VLDB Journal manuscript No.
(will be inserted by the editor)

The CQL Continuous Query Language: Semantic Foundations and Query Execution*

Arvind Arasu, Shivnath Babu, Jennifer Widom

Stanford University
e-mail: {arvinda,shivnath,widom}@cs.stanford.edu

Abstract. *CQL*, a *Continuous Query Language*, is supported by the STREAM prototype Data Stream Management System at Stanford. CQL is an expressive SQL-based declarative language for registering continuous queries against streams and stored relations. We begin by presenting an abstract semantics that relies only on "black box" mappings among streams and relations. From these mappings we define a precise and general interpretation for continuous queries. CQL is an instantiation of our abstract semantics using SQL to map from relations to relations, window specifications derived from SQL-99 to map from streams to relations, and three new operators to map from relations to streams. Most of the CQL language is operational in the STREAM system. We present the structure of CQL's query execution plans as well as details of the most important components: operators, inter-operator queues, synopses, and sharing of components among multiple operators and queries.

Examples throughout the paper are drawn from the *Linear Road* benchmark recently proposed for Data Stream Management Systems. We also curate a public repository of data stream applications that includes a wide variety of queries expressed in CQL. The relative ease of capturing these applications in CQL is one indicator that the language contains an appropriate set of constructs for data stream processing.

Keywords: Data Streams, Continuous Queries, Query Language, Query processing

1 Introduction

There has been a considerable surge of research in many aspects of processing continuous queries over unbounded data streams [24, 25]. Many papers include example continuous queries expressed in some declarative language, e.g., [2, 19, 20, 23, 28, 32]. However, these queries tend to be simple and primarily for illustration—a precise language semantics, particularly for more complex queries, often is left unclear. Furthermore, very little has been published to date covering execution details of general-purpose continuous queries. In this paper we present the *CQL* language and execution engine for general-purpose continuous queries over streams and stored relations. CQL (for *Continuous Query Language*) is an instantiation of a precise abstract continuous semantics also presented in this paper, and CQL is implemented in the *STREAM* prototype Data Stream Management System (DSMS) at Stanford [40].

It may appear initially that defining a continuous query language over (relational) streams is not difficult: 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. Consider the following simple query:

```
Select P.price
From Items[Rows 5] as I, PriceTable as P
Where  I.itemID = P.itemID
```

`Items` is a stream of purchased items, `PriceTable` is a stored relation containing the price of items, and `[Rows 5]` specifies a 5-element *sliding window*. Even this simple query has no single obvious interpretation that we know of. For example, is the result of the query a stream or a relation? What happens to the query result when the price of a recently-purchased item—i.e., an item still within the 5-element window—changes?

In this paper, we initially define a precise *abstract semantics* for continuous queries. Our abstract semantics is based on two data types—streams and relations—and three classes of operators over these types: operators that produce a relation from a stream (*stream-to-relation*), operators that produce a relation from other relations (*relation-to-relation*), and

* This work was supported by the National Science Foundation under grants IIS-0118173 and IIS-9817799, and by a 3Com Stanford Graduate Fellowship.

operators that produce a stream from a relation (*stream-to-relation*). The three classes of operators are "black box" components of our abstract semantics: the abstract semantics does not depend on the actual behavior of the operators in these classes, but only on their input and output types.

CQL instantiates the black boxes in our abstract semantics: It uses SQL to express its relation-to-relation operators, a window specification language derived from SQL-99 to express its stream-to-relation operators, and a set of three operators for its relation-to-stream operators. Most of CQL is fully operational in our prototype DSMS [40]. CQL has been used to specify the *Linear Road* benchmark proposed for data stream systems [4], and to specify a variety of other stream applications in a public repository we are curating [37].

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

1. We wanted to exploit well-understood relational semantics (and by extension relational rewrites and execution strategies) 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. We wanted the language to have sufficient constructs to capture a wide variety of stream applications, without allowing the "feature creep" that can result in an esoteric, difficult to understand, or difficult to implement language. That is, we wanted to keep the language as simple as possible without sacrificing too much expressiveness.

We believe these goals have been achieved to a large extent.

The STREAM query processing engine is based on *physical query plans* generated from CQL textual queries. Often query plan merging occurs, so a single query plan may compute multiple continuous queries. In this paper we focus on the structure and details of the execution plans themselves, not on how initial plans are selected or how plans adapt over time (which is the subject of other papers [10, 11, 13]).

In developing the structure of our query execution plans we had certain goals in mind:

4. We wanted plans built from modular and pluggable components based on generic interfaces, especially for operators and synopsis structures.
5. We wanted an execution model that efficiently captures the combination of streams and relations that forms the basis of our language.
6. We wanted an architecture permitting easy experimentation with different strategies for operator scheduling, overflowing state to disk, sharing state and computation among multiple continuous queries, and other crucial issues affecting performance.

Here too we believe these goals have been achieved to a large extent.

To summarize the contributions of this paper:

- We formalize streams and relations that are updated over time (Section 4), and we define an abstract semantics for continuous queries based on three black-box classes of operators: stream-to-relation, relation-to-relation, and relation-to-stream (Section 5).
- We define our concrete language, CQL, which instantiates the black boxes in the abstract semantics as discussed earlier (Section 6). We define syntactic shortcuts and defaults in CQL for convenient and intuitive query formulation, and we point out a few equivalences in the language (Section 10).
- We illustrate CQL using a hypothetical road traffic management application proposed as a benchmark for data stream systems [4] (Sections 3 and 7). We also compare the expressiveness of CQL against related query languages (Section 11).
- We describe the query execution plans and strategies used in the STREAM system for CQL queries. We focus specifically on operators, inter-operator queues, synopses, and sharing of components among multiple operators and queries (Section 12).

2 Related Work

A preliminary description of our abstract semantics and the CQL language appeared as an invited paper in [3]. That paper did not include query transformations, the Linear Road benchmark, or any of the material on query execution included in this paper.

A comprehensive description of work related to data streams and continuous queries is given in [8]. 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* [26] are a form of continuous query, since a view is continuously updated to reflect changes to its base relations. Reference [29] 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* [43] with a SQL-based language called *TQL*. (A similar language is considered in [14].) 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., [20,31,34]. The semantics of continuous queries in these systems is also based on periodic execution of one-time queries as in Tapestry. In Section 11, 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 [18] proposes a declarative language for continuous queries with a particular focus on expressive windowing constructs. The *TelegraphCQ* language is discussed again briefly in Section 11.6. *ATLas* [30,47,48] proposes simple extensions to SQL-99 user-defined aggregates (UDAs) that make the resulting language Turing complete, suitable for various data-mining and data streams applications. Intuitively, the extensions let users express *initialize*, *iterate*, and *terminate* parts of a SQL-99 UDA specification using SQL update constructs rather than using procedural code. GSQL [21] is a SQL-like language developed for Gigascope, a DSMS for network monitoring applications. GSQL is compared against CQL in Section 11.

Several systems support procedural continuous queries, as opposed to the declarative approach in this paper. The *Aurora* system [16] 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 [41] 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 11.

Temporal query languages [35] are based on a rich model of history, and with their many special-purpose language constructs they dramatically subsume CQL in terms of expressiveness. Based on the several different applications we studied [37], the full expressive power of temporal query languages seems unnecessary for most stream applications. As mentioned in goal #3 earlier, our intention is to keep CQL much simpler than temporal languages, for ease of implementation and ease of understanding. While *sequence query languages* [36] are not as complex as temporal query languages, they still largely subsume CQL in terms of expressiveness, and they include many special-purpose operators specifically for manipulating sequences. Finally, *event-processing languages* are geared largely toward matching single events or specific event patterns against queries, usually in a *publish-subscribe* setting [49]. The fine-grained event-matching constructs of these languages also were not needed for the stream applications we studied.

Note that our abstract semantics and even our concrete language permit temporal, sequence, or event-processing capabilities to be "plugged in" as black box operators. However, if we discovered that such capabilities were needed frequently in future stream applications, it would be better to fully incorporate them as a feature of the language.

3 Introduction to Running Example

We introduce a running example based on a hypothetical road traffic management application introduced in the *Linear Road* benchmark for data stream management systems [4]. We use a simplified version of the Linear Road application to illustrate various aspects of our language, semantics, and execution plans; full details can be found in the original specification [4].

The Linear Road application implements *variable tolling* —adaptive, real-time computation of vehicle tolls based on traffic conditions—to regulate vehicular traffic on a highway. To enable variable tolling each vehicle is equipped with a sensor that continuously relays its position and speed to a central server. The server aggregates the information received from all vehicles on the highway system, computes tolls in real-time, and transmits tolls back to vehicles using the sensor network.

Figure 1 shows the highway system used by our simplified Linear Road. There is a single highway 100 miles long, which is divided into 100 1-mile *segments*. The highway has entrance and exit ramps at segment boundaries. Traffic flows in a single direction from left to right. When a vehicle is on the highway, it reports its current speed (miles per hour) and position (number of feet from the left end) to the server once every 30 seconds. (In the complete Linear Road application [4] there are 10 highways with multiple lanes, and traffic flows in both directions.)

