], \ldots, S_n[\texttt{Now}]))$.
- Corresponding to each window operator W in L, there is a unary stream-to-stream operator W_s in L_s. The semantics of $S[W_s]$ when expressed in L is $\texttt{Rstream}(S[W])$.
- There are no operators in L_s corresponding to relation-to-stream operators of L.

It can be shown that L and L_S have essentially the same expressiveness. Clearly any query in L_S can be rewritten in L. Given a query Q in L, we obtain a query Q_S in L_S by performing the following three steps. First, transform Q to an equivalent query Q' that has `Rstream` as its only relation-to-stream operator (this step is always possible as indicated in Section 5.1). Second, replace every input relation R_i in Q' with $\texttt{Rstream}(R_i)$. Finally, replace every relation-to-relation and window operator in Q with its L_S equivalent according to the definitions above. As it turns out, the language L is quite similar to the stream-to-stream approach being taken in *TelegraphCQ* [C+03].

We chose our dual approach over the stream-only approach for at least two reasons. First, our experience with a large number of queries [SQR] suggests that the dual approach results in more intuitive queries than the stream-only approach. Second, having both relations and streams cleanly generalizes materialized views, as discussed in detail in Section 6.4. Note that the *Chronicle Data Model* [JMS95] also takes an approach similar to ours—it supports both chronicles (closely related to streams) and materialized views (relations). The Chronicle Data Model was also discussed in detail in Section 6.4.

7.2 Equivalences and Query Transformations

Recall goal #2 from Section 1: We should not inhibit standard relational transformations with our new constructs, operators, or semantic interpretation, and we should enable

new transformations specific to streams. Our abstract semantics guarantees that all equivalences that hold in SQL with standard relational semantics continue to hold in CQL, including subquery flattening, join reordering, predicate pushdown, etc. Furthermore, since any CQL query or subquery producing a relation can be thought of as a materialized view, all equivalences from materialized view maintenance [GM95] can be applied to CQL. For example, a materialized view joining two relations generally is maintained incrementally rather than by recomputation, and the same approach can be used to join two relations (or windowed streams) in CQL.

Here we consider two stream-based transformations: *window reduction* and *filter-window commutativity*. The identification of other useful stream-based syntactic transformations is left as future work, noting that we may also perform some transformations at the query execution plan level instead [MW+03].

7.2.1 Window Reduction

The following equivalence can be used to rewrite CQL queries or subqueries with an `Unbounded` window and an `Istream` operator into an equivalent (sub)query with a `Now` window and an `Rstream` operator. Here, L is any select-list, S is any stream (including a subquery producing a stream), and C is any condition.

```
Select Istream(L) From S[Range Unbounded]
Where C
```

$\equiv$

```
Select Rstream(L) From S[Now] Where C
```

Furthermore, if stream S has a key (no duplicates), then we need not replace the `Istream` operator with `Rstream`, although once a `Now` window is applied there is little difference in efficiency between `Istream` and `Rstream`.[4] We saw an example of the window-reduction transformation in Query 6.1 from Section 6.3.

Transforming `Unbounded` to `Now` obviously suggests a much more efficient implementation—logically, `Unbounded` windows require buffering the entire history of a stream, while `Now` windows allow a stream tuple to be discarded as soon as it is processed. In separate work we have developed techniques for transforming `Unbounded` windows into `[Rows N]` windows, but those transformations rely on many-one joins and sophisticated *constraints* over the streams [BW02].

We may find other cases or more general criteria whereby `Unbounded` windows can be replaced by `Now` windows; a detailed exploration is left as future work.

7.2.2 Filter-Window Commutativity

Another equivalence that can be useful for query-rewrite optimization is the commutativity of selection conditions and time-based windows. Here, L is any select-list, S is any stream (including a subquery producing a stream), C is any condition, and T is any time-interval.

```
(Select L From S Where C)[Range T]
```

$\equiv$

```
Select L From S[Range T] Where C
```

For example, this equivalence can be used to rewrite Query 6.2 from Section 6.3 as:

```
Select Count(*) From
 (Select * From Bid
  Where item_id >= 100 and item_id <= 200)
 [Range 1 Hour]
```

If the system uses a query evaluation strategy based on materializing the windows specified in a query, then filtering before applying the window instead of after is preferable since it reduces steady-state memory overhead [MW+03]. Note that the converse transformation might also be applied. If the author writes the query as specified here, we might prefer to move the filtering condition out of the window as originally specified, in order to allow the window to be shared by multiple queries or subqueries [MW+03].

7.3 Timestamps and Physical Time

Our language semantics is independent of the time domain $\mathcal{T}$ used by an application—we only require that the domain be discrete and ordered. Furthermore, there need be no direct relationship between $\mathcal{T}$ and physical clock-time at the Data Stream Management System (DSMS). Specifically, the timestamp of input stream elements or relation-updates need not be related to their time of arrival at the DSMS. (Note that in order to precisely determine instantaneous input relations at every time $\tau \in \mathcal{T}$ the updates to the relations have to be timestamped from the time domain $\mathcal{T}$. In the reminder of this section, for brevity, when we say input stream elements we mean both actual stream elements and timestamped input relation updates). For example, stream applications may use sequence numbers as timestamps if only the relative ordering of the elements is important. In addition, the order of arrival of stream elements at the DSMS need not be consistent with their timestamp ordering. For example, stream elements may be generated by a remote source, and the network conveying the elements to the DSMS may not guarantee in-order transmission. Our one requirement is that, for a given query, all stream element timestamps are from the same time domain $\mathcal{T}$.

However, an application time domain $\mathcal{T}$ and physical time cannot be completely unrelated in a practical DSMS. At the very least, for each time $\tau \in \mathcal{T}$ the system implementing our semantics must know at some physical time t that no new stream elements with timestamp $\leq \tau$ will ever arrive—otherwise, the system cannot produce any output with timestamp τ or greater.