Vehicles pay a toll whenever they drive in a *congested* segment, while no toll is charged for uncongested segments. A segment is congested if the average speed of all vehicles in the segment over the last 5 minutes is less than 40 miles per hour (MPH). The toll for a congested segment is given by the formula $2 \times (numvehicles - 50)^2$, where $numvehicles$ is the number of vehicles currently in the segment. Note that the toll for a congested segment changes dynamically as vehicles enter and leave the segment. When the server detects that a vehicle has entered a congested segment, the server outputs the current toll for the segment which is conveyed back to the vehicle.

In stream terminology, the simplified Linear Road application in this paper has:

– A single input stream—the stream of positions and speeds of vehicles.
– A single continuous query computing the tolls.
– A single output stream containing the tolls for vehicles.

As the paper progresses we will illustrate how we model, express, and execute this application using our language, semantics, and query execution strategies.

4 Streams and Relations

In this section we define a formal model of streams and relations. As in the standard relational model, each stream and relation has a fixed schema consisting of a set of named attributes. For stream element arrivals and relation updates we assume a discrete, ordered time domain $\mathcal{T}$. A *time instant* (or simply *instant*) is any value from $\mathcal{T}$. For concreteness, we

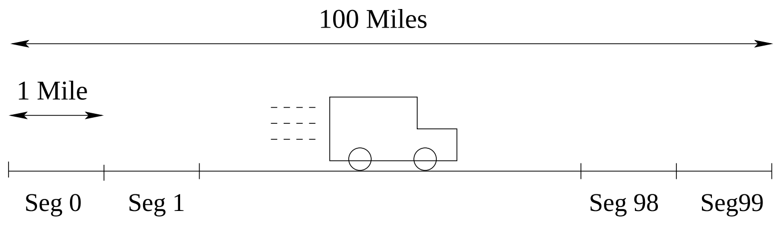


Fig. 1 The Linear Road highway system

represent $\mathcal{T}$ as the nonnegative integers $\{0, 1, \ldots\}$; in particular note that 0 stands for the earliest time instant. Time domain $\mathcal{T}$ models an application's notion of time, not particularly system or wall-clock time. Thus, although $\mathcal{T}$ may often be of type *Datetime*, our semantics only requires any discrete, ordered domain. (A thorough discussion of time issues in Data Stream Management Systems appears in [38].)

Definition 1 (Stream) *A stream S is a (possibly infinite) bag (multiset) of* elements $\langle s, \tau \rangle$, *where s is a tuple belonging to the schema of S and $\tau \in \mathcal{T}$ is the* timestamp *of the element.*

Note that the timestamp is not part of the schema of a stream, and there could be zero, one, or multiple elements with the same timestamp in a stream. We only require that there be a finite (but unbounded) number of elements with a given timestamp.

There are two classes of streams: *base streams*, which are the source data streams that arrive at the DSMS, and *derived streams*, which are intermediate streams produced by operators in a query. We use the term *tuple of a stream* to denote the data (non-timestamp) portion of a stream element.

Example 1 In the Linear Road application there is just one base stream containing vehicle speed-position measurements, with schema:

```
PosSpeedStr (vehicleId, speed, xPos)
```

Attribute `vehicleId` identifies the vehicle, `speed` denotes its current speed in MPH, and `xPos` denotes its current position within the highway in feet as described in Section 3. The time domain is of type *Datetime*, and for this application the timestamp of a stream element denotes the physical time when the position and speed measurements were taken. □

Definition 2 (Relation) *A relation R is a mapping from each time instant in $\mathcal{T}$ to a finite but unbounded bag of tuples belonging to the schema of R.*

A relation R defines an unordered bag of tuples at any time instant $\tau \in \mathcal{T}$, denoted $R(\tau)$. Note the difference between this definition for relation and the standard one: in the standard 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 the bag of tuples in a relation at a given point in time. Thus, if R denotes a relation according to Definition 2, $R(\tau)$ denotes an instantaneous relation. Often when there is no ambiguity we omit the term *instantaneous*. We use the term *base relation* for input relations and *derived relation* for relations produced by query operators.

Example 2 The Linear Road application contains no base relations, but several derived relations are useful in toll computation. For example, the toll for a congested segment depends on the current number of vehicles in the segment. We can represent the current number of vehicles in a segment using the derived relation:

```
SegVolRel (segNo, numVehicles)
```

Attribute `segNo` denotes the segment (0-99) and `numVehicles` the number of vehicles in the segment. At time τ, `SegVolRel`(τ) contains the count of vehicles in each highway segment as of time τ. Section 7 shows how `SegVolRel` can be computed from the base stream `PosSpeedStr`, and how it can be used to compute tolls. □

As this example suggests, the concept of a relation is useful even in applications whose inputs and outputs are all streams. It seems more natural to model "the current number of vehicles in a segment" as a time-varying relation, rather than as a stream of the latest values. From an expressiveness point of view, it is not necessary to have both streams and relations: we could have picked just one of streams and relations and designed our language around it without loss of expressiveness; this issue is discussed further in Section 11.6. In our implementation we encode both streams and relations uniformly as "plus-minus streams," as discussed in Section 12.

5 Abstract Semantics

This section presents our abstract semantics for continuous queries. Recall from Section 1 that our semantics is based on three classes of operators over streams and relations:

– *stream-to-relation* operators that produce a relation from a stream
– *relation-to-relation* operators that produce a relation from one or more other relations
– *relation-to-stream* operators that produce a stream from a relation

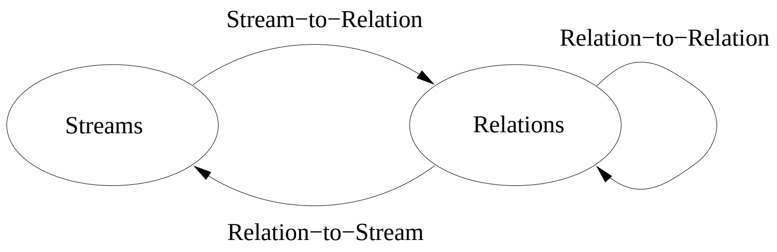


Fig. 2 Operator classes and mappings used in abstract semantics

Recall that by "relation" we are referring to our formal notion of a time-varying relation, as defined in Section 4.

Stream-to-stream operators are absent—they have to be composed from operators of the three classes above. As we will discuss in detail in Section 6, the rationale for this decision is based primarily on our goal #1 from Section 1: exploiting well-understood relational semantics (and by extension relational rewrites and execution strategies) to the extent possible.

First some terminology: S *up to* τ denotes the bag of elements in stream S with timestamps $\leq \tau$, *i.e.*, $\{\langle s, \tau' \rangle \in S : \tau' \leq \tau\}$. S *at* τ denotes the bag of elements of S with timestamp τ, *i.e.*, $\{\langle s, \tau' \rangle \in S : \tau' = \tau\}$. Similarly, R *up to* τ denotes the collection of instantaneous relations $R(0), \ldots, R(\tau)$, and R *at* τ denotes the instantaneous relation $R(\tau)$.

1. A *stream-to-relation* operator takes a stream S as input and produces a relation R as output with the same schema as S. At any instant τ, $R(\tau)$ should be computable from 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 $R_1(\tau), \ldots, R_n(\tau)$.
3. A *relation-to-stream* operator takes a relation R as input and produces a stream S as output with the same schema as R. At any instant τ, S at τ should be computable from R up to τ.

Now we define our abstract semantics.

Definition 3 (Continuous Semantics) *Consider a query Q that is any type-consistent composition of operators from the above three classes. Suppose the set of all inputs to the innermost (leaf) operators of Q are streams $S_1, \ldots, S_n$ ($n \geq 0$) and relations $R_1, \ldots, R_m$ ($m \geq 0$). We define the result of continuous query Q* at a time τ, *which denotes the result of Q once all inputs up to τ are "available" (a notion discussed below). There are two cases:*

– *Case 1: The outermost (topmost) operator in Q is relation-to-stream, producing a stream S (say). The result of Q at time τ is S up to τ, produced by recursively applying the operators comprising Q to streams $S_1, \ldots, S_n$ up to τ and relations $R_1, \ldots, R_m$ up to τ.*
– *Case 2: The outermost (topmost) operator in Q is stream-to-relation or relation-to-relation, producing a relation R (say). The result of Q at time τ is $R(\tau)$, produced by recursively applying the operators comprising Q to streams $S_1, \ldots, S_n$ up to τ and relations $R_1, \ldots, R_m$ up to τ.*

Based on this definition, informally we can think of continuous queries operationally as follows. Let time "advance" within domain $\mathcal{T}$, further discussed below. First consider a query producing a stream. At time $\tau \in \mathcal{T}$, all inputs up to τ are processed and the continuous query emits any new stream result elements with timestamp τ. Because of our assumptions on operators, no new stream elements with timestamp $< \tau$ can be produced from inputs with timestamp $\geq \tau$. A query producing a relation is similar: At time τ, all inputs up to τ are processed and the continuous query updates the output relation to state $R(\tau)$.

Now let us understand what it means for time to advance within domain $\mathcal{T}$. The relationship between application time, wall-clock time, and system time is a complex issue discussed in depth in a separate paper [38]. However, for precise query semantics we need make no additional assumptions beyond those already made in this paper. Time "advances" to τ from $\tau - 1$ when all inputs up to $\tau - 1$ have been processed. It appears we are tacitly assuming that streams arrive in timestamp order, relations are updated in timestamp order, and there is no timestamp "skew" across streams or relations. In practice, to implement our semantics correctly, systems cope with out of order and skewed inputs. This issue is revisited in Section 8 and thoroughly covered by [38].

Example 3 Consider the query `Istream(Filter (LastRow(S)))` constructed from three operators and operating on stream `S`.[1] Let stream `S` have a single attribute and consist of the elements $\{\langle (a_0), 0 \rangle, \langle (a_1), 1 \rangle, \langle (a_2), 2 \rangle, \ldots\}$. `LastRow` is a stream-to-relation operator; at any point in time the relation output by `LastRow` contains the last tuple that arrived on S. `Filter` is a relation-to-relation operator that produces its output relation by applying a filter condition on its input. Suppose that tuples $(a_0), (a_2), (a_4), \ldots$ satisfy the filter condition while tuples $(a_1), (a_3), (a_5), \ldots$ do not. Finally, `Istream` is a relation-to-stream operator (defined formally in Section 6.3) that "streams" every new tuple inserted into its input relation. Figure 3 shows the outputs produced by each of the three operators as time progresses. □

[1] In CQL, this query is expressed as "`Select Istream(*) From S [Rows 1] Where <filter>`."

Time	`S`	`LastRow`	`Filter`	`Istream`
0	$\langle(a_0),0\rangle$	(a_0)	(a_0)	$\langle(a_0),0\rangle$
1	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$	(a_1)	ϕ	$\langle(a_0),0\rangle$
2	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$	(a_2)	(a_2)	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$
3	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$	(a_3)	ϕ	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$
4	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$ $\langle(a_4),4\rangle$	(a_4)	(a_4)	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$ $\langle(a_4),4\rangle$
⋮	⋮	⋮	⋮	⋮

Fig. 3 Output produced by operators of Example 3 as time progresses.

Example 4 In the Linear Road application the sequence of operators producing derived relation `SegVolRel` of Example 2 conceptually produces, at every time instant τ, the instantaneous relation `SegVolRel`(τ) containing the current number of vehicles in each segment. In a DSMS implementing our semantics, `SegVolRel`(τ) cannot be produced until it is known that all elements on input stream `PosSpeedStr(vehicleId,speed,xPos)` with timestamp $\leq \tau$ have been received. Furthermore, once they have, there may be additional lag before the relation is actually updated due to query processing time. Our semantics does not dictate "liveness" of continuous query output—that issue is relegated to latency management in the query processor [7, 17]. □

A formal specification of the abstract semantics using *denotational semantics* can be found in [5].

6 Continuous Query Language

This section presents our concrete language, CQL, which is defined by instantiating the operators of our abstract semantics. We also specify a few syntactic shortcuts and defaults to simplify expression of some common operations.

Broadly, our approach to designing operators in CQL is as follows: Support a large class of relation-to-relation operators, which perform the bulk of data manipulation in a typical CQL query, along with a small set of stream-to-relation and relation-to-stream operators that convert streams to relations and back. The primary advantage of this approach is the ability to reuse the formal foundations and huge body of implementation techniques for relation-to-relation languages such as relational algebra and SQL, instead of starting from scratch with a heavily stream-based language. Furthermore, as we will see, queries in CQL are quite natural to express.

Technically, we cannot directly import existing conventional relation-to-relation operators into our concrete language, since they operate on instantaneous relations while we operate on time-varying relations, but the mapping is obvious: Let O_r denote a traditional relational operator or query over instantaneous input relations $R_1, \ldots, R_n$. The corresponding relation-to-relation operator O_c in CQL produces the time-varying relation R such that at each time τ, $R(\tau) = O_r(R_1(\tau), \ldots, R_n(\tau))$.

An apparent drawback of our approach is that even a simple filter on a stream requires three operators: one to turn the stream into a relation, one to perform a relational filter, and one to turn the relation back into a stream. However, CQL's defaults and syntactic shortcuts make filters and other simple queries easy to express (Section 6.4).

Although we do not specify them explicitly as part of our language, incorporating user-defined procedures, aggregates, and windows, as may be required for more complex, application-specific stream processing, is straightforward in CQL, at least from the semantics perspective.

Next in Sections 6.1–6.3 we cover the three classes of operators in CQL.

6.1 Stream-to-Relation Operators

Currently all stream-to-relation operators in CQL are based on the concept of a *sliding window* over a stream: a window that at any point of time contains a historical snapshot of a finite portion of the stream. We have three classes of sliding-window operators in CQL: *time-based*, *tuple-based*, and *partitioned*, defined below. Syntactically, the sliding-window operators are specified using a window specification language derived from SQL-99. Other types of sliding windows such as *fixed* windows [41], *tumbling* windows [16], *value-based* windows [36], 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 the development of our abstract semantics.

6.1.1 Time-based sliding windows A time-based sliding window on a stream S takes a time-interval T as a parameter and is specified by following the reference to S with `[Range T]`.[2] We do not specify a syntax or restrictions for time-interval T at this point, but we assume it specifies a computable period of application time. Intuitively, a time-based window defines its output relation over time by sliding an interval of size T time units capturing the latest portion of an ordered stream. More formally, the output relation R of "`S [Range T]`" is defined as:

[2] In all three of our window types we dropped the keyword `Preceding` appearing in the SQL-99 syntax and in our earlier specification [33]—we only have "preceding" windows for now so the keyword is superfluous.

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

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]`." We use the terms `Now` *window* and `Unbounded` *window* to refer to these two special windows.

Example 5 "`PosSpeedStr [Range 30 Seconds]`" denotes a time-based sliding window of 30 seconds over input stream `PosSpeedStr`. At any time instant, the output relation of the sliding window contains the bag of position-speed measurements from the previous 30 seconds. Similarly, at any instant "`PosSpeedStr [Now]`" contains the (possibly empty) bag of position-speed measurements from that instant, and "`PosSpeedStr [Range Unbounded]`" contains the bag of all position-speed measurements so far. □

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 the reference to S in the query with `[Rows N]`. Intuitively, a tuple-based window defines its output relation over time by sliding a window of the last N tuples of an ordered stream. More formally, for the output relation R of "`S [Rows N]`," $R(\tau)$ consists of the N tuples of S with the largest timestamps $\leq \tau$ (or all tuples 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 for clarity let us 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]`.

Example 6 A tuple-based sliding window does not make much sense over stream `PosSpeedStr` (except the case of $N = \infty$) since stream element timestamps are not unique. For example, at any instant sliding window `PosSpeedStr[Rows 1]` denotes the "latest" position-speed measurement, which is ambiguous whenever multiple measurements carry the same timestamp—a common occurrence in the Linear Road application. □

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 the reference to S in the query with `[Partition By` A_1`,...,`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 $\langle s, \tau' \rangle \in S$, $\tau' \leq \tau$ such that τ' is among the N largest timestamps of 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.

Example 7 The partitioned window "`PosSpeedStr [Partition By vehicleId Rows 1]`" partitions stream `PosSpeedStr` into substreams based on `vehicleId` and picks the latest element in each substream. (Note that there is no ambiguity in picking the latest element in each substream, since position-speed reports for a particular vehicle are made only once in 30 seconds and the granularity of *Datetime* is one second.) At any time instant, the relation defined by the window contains the latest speed-position measurement for each vehicle that has ever transmitted a measurement. □

6.1.4 Windows with a "slide" parameter Windows can optionally contain a *slide* parameter, indicating the granularity at which the window slides. The slide parameter is a time-interval for time-based windows and a positive integer for row-based and partitioned windows.

A time-based window over stream S with window size T and slide parameter L is denoted as "`S [Range T Slide L]`". Its output relation is:

$$R(\tau) = \begin{cases} \phi & \text{if } \tau < L - 1 \\ \{s \mid \langle s, \tau' \rangle \in S \ \wedge \ (\tau' \geq \tau_{end}) \\ \qquad \wedge \ (\tau' \leq \tau_{start})\} & \text{otherwise} \end{cases}$$

where

$$\tau_{start} = \lfloor \tau / L \rfloor \cdot L$$
$$\tau_{end} = \max\{\tau_{start} - T, 0\}$$

(The expression $(\lfloor \tau / L \rfloor \cdot L)$ computes the largest multiple of L smaller than τ.) Intuitively, "`S [Range T Slide L]`" defines its output relation by sliding an interval of width T time units over S, but the interval slides once only every L time units, by an amount L. Note that we can treat "`S [Range T]`" as an abbreviation for `S [Range T Slide 1]`, where 1 is the granularity of the time domain. The following example illustrates the use of the slide parameter.