Our current approach is to assume an additional "meta-input" to the system called *heartbeats*. A heartbeat consists

[4] More generally, `Istream` and `Rstream` are equivalent over any relation R for which $R(\tau) \cap R(\tau - 1) = \emptyset$ for all τ. A common example is a relation produced by a `Now` window on an input stream with a key.

of a timestamp $\tau \in \mathcal{T}$, and has the semantics that the system will receive no future stream elements with timestamp $\leq \tau$. There are various ways by which heartbeats may be generated. Here are three examples:

1. In the easiest and a fairly common case, timestamps are assigned using the DSMS clock when stream tuples arrive at the system. Therefore stream elements are ordered, and the clock itself provides a hearbeat.
2. The source of an input stream might generate *source heartbeats*, which indicate that no future elements in that stream will have timestamp less than or equal to that specified by the heartbeat. If all the sources of input streams generate source heartbeats, an application-level or query-level heartbeat can be generated by taking the minimum of all the source heartbeats. Note that this approach is feasible only if the heartbeats and the stream elements within a single input stream reach the DSMS in timestamp order.
3. Properties of stream sources and the system or networking environment may be used to generate heartbeats. For example, if we know that all sources of input streams use a global clock for timestamping and there is an upper bound D in delay of stream elements reaching the DSMS, at every global time t we can generate a heartbeat with timestamp $t - D$.

There is much interesting work to do in heartbeat generation and other aspects of time, which we are exploring in ongoing research.

8 Conclusion

To the best of our knowledge this paper is one of the first to provide an exact semantics for general-purpose declarative continuous queries over streams and relations. We first presented an abstract semantics based on any relational query language, any window specification language to map from streams to relations, and a set of operators to map from relations to streams. We then proposed CQL, a concrete language that instantiates the "black boxes" in our abstract semantics using SQL as the relational query language and window specifications derived from SQL-99. We showed how CQL encompasses several previous languages and semantics for continuous queries in terms of expressiveness. We identified several practical issues arising from CQL and its semantics: syntactic shortcuts and defaults for convenient and intuitive query formulation, equivalences for query optimization, and flexible application-defined time domains.

We are implementing CQL as part of a general-purpose Data Stream Management System at Stanford. Details of our overall system and our approach to query processing—including issues of resource management and approximation not touched on in this paper—are provided in [MW+03]. To date (Winter 2003) a significant fraction of CQL is running, albeit using relatively naive query execution strategies.

Acknowledgments

We are grateful to Brian Babcock for his first heroic attempts at codifying a semantics and thereby stimulating this work in his absence, to Bruce Lindsay for his excellent suggestions for dealing with time-related issues, and to the entire STREAM group at Stanford for many stimulating discussions.

References

[ABB+02] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 221–232, May 2002.

[Bar99] D. Barbara. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, December 1999.

[BBD+02] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 2002 ACM Symp. on Principles of Database Systems*, pages 1–16, June 2002.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, September 2001.

[BW02] S. Babu and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data streams. Technical report, Stanford University Database Group, November 2002. Available at http://dbpubs.stanford.edu/pub/2002-52.

[C+03] S. Chandrasekaran et al. TelegraphCQ: Continuous Datafbw Processing for an Uncertain World. In *Proc. of the 2003 Conf. on Innovative Data Systems Research*, pages 269–280, 2003.

[CCC+02] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams–a new class of data management applications. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002. Material augmented by personal communication.

[CDTW00] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[CF02] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proc. 28th Intl. Conf. on Very Large Data Bases*, August 2002.

[DGGR02] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 61–72, 2002.

[GM95] A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Data Engineering Bulletin*, 18(2):3–18, June 1995.

[JMS95] H.V. Jagadish, I.S. Mumick, and A. Silberschatz. View maintenance issues for the Chronicle data model. In *Proc. of the 1995 ACM Symp. on Principles of Database Systems*, pages 113–124, May 1995.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engineering*, 11(4):583–590, August 1999.

[MF02] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, pages 555–566, February 2002.

[MSHR02] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[MW+03] R. Motwani, J. Widom, et al. Query processing, resource management, and approximation in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, pages 245–256, January 2003.

[NACP01] B. Nguyen, S. Abiteboul, G. Cobena, and M. Preda. Monitoring XML data on the web. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, pages 437–448, May 2001.

[OS95] G. Ozsoyoglu and R. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowledge and Data Engineering*, 7(4):513–532, August 1995.

[PD99] N. Paton and O. Diaz. Active database systems. *ACM Computing Surveys*, 31(1), 1999.

[SLR95] P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: a model for sequence databases. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pages 232–239, March 1995.

[SQR] SQR – A Stream Query Repository. http://www.db.stanford.edu/stream/sqr. Joint effort of several data stream research groups.

[Sul96] M. Sullivan. Tribeca: A stream database manager for network traffi c analysis. In *Proc. of the 1996 Intl. Conf. on Very Large Data Bases*, page 594, September 1996.

[TGNO92] D.B. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD Intl. Conf. on Management of Data*, pages 321–330, June 1992.

[TTPM02] P. A. Tucker, K. Tufte, V. Papadimos, and D. Maier. NEXMark – a benchmark for querying data streams, 2002. Manuscript available at http://www.cse.ogi.edu/dot/niagara/NEXMark/.

[Vit85] J.S. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, 1985.

[VN02] S. Viglas and J. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 37–48, June 2002.

[WZL02] H. Wang, C. Zaniolo, and R.C. Luo. ATLaS: A Turing-complete extension of SQL for data mining applications and streams, 2002. Manuscript available at http://wis.cs.ucla.edu/publications.html.