Example 8 "`PosSpeedStr [Range 1 Minute Slide 1 Minute]`" denotes a one-minute window over `PosSpeedStr` that slides at a one-minute granularity. At any point in time the window contains the speed-position reports from the last clock minute. For example, at time instant 60-sec, the window contains the first minute (1-60) of speed-position reports, and continues to contain the same bag of tuples until the time instant 120-sec, when it shifts to the next minute (61-120), and so on. This type of window in which $L = T$ has been referred to as a *tumbling* window in previous work [16, 18]. □

Tuple-based and partitioned windows with a slide parameter are defined analogously.

6.2 Relation-to-Relation Operators

The relation-to-relation operators in CQL are derived from traditional relational queries expressed in SQL, with the straightforward semantic mapping to time-varying relations specified at the beginning of this section. Anywhere a traditional relation is referenced in a SQL query, a (base or derived) relation can be referenced in CQL.

Example 9 Consider the following CQL query for the Linear Road application:

```
Select Distinct vehicleId
From PosSpeedStr [Range 30 Seconds]
```

This query is composed from a stream-to-relation sliding-window operator, followed by a relation-to-relation operator that performs projection and duplicate-elimination. The output relation of this query contains, at any time instant, the set of "active vehicles"—those vehicles having transmitted a position-speed measurement within the last 30 seconds. □

6.3 Relation-to-Stream Operators

CQL has three relation-to-stream operators: *Istream*, *Dstream*, and *Rstream*. In the following formal definitions, operators $\cup$, $\times$, and $-$ are assumed to be the bag versions.

1. `Istream` (for "insert stream") applied to relation R contains a stream element $\langle s, \tau \rangle$ whenever tuple s is in $R(\tau) - R(\tau - 1)$. Assuming $R(-1) = \phi$ for notational simplicity, we have:
$$\texttt{Istream}(R) = \bigcup_{\tau \geq 0} ((R(\tau) - R(\tau - 1)) \times \{\tau\})$$
2. 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:
$$\texttt{Dstream}(R) = \bigcup_{\tau > 0} ((R(\tau - 1) - R(\tau)) \times \{\tau\})$$
3. `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:
$$\texttt{Rstream}(R) = \bigcup_{\tau \geq 0} (R(\tau) \times \{\tau\})$$

A careful reader may observe that `Istream` and `Dstream` are expressible using `Rstream` along with time-based sliding windows and some relational operators. However, we retain all three operators in CQL in keeping with goal #2 from Section 1: easy queries should be easy to write.

Example 10 Consider the following CQL query for stream filtering:

```
Select Istream(*)
From PosSpeedStr [Range Unbounded]
Where speed > 65
```

(Note the syntax of the relation-to-stream operator in the `Select` clause.) This query is composed from three operators: an `Unbounded` window producing a relation that at time τ contains all speed-position measurements up to τ, a relational filter operator that restricts the relation to those measurements with speed greater than 65 MPH, and an `Istream` operator that streams new values in the (filtered) relation as the result of the query. The effect is a simple filter over `PosSpeedStr` that outputs all input elements with speed greater than 65 MPH. The same filter query can be written using the `Rstream` operator and a `Now` window:

```
Select Rstream(*)
From PosSpeedStr [Now]
Where speed > 65
```

As we will see shortly, our defaults also permit this query to be written in its most intuitive form:

```
Select *
From PosSpeedStr
Where speed > 65
```

□

Example 11 The following query illustrates the use of `Dstream`:

```
Select Dstream(VehicleId)
From PosSpeedStr [Range 30 Seconds]
```

This query is composed from three operators. The time-based window operator produces the relation containing the speed-position reports in the previous 30 seconds. The relation-to-relation operator (`Select-From` clause) projects the `vehicleId` attribute from this relation. Finally, the `Dstream` operator produces a `vehicleId` in the output whenever a vehicle is deleted from the above relation. In other words, the element $\langle \mathtt{v}, \tau \rangle$ appears in the output stream whenever vehicle `v` reported its position and speed at time $\tau - 30$, but did not do so at time τ. This query thus detects when vehicles exit from the highway. (Recall from Section 3 that a vehicle on a highway reports its speed and position every 30 seconds.) □

The `Istream` operator is used most commonly with `Unbounded` windows to express filter conditions as shown above, or to stream the results of sliding-window join queries. The `Rstream` operator is used most commonly with `Now` windows to express filter conditions as shown above, or to stream the results of joins between streams and relations, as we will see in Query 6 of Section 7. The `Dstream` operator is used less frequently than `Istream` or `Rstream`; see [37] for examples of its use.

6.4 Syntactic Shortcuts and Defaults

In keeping with goal #2 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 and omitting relation-to-stream operators.

Default Windows When a stream is referenced in a CQL query where a relation is expected (most commonly in the `From` clause), an `Unbounded` window is applied to the stream by default. 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 in Section 7.

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 [33].
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 $\tau_1 \leq \tau_2$. Since we cannot test monotonicity in the general case, we use a conservative static monotonicity test. 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.

Example 12 Now we see why the filter query of Example 10 can written in its most intuitive form:

```
Select *
From PosSpeedStr
Where speed > 65
```

Since `PosSpeedStr` is referenced without a window specification, an `Unbounded` window is applied by default. Further, since the output relation of the window and filter operators is monotonic, we add a default `Istream` operator to the result. □

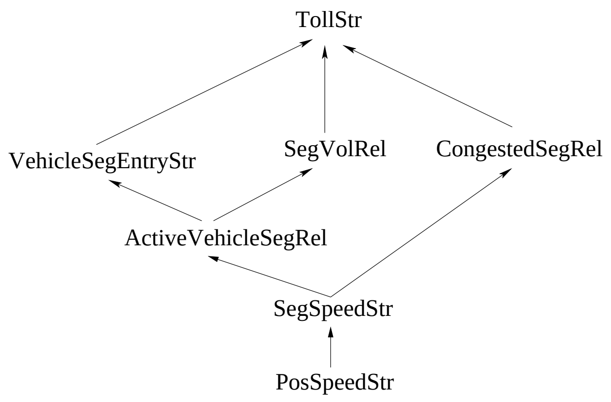


Fig. 4 Derived relations and streams for Linear Road queries

7 Linear Road in CQL

Recall that the Linear Road application has one base input stream, `PosSpeedStr`, containing speed-position measurements of vehicles using the highway. The output is a single stream `TollStr(vehicleId,toll)` specifying tolls for vehicles. Whenever a vehicle with `vehicleId v` enters a congested segment at time τ, `TollStr` contains the element $\langle(\texttt{v},\texttt{l}),\tau\rangle$ where `l` denotes the toll for the congested segment at time τ.

We incorporate two assumptions suggested in the original Linear Road specification [4] for computing tolls:

1. A vehicle is considered to have entered a segment when the first speed-position measurement for the vehicle is transmitted from that segment. The vehicle is considered to remain in the segment until it exits (see Assumption 2 next) or enters another segment (*i.e.*, a speed-position measurement is transmitted from a different segment).
2. A vehicle is considered to have exited the highway when no speed-position report for that vehicle is transmitted for 30 seconds.

These assumptions are necessary given that each vehicle transmits its speed-position measurement only once each 30 seconds.

Since the continuous query producing `TollStr` is fairly complex, we express it using several named derived relations and streams. Figure 4 shows the derived relations and streams that we use, and their interdependencies. For example, `TollStr` is produced from derived stream `VehicleSegEntryStr` and derived relations `CongestedSegRel` and `SegVolRel`. Our one base input stream `PosSpeedStr` naturally appears as the source. We present specifications for the derived streams and relations in topological order according to Figure 4. For each derived stream and relation, we first describe its meaning, followed by the CQL (sub)query that produces it.

Query 1 `SegSpeedStr(vehicleId,speed,segNo)`: This stream is obtained from `PosSpeedStr` by replacing the `xPos` attribute of each element with the corresponding segment number. Since segments are exactly 1 mile long, the segment number is computed by (integer-)dividing `xPos` by 5280, the number of feet in a mile.

```
Select vehicleId, speed,
       xPos/5280 as segNo
From PosSpeedStr
```

Note the use of a default `Unbounded` window and a default `Istream` operator in this query. □

Query 2 `ActiveVehicleSegRel(vehicleId,segNo):` At any instant τ, this relation contains the current segments of "active" vehicles, *i.e.*, vehicles currently using the highway system.

```
Select vehicleId, segNo
From SegSpeedStr [Range 30 Seconds]
```

Informally, the query uses a time-based window to identify currently active vehicles based on Assumption 2 above.[3] □

Query 3 `VehicleSegEntryStr(vehicleId,segNo):` A vehicle `v` entering a segment `s` at time τ produces an element $\langle($`v,s`$), \tau\rangle$ on this stream.

```
Select Istream(*)
From ActiveVehicleSegRel
```

`VehicleSegEntryStr` is produced by applying the `Istream` operator to `ActiveVehicleSegRel`. A vehicle `v` entering a segment `s` at time τ causes a new tuple to appear in `ActiveVehicleSegRel` at τ, which causes the `Istream` operator to produce an element $\langle($`v,s`$), \tau\rangle$ in `VehicleSegEntryStr`. □

Query 4 `CongestedSegRel(segNo)`: At any instant τ, this relation contains the current set of congested segments. Recall from Section 3 that a segment is considered congested if the average speed of vehicles in the segment in the previous 5 minutes is less than 40 MPH.

```
Select segNo
From SegSpeedStr [Range 5 Minutes]
Group By segNo
Having Avg(speed) < 40
```

□

Query 5 `SegVolRel(segNo,numVehicles)`: This relation was introduced in Example 2. At any instant τ, this relation contains the current count of vehicles in each segment.

```
Select segNo,
       count(vehicleId) as numVehicles
From ActiveVehicleSegRel
Group By segNo
```

□

Query 6 `TollStr(vehicleId,toll)`: This is the final output toll stream.

```
Select Rstream(E.vehicleId,
               2 * (V.numVehicles-50)
                 * (V.numVehicles-50)
               as toll)
From VehicleSegEntryStr [Now] as E,
     CongestedSegRel as C,
     SegVolRel as V
Where E.segNo = C.segNo and
      C.segNo = V.segNo
```

At any instant τ, the `Now` window on the stream `VehicleSegEntryStr` identifies the set of vehicles that have entered new segments at τ. This set of vehicles is joined with `CongestedSegRel` and `SegVolRel` to determine which vehicles have entered congested segments, and to compute tolls for such vehicles. Recall from Section 3 that the toll for a congested segment is given by the formula $2 \times (numvehicles - 50)^2$, where $numvehicles$ is the number of vehicles currently in the segment.

This query provides an example where the default `Unbounded` window would not yield the intended behavior if a window specification were omitted. In general, if a stream is joined with a relation in order to add attributes to and/or filter the stream, then a `Now` window on the stream coupled with an `Rstream` operator usually provides the desired behavior. □

Recall that the Linear Road specification in this paper is a simplified version of the original [4]. A CQL specification of the complete Linear Road benchmark as well as a number of other stream applications, such as network monitoring and online auctions [44], is available at [37].

8 Time Management

Recall from Sections 4 and 5 that our abstract semantics assumes a discrete, ordered time domain $\mathcal{T}$. Specifically, our continuous semantics is based on time logically advancing within domain $\mathcal{T}$. Conceptually, at time $\tau \in \mathcal{T}$ all inputs up to τ are processed and the output corresponding to τ (stream elements with timestamp τ or instantaneous relation at τ) is produced. In this section we briefly discuss how a DSMS might implement this semantics under realistic conditions. The topic is covered in much more depth in [38].

For exposition in the remainder of this section let us assume that relations are updated via timestamped relational update requests that arrive on a stream. Thus, without loss of generality we can focus on streams only. For a DSMS to produce output corresponding to a time $\tau \in \mathcal{T}$, it must have processed all input stream elements at least through τ. In other words, it must know at some "real" (wall-clock) time t that no new input stream elements with timestamp $\leq \tau$ will arrive after t. Making this determination is straightforward when all of the input streams are "alive" and their elements arrive in timestamp order. However, in many stream applications (including the Linear Road) input streams may be generated by remote sources, the network conveying the stream elements

[3] In this query, we assume that a vehicle does not exit the highway and re-enter within 30 seconds. We could handle this case by using an additional `Partition By` window.

to the DSMS may not guarantee in-order transmission, particularly across sources, and streams may pause and restart.

In the STREAM prototype our approach is to assume an additional "meta-input" to the system called *heartbeats*. A heartbeat consists simply of a timestamp $\tau \in \mathcal{T}$, and has the semantics that after arrival of the heartbeat 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 the heartbeats.
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 $t - D$.

Heartbeats are also important internally within the STREAM implementation of CQL, in order to communicate time-related information among different operators in a query plan. These and other details related to heartbeat generation can be found in [38].

8.1 Operations over Timestamps

Recall from Section 4 that the presence of timestamps is implicit, not part of the schema of a stream. Thus CQL does not permit direct references to timestamps in queries. We decided to make timestamps implicit for the following reasons:

1. Timestamps have certain properties (e.g., monotonicity) that we rely on in our semantics (and implementation). Therefore, we cannot permit queries to perform arbitrary transformations on timestamps, and we decided against making them explicit and monitoring the operations for violation of these properties.
2. Making timestamps implicit limits the operations performed on them, simplifying query plan generation and optimization.

If an application wishes to pose queries referring to timestamps explicitly, it can do so by simply mirroring the timestamp attribute in its stream schemas. The following example illustrates this point.

Example 13 We can add an explicit timestamp attribute to `PosSpeedStr`, resulting in the schema `PosSpeedStr(vehicleId,Speed,xPos,tstamp)`. When a vehicle `v` reports its position `x` and speed `s` at timestamp τ, the element $\langle$(`v, s, x,` τ)$, \tau\rangle$ arrives on `PosSpeedStr`. The following query computes the delay between the last two speed-position reports received from any vehicle:

```
Select Max(tstamp) - Min(tstamp)
From PosSpeedStr [Rows 2]
```

This query cannot be expressed through implicit timestamps only. □

9 Common Constructs

In this section, we illustrate a few constructs we found to appear frequently in CQL queries, primarily based on our experience with the Stream Query Repository [37].

9.1 Stream Filters:

A filter over a stream can be expressed in two ways: using an `Istream-Unbounded` window combination or an `Rstream-Now` window combination. Both of these were illustrated in Example 10. Note the `Istream-Unbounded` window combination is the default for streams whose window specification is omitted (recall Section 6.4).

9.2 Stream-Relation Joins:

When a stream is joined with a relation, it is usually most meaningful to apply a `Now` window over the stream, and an `Rstream` operator over the join result. Consider a stream `Item` of purchased items and a relation `PriceTable` of current item prices. The query:

```
Select Rstream(Item.id,
               PriceTable.price)
From Item [Now], PriceTable
Where Item.id = PriceTable.itemId
```

produces the streamed items with their current price appended. Using other types of windows or other relation-to-stream operators usually does not produce "correct" results. For example, the query:

```
Select Istream(Item.id,
               PriceTable.price)
From Item [Range Unbounded],
     PriceTable
Where Item.id = PriceTable.itemId
```

produces, along with new items, the (new) price for all previously purchased items whenever the price for an item changes.

9.3 Sliding-Window Joins:

Sliding-window joins of two streams is an operation that has received great deal of attention, e.g., [10, 22, 39]. If both streams in the join have keys (i.e., there are no duplicate tuples), this type of join can be expressed in CQL using an `Istream` operator. For example:

```
Select Istream(*)
From   S1 [Rows 5], S2 [Rows 10]
Where  S1.A = S2.A
```

is a sliding-window natural-join of `S1` and `S2` with a 5-tuple window on `S1` and a 10-tuple window on `S2`. A new tuple of `S1` produces an output join tuple if it joins with one of the last 10 tuples of `S2`; a new tuple of `S2` produces an output join tuple in a similar manner.

If either stream can have duplicates, the above query may not have the expected semantics of a sliding-window join. Suppose a new tuple of `S2` is identical to the tuple 10 positions earlier. Then the new `S2` tuple does not produce any join result tuples, even if it joins with one of the last 5 tuples of `S1`. The more general sliding-window join operation handling duplicates can be expressed in CQL, but it is somewhat more involved.

9.4 Streaming Aggregations:

Aggregation produces relations, not streams, in CQL since aggregations are relation-to-relation operators. We describe two commonly used ways of streaming an aggregated value:

1. *Stream the value of the aggregation whenever it changes:* This can be expressed using the `Istream` operator over the aggregation. For example:

   ```
   Select Istream(Count(*))
   From PosSpeedStr [Range 1 Minute]
   ```

   counts the speed-position reports over the previous minute, and streams the count whenever it changes.

2. *Stream the value of the aggregation periodically:* This can be expressed using windows with a slide parameter and `Istream`. For example:

   ```
   Select Istream(Count(*))
   From PosSpeedStr [Range 1 Minute
                     Slide 1 Minute]
   ```

   streams the number of speed-position reports over the last minute once every minute. A small subtlety with this query is that it will not stream the aggregation value if it remains unchanged from one minute to the next. Ensuring the value is streamed even when unchanged requires a more complex query.

10 Equivalences in CQL

In this section we briefly consider syntactic equivalences in the CQL language. As in any declarative language, equivalences can enable important query-rewrite optimizations, however the optimization process itself is not a central topic of this paper.

All equivalences that hold in SQL with standard relational semantics carry over to the relational portion of CQL, including join reordering, predicate pushdown, subquery flattening, etc. Furthermore, since any CQL query or subquery producing a relation can be thought of as a materialized view that is updated over time, all equivalences from materialized view maintenance [26] 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. In fact, this equivalence is incorporated into our `binary-join` physical query plan operator (Section 12).

Here we consider two equivalences based on streams: *window reduction* and *filter-window commutativity*. The identification of other useful stream-based syntactic equivalences is a topic of future work.

10.1 Window Reduction

The following equivalence can be used to rewrite any CQL query or subquery 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]

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 constraints over the streams [12].

[4] More generally, `Istream` and `Rstream` are equivalent over any relation R for which $R(\tau) \cap R(\tau - 1) = \emptyset$ for all τ.

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.

10.2 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]
      ≡
Select L From S [Range T] Where C
```

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 [33]. Note that the converse transformation might also be applied: We might prefer to move the filtering condition out of the window in order to allow the window to be shared by multiple queries with different selection conditions [33]. Finally note that filters and tuple-based windows generally do not commute.

11 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* [43], *Tribeca* [41], GSQL [21], and materialized views over relations with or without *chronicles* [29]. We also discuss *Aurora* [16], although it is difficult to compare CQL against Aurora because of Aurora's graphical, procedural nature.

At the end of this section we discuss our choice to define a language based on both relations and streams, instead of taking a stream-only approach. Included is a discussion of the stream-only query language of *TelegraphCQ* [18].

11.1 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*) [29] 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 $S_1 \bowtie_{S_1.A=S_2.B} S_2$ in CDM is equivalent to the CQL query:

```
Select Rstream(*)
From S1 [Now], S2 [Now]
Where S1.A = S2.B
```

The second class of operators takes a stream as input and produces 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.

11.2 Tapestry

Tapestry queries [43] 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.

11.3 Tribeca

Tribeca is based on a set of stream-to-stream operators and we have shown that all of the Tribeca operators specified in [41] can be expressed in CQL; details are omitted. 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.

11.4 Gigascope

GSQL is a SQL-like query language developed for *Gigascope*, a DSMS designed specifically for network monitoring applications [21]. GSQL is a stream-only language, but relations can be created and manipulated using user-defined functions. Over streams GSQL's primary operators are selection, join, aggregation, and *merge*. Constraints on join and aggregation ensure that they are nonblocking: a join operator must contain a predicate involving an "ordered" attribute from each of the joining streams, and an aggregation operator must have at least one grouping attribute that is ordered. (Ordered attributes are generalizations of CQL timestamps.)

The four primary operations in GSQL can be expressed in CQL: Selection is straightforward. The GSQL `Merge` operator can be expressed using `Union` in CQL. The GSQL join operator translates to a sliding-window join with an `Istream` operator in CQL. Finally, although it is nontrivial to express GSQL aggregation in CQL (requiring grouping and aggregation, projection, and join), it always is expressible; details are omitted.

11.5 Aurora

Aurora queries are built from a set of eleven operator types [16]. 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 set by the user. For example, these parameters can direct the operator to take certain action if no stream elements have arrived for T wall-clock seconds, making the semantics dependent on stream arrival and processing rates.

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

11.6 Stream-Only Query Language

Our abstract semantics and therefore CQL distinguish two fundamental data types, relations and streams. At the end of this section we outline several motivations for choosing our dual approach over a purely stream-based approach. Nevertheless, it is worth noting that we can always derive a stream-only language L_s from our language L (either CQL or another instantiation of our abstract semantics) as follows.

1. 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 $\texttt{Rstream}(O(S_1\ [\texttt{Now}], \ldots, S_n\ [\texttt{Now}]))$.
2. Corresponding to each stream-to-relation 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 $\texttt{Rstream}(S[W])$.
3. There are no operators in L_s corresponding to relation-to-stream operators of L.

L and L_s have essentially the same expressive power. 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 6.3). Second, replace every input relation R_i in Q' with $\texttt{Rstream}(R_i)$. Finally, replace every relation-to-relation and stream-to-relation operator in Q with its L_s equivalent according to the definitions above. As it turns out, the language L_s derived from CQL is quite similar to the stream-to-stream approach being taken in *TelegraphCQ* [18].

We chose our dual approach over the stream-only approach for at least three reasons:

1. Reiterating goal #1 from Section 1, we wanted to exploit the wide body of understanding and work on the existing relational model to the extent possible.
2. Our experience with a large number of queries [37] suggests that the dual approach results in more intuitive queries than the stream-only approach. As illustrated in our Linear Road examples (Section 7), even applications with purely stream-based input and output specifications may include fundamentally relational components.
3. Having both relations and streams cleanly generalizes materialized views, as discussed in detail in Section 11.1.

Note that the *Chronicle Data Model* [29] discussed in Section 11.1 also takes an approach similar to ours—it supports both chronicles (closely related to streams) and materialized views (relations).

12 CQL Implementation in STREAM

In this section we describe the query plans and execution strategies we use in our implementation of CQL as the declarative query language in the STREAM prototype Data Stream Management System [1]. Section 12.1 describes the physical representation of streams and relations within query plans, Section 12.2 describes query plan structure, Section 12.5 enumerates the operators used in query plans, and Section 12.6 briefly discusses query plan generation. (Details of STREAM's adaptive approach to query optimization appear in other papers [10, 11, 13].) Finally, Section 12.7 presents STREAM's graphical user interface for viewing, monitoring, and altering query plans.

Overall, STREAM's query execution strategy is based heavily on *incremental processing*: stream-to-relation operators transform new elements on their input streams to changes in their output relations; relation-to-relation operators transform changes in their input relations to changes in their output relations; relation-to-stream operators transform changes in their input relations to new elements in their output streams. All the components of STREAM's architecture—the operators, operator state, and internal representations of relations and streams—are designed to facilitate incremental processing. Our execution strategy is somewhat related to work on incremental view maintenance [26, 27], but there are important differences: In STREAM timestamps play an important role in the generation and processing of increments, and there are several new operators in STREAM that are not part of traditional view maintenance.

12.1 Internal Representation of Streams and Relations

Recall the formal definitions of streams and relations from Section 4. A stream is a bag of tuple-timestamp pairs, which can be represented as a sequence of timestamped tuple "insertions." A relation, which is a time-varying bag of tuples, can also be represented as a sequence of timestamped tuples, except now we have both *insertion tuples* and *deletion tuples* to capture the changing state of the relation.

STREAM exploits this similarity between streams and relations and uses a common physical representation for both of them: sequences of *tagged tuples*. The tuple part contains a value for each attribute in the schema of the stream or relation; the tag contains a timestamp and indicates whether the tuple is an insertion or a deletion. The tagged-tuple sequences are append-only and are always in nondecreasing order by timestamp. (Base streams and relation updates that arrive out of timestamp order can be converted into sorted sequences as described in Section 8, and operators always emit sorted sequences.)

12.2 STREAM Query Plans

When a continuous query specified in CQL is registered with the STREAM system it is compiled into a query plan. The query plan is merged with existing query plans whenever possible, in order to share computation and state. Plan generation itself is not a focus of this paper, although we discuss it again briefly in Section 12.6. Each query plan runs continuously and is composed of three different types of components: *operators*, *queues*, and *synopses*.

12.2.1 Operators Each query plan operator reads from one or more input queues, processes the input based on its semantics, and writes its output to an output queue. Section 12.5 describes the current set of operators supported in STREAM. Since queues (described next) encode both streams and relations, query plan operators can implement all three operator types in our abstract semantics and CQL: stream-to-relation, relation-to-relation, and relation-to-stream.

It is important to distinguish between the *logical operators* of the previous sections, which are used primarily to specify precise query semantics, from the *physical operators* that occur in query plans. As is typical in all DBMS's, physical operators are related to logical operators, but there exists no one-one correspondence between them. In the rest of this section, operators refer to physical operators.

12.2.2 Queues A queue connects its input operator O_I to its output operator O_O. At any time a queue contains a (possibly empty) sequence representing a portion of a stream or relation in our physical representation as described in Section 12.1. The contents that pass through the queue over time correspond to the stream or relation produced by O_I. The queue buffers the insertions and deletions in the output of O_I until they are processed by O_O.

12.2.3 Synopses Synopses store the intermediate state needed by continuous query plans. In our query plans a synopsis is always "owned" by a single operator O, and the state the synopsis contains is that needed for future evaluation of O. For example, to perform a windowed join of two streams, the join operator must have access to all tuples in the current window on each input stream. Thus, a join operator maintains one synopsis (e.g., a hash table) for each of the joined inputs. On the other hand, operators such as selection and duplicate-preserving union do not require a synopsis since they do not require saved state.

The most common use of a synopsis in our system is to materialize the current bag of tuples of a relation, such as the contents of a sliding window, or of a relation derived by a subquery. Synopses also may be used to store a summary of the tuples in a stream or a relation for approximate query answering. Example synopsis types for this purpose are *reservoir samples* [46] over streams, and *Bloom filters* [15].

12.2.4 Memory management Currently queues and synopses are stored entirely in memory, although we are in the process of implementing the capability for them to spill to disk. A common pool of memory is allocated to queues and synopses on demand at the page granularity. To minimize copying and proliferation of tuples, all tuple data is stored in synopses and is not replicated. Queues contain references to tuple data within synopses, along with tags containing a timestamp and an insertion/deletion indicator. In addition, some synopses are simply "stubs" that point to data in other synopses, as discussed in Section 12.4.

12.3 Example Query Plans

Figure 5 illustrates a merged STREAM query plan for two continuous queries, Q_1 and Q_2, over input streams S_1 and S_2. Query Q_1 is a windowed-aggregate query: it maintains the maximum value of $S_1.A$ for each distinct value of $S_1.B$ over a 50,000-tuple sliding window on stream S_1. In CQL:

```
Q1: Select B, max(A)
    From   S1 [Rows 50,000]
    Group By B
```

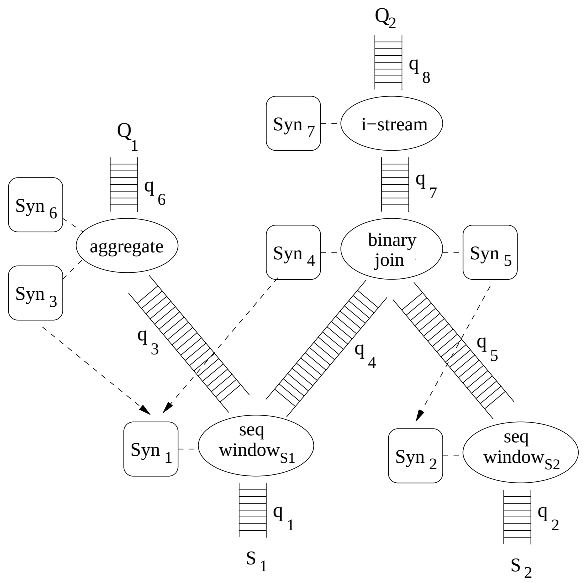


Fig. 5 STREAM continuous query plan for two queries

Query Q_2 streams the result of a sliding-window join over streams S_1 and S_2. The window on S_1 is a tuple-based window containing the last 40,000 tuples, while the window on S_2 is a 10-minutes time-based window. In CQL:

```
Q2: Select Istream(*)
    From   S1 [Rows 40,000],
           S2 [Range 600 Seconds]
    Where  S1.A = S2.A
```

The plan contains five operators, `seq-window`$_{S1}$, `seq-window`$_{S2}$, `aggregate`, `binary-join`, and `i-stream`, seven synopses *Syn*$_1$–*Syn*$_7$, and eight queues q_1–q_8. Operators `seq-window`$_{S1}$ and `seq-window`$_{S2}$ are stream-to-relation, `aggregate` and `binary-join` are relation-to-relation, and `i-stream` is relation-to-stream. We explain in some detail how each of the five operators behave.

– Operator `seq-window`$_{S1}$ is a sliding-window operator that reads input stream S_1's tuples from input queue q_1,[5] updates the *sliding-window synopsis Syn*$_1$, and outputs the insertions and deletions to this window (which is a relation) on both queues q_3 and q_4. *Syn*$_1$ always contains the 50,000 most recent tuples in stream S_1 that `seq-window`$_{S1}$ has processed, which is the larger of the two windows applied to S_1.

– Sliding-window operator `seq-window`$_{S2}$ processes input stream S_2's tuples from queue q_2. `seq-window`$_{S2}$ maintains synopsis *Syn*$_2$, which contains all tuples from S_2 whose timestamps are within the last 600 seconds. `seq-window`$_{S2}$ outputs insertions and deletions to this window (which is a relation) on queue q_5. Specifically, for an S_2 tuple s with timestamp τ, `seq-window`$_{S2}$ will produce an insertion of s into this window with timestamp τ, and a deletion of s from this window with timestamp $\tau + 601$ seconds. Recall from Section 12.1 that all insertions and deletions are emitted by the operator in nondecreasing timestamp order.

– Operator `aggregate` maintains the maximum value of $S_1.A$ for each distinct value of $S_1.B$ in its input relation, which is the 50,000-tuple sliding window on stream S_1. The aggregate values change based on insertions and deletions to the window, which recall are provided by operator `seq-window`$_{S1}$ on queue q_3. Operator `aggregate` maintains the current aggregation result in synopsis *Syn*$_6$, and it outputs insertions and deletions to this result on queue q_6. Because `max` is not incrementally maintainable—when the current maximum $S_1.A$ for an $S_1.B$ value leaves the window we may need to inspect the entire window to find the new maximum—operator `aggregate` maintains a second synopsis *Syn*$_3$ with sufficient information to maintain the maximums. It turns out that *Syn*$_3$ is simply a time-shifted version of *Syn*$_1$: *Syn*$_3$ contains the same 50,000 tuple window except it is "behind" by the tuples in q_3. Instead of duplicating information, the plan shares tuple data between *Syn*$_1$ and *Syn*$_3$, as indicated by the dotted arrow between them in Figure 5.

– Operator `binary-join` joins two input relations, which are sliding windows on streams S_1 and S_2. `binary-join` computes the join incrementally using the insertions and deletions to these windows provided on queues q_4 and q_5 respectively. `binary-join` processes tuples in nondecreasing timestamp order across both of its inputs. That is, each time it is ready to process an input tuple, it selects the tuple from queue q_4 or q_5 with the lowest timestamp. Logically, `binary-join` maintains a synopsis for each input: *Syn*$_4$ and *Syn*$_5$ for q_4 and q_5. An insertion (deletion) on q_4 is joined with *Syn*$_5$ to compute the corresponding insertions (deletions) to the join result which are output on queue q_7; similarly for q_5. These joins can be nested-loop or index joins depending whether we choose to build indexes on the join attributes in *Syn*$_4$ and *Syn*$_5$. Logically *Syn*$_4$ contains the 40,000 most recent tuples in stream S_1 that `binary-join` has processed. In our implementation, actual tuple data is shared rather than duplicated between synopses *Syn*$_4$ and *Syn*$_1$ as indicated by the dotted arrow between them in Figure 5. Similarly, the plan shares tuple data between *Syn*$_5$ and *Syn*$_2$. Furthermore, the subplan rooted at `seq-window`$_{S1}$ is shared by `aggregate` and `binary-join`, thus sharing the window computation on S_1 between queries Q_1 and Q_2. Our query plans always perform window sharing on a stream, with a single physically-stored synopsis automatically containing all stream elements needed by any window on that stream; see Section 12.5.1.

– Operator `i-stream` converts the relation produced by the join into a stream of relation insertions using the se-

[5] In reality, a special operator `stream-shepherd` handles incoming streams, placing the elements onto corresponding stream input queues; see Section 12.5.4.

mantics of the `Istream` operator from Section 6.3. Since `i-stream` receives the insertions and deletions to the join result in queue q_7, it might appear that `i-stream` can simply pass along all insertions and drop all deletions. However, if a tuple happens to be both inserted and deleted from the join result with the same timestamp, then for correct semantics `i-stream` must detect this case and not pass along the insertion. (This case can occur, for example, when the window on S_1 slides and the tuple values that enter and leave the window happen to be identical.) To handle this case correctly, `i-stream` buffers input insertions for a timestamp τ in synopsis S_7 until it knows that it will see no further tuples with that timestamp (recall Section 8).

It is important for timestamps on the tuples output by a query operator to properly reflect our semantics as specified earlier in this paper. In most operators, one or more output tuples are produced as the result of an input tuple being consumed, and it turns out our semantics is maintained correctly by copying the input tuple's timestamp to the output tuple. For example, when an input tuple with timestamp τ causes a tuple-based window to slide, both the insert and the delete generated by the slide are timestamped τ. When join results are produced from a new input tuple with timestamp τ, the new join results have timestamp τ, which always is the later of the timestamps on the two joining tuples since we process join inputs in nondecreasing timestamp order across inputs. Aggregation is similar: a new input tuple with timestamp τ produces one or more insertions and deletions to the aggregate result also timestamped τ. Time-based windows are trickier, as illustrated above. For a time-based window of size T, a stream tuple s with timestamp τ generates an inserted tuple with timestamp τ and generates a deleted tuple with timestamp $\tau + T + 1$, where the timestamp arithmetic, and particularly the meaning of "$+1$" depends on the time domain to which the timestamps belong. Recall from Section 4 that our semantics assumes a discrete and ordered time domain, so here "$+1$" denotes an increment in this domain, e.g., one second in the *Datetime* domain.

The execution of query plans is controlled by a global *scheduler*, which currently runs in the same thread as all of the query operators in the system. Each time the scheduler is invoked, it selects an operator to execute and calls a specific procedure defined for that operator, passing as a parameter the maximum number of input tuples that the operator should process before returning control to the scheduler. We currently use a simple round-robin scheduler, but we intend to incorporate a more appropriate scheduling algorithm based on recent research [7].

12.4 A Note on Synopses

From the example STREAM query plan of Figure 5 we see that our query plans tend to be overloaded with synopses. For example, the plan for the windowed-join query Q_2 in Figure 5 uses five synopses—one each for the two `seq-window` operators, two for the `binary-join` operator, and one for the `i-stream` operator. Our technique of generating numerous synopses made it much easier to implement a plan generation algorithm that works for arbitrary CQL queries. We could add a post-processing step that traverses query plans and physically merges some synopses. However, our approach so far has been to leave all the synopses in place, but make many of them logical "stubs" that primarily point into other synopses. For example, the plan for query Q_2 in Figure 5 materializes only three synopses instead of five since the two synopses of the `binary-join` operator are shared with the synopses of the corresponding `seq-window` operators. Synopses *Syn*$_4$ and *Syn*$_5$ in Figure 5 are stubs pointing to synopses *Syn*$_1$ and *Syn*$_2$ respectively.

Currently, we have implemented only simple strategies for sharing synopsis state across different operators. As described above, our most common strategy is to share state between an operator that produces a relation (e.g., a sliding window) and an operator that accesses the relation (e.g., a join over the window). In related work [6], we present more sophisticated strategies for sharing state, specifically algorithms for sharing state across different aggregations over sliding windows. These algorithms could be incorporated into the STREAM prototype although we have not yet done so.

12.5 STREAM Query Operators

In this section we specify all of the query operators currently implemented in the STREAM prototype and used in query plans. Every operator is either a *data operator* or a *system operator*. Data operators are the main data processing units. They can be categorized into three classes—stream-to-relation, relation-to-relation, and relation-to-stream—just like the logical operators of CQL. However, we note again that all operators perform incremental processing, so effectively they are all performing a type of stream-to-stream processing.

All five operators discussed in the previous section were data operators. System operators isolate the data operators from lower-level issues such as asynchronous and out-of-order arrival of streams, *load shedding* [9,42], and external stream data formats. The operators are listed in Table 1.

12.5.1 Stream-to-Relation Operators The sliding-window operator `seq-window` is the only stream-to-relation operator implemented in STREAM. It supports the tuple-based, time-based, and partitioned window specifications introduced in Section 6.1. (STREAM does not yet support windows with a slide parameter, as described in Section 6.1.4.) As illustrated in Figure 5, STREAM instantiates one sliding-window operator, denoted `seq-window`$_S$, for each stream S that has at least one continuous query specifying a window over S. Operator `seq-window`$_S$ has one input queue providing the tuples in S in nondecreasing timestamp order. For each sliding window on S used in a query plan, `seq-window`$_S$ maintains an output queue containing the insertions and deletions

Name	Operator Type	Description
`seq-window`	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
`select`	relation-to-relation	Filters tuples based on predicate(s)
`project`	relation-to-relation	Duplicate-preserving projection
`binary-join`	relation-to-relation	Joins two input relations
`mjoin`	relation-to-relation	Multiway join from [45]
`union`	relation-to-relation	Bag union
`except`	relation-to-relation	Bag difference
`intersect`	relation-to-relation	Bag intersection
`antisemijoin`	relation-to-relation	Antisemijoin of two input relations
`aggregate`	relation-to-relation	Performs grouping and aggregation
`duplicate-eliminate`	relation-to-relation	Performs duplicate elimination
`i-stream`	relation-to-stream	Implements `Istream` semantics
`d-stream`	relation-to-stream	Implements `Dstream` semantics
`r-stream`	relation-to-stream	Implements `Rstream` semantics
`stream-shepherd`	system operator	Handles input streams arriving over the network
`stream-sample`	system operator	Samples specifi ed fraction of tuples
`stream-glue`	system operator	Adapter for merging a stream-producing view into a plan
`rel-glue`	system operator	Adapter for merging a relation-producing view into a plan
`shared-rel-op`	system operator	Materializes a relation for sharing
`output`	system operator	Sends results to remote clients

Table 1 Operators in STREAM query plans

to that window. By using a single window operator per stream S, the system is able to share the computation and memory required for window maintenance across all queries referencing S.

12.5.2 Relation-to-Relation Operators Each relation-to-relation operator processes insertions and deletions in timestamp order from one or more input queues, computes its output incrementally, and writes the output insertions and deletions to an output queue in timestamp order. As listed in Table 1, STREAM supports relation-to-relation operators corresponding to all standard relational operators. Notice that STREAM supports two join operators: `binary-join`, a binary join operator as illustrated in Figure 5, and *mjoin*, the multiway join operator proposed in [45]. Consequently, a multiway join can be processed in two ways—using `mjoin` which does not materialize intermediate results [45], or using a tree of binary joins. Deciding which strategy to use is a query optimization issue not covered in this paper.

12.5.3 Relation-to-Stream Operators STREAM supports three relation-to-stream operators—`i-stream`, `d-stream`, and `r-stream`—corresponding to the `Istream`, `Dstream`, and `Rstream` operators defined in Section 6.3. Each operator processes insertions and deletions in timestamp order from its single input queue, and writes the output stream insertions to an output queue in timestamp order. The `i-stream` operator was described in Section 12.3, and the `d-stream` operator is very similar. The `r-stream` operator maintains the entire current state of its input relation in a synopsis and outputs all of the tuples as insertions at each time step. While this may appear expensive, recall from Section 6.3 that the `Rstream` operator is used most commonly with `Now` windows, so the "current state of the relation" is generally very small.

12.5.4 System Operators The primary purpose of the system operators in STREAM is to isolate the data operators from dealing with various lower-level issues. For completeness and for understanding the STREAM query plans in Section 12.7, we briefly discuss some of these operators. The `stream-shepherd` operator for a stream S serves as the source of S to all query plans accessing S: primarily it receives tuples arriving asynchronously over the network, transforms them to STREAM's internal representation, and writes them to the appropriate input queues. In the future, this operator also will be responsible for buffering input tuples for proper ordering, generating heartbeats when source applications don't provide them [38], and performing load shedding under overload [9].

The `stream-sample` operator currently is used only for system-managed load shedding [9], although we also intend to implement a `sample` clause in our query language [8]. `stream-sample` drops a specified fraction of stream tuples from its input queue based on a uniform random sample ("coin toss"). Other implemented system operators serve as materialization points for relations (`shared-rel-op`), enable plans for views to be merged into new query plans (`rel-glue` and `stream-glue`), and send query results to remote clients (`output`).

12.6 Query Plan Generation

Most of the CQL language is operational in the STREAM system. Our query plan generator is fairly simple, using hard-coded heuristics to generate initial query plans. Our current approach is to generate simple plans, which are then monitored and possibly restructured automatically using STREAM's adaptivity component, *StreaMon* [10, 11, 13]. Given the promise of this approach, currently we do not expect to build a complex query optimizer for plan generation. However, we do apply heuristics to generate our initial simple plan, including:

1. Push selections below joins.
2. Maintain and use indexes for synopses on `binary-join`, `mjoin`, and `aggregate` operators.
3. Share synopses and operators within query plans whenever possible.[6]

We are actively moving toward one-time and dynamic cost-based optimization of CQL queries. Since CQL uses SQL as its relational query language, we can also leverage many of the one-time optimization techniques used in traditional relational systems. In addition, we are exploring adaptive query optimization techniques that are coarser-grained than *Eddies* (as used in the *Telegraph* project [18]). Our approach relies on two interacting components: a *monitor* that captures properties of streams and system behavior, and an *optimizer* that can reconfigure query plans and resource allocation as properties change over time.

12.7 Example STREAM Plans

Lastly, we present two snapshots of query plans taken from STREAM's graphical query plan visualizer. Through the visualization interface users can inspect the plan generated for a continuous query as soon as the query is registered, can monitor plan behavior during execution, and can even alter plan structure and attributes of plans such as memory allocation, for the purpose of experimentation.

Figure 6 shows the query plan for the following simple illustrative CQL query over streams S1 and S2:

```
Select S2.name, max(S1.num)
From   S1 [Rows 50,000],
       S2 [Rows 50,000]
Where  S1.name <= 'i' and
       S1.num = S2.num
Group By S2.name
```

[6] So far the sharing techniques implemented in our system are simple, but this is an important area of future research. For example, more sophisticated techniques such as [6, 20, 32] should be applicable.

This query is a windowed two-way join with a filter predicate on $S1$, followed by an aggregation. The system operators used in the query plan in Figure 6 are `stream-shepherd` operators for streams $S1$ and $S2$ and an `output` operator to send the query result continuously to the client that submitted the query. The data operators are `seq-window` for $S1$ and $S2$, `select`, `binary-join`, `aggregate`, and (duplicate-preserving) `project`. Notice that synopses are shared between the window operators and the join, and the selection has been pushed below the join. The selection cannot be pushed below the window operator since in general tuple-based windows and selection do not commute. The `binary-join` operator materializes its output relation in a synopsis that is shared with the `aggregate` operator, similar to our example in Section 12.3.

Figure 7 shows the complete query plan for the `TollStr` Linear Road query from Section 7, which incorporates plans for all subqueries that were used to write this query in CQL. We do not expect readers to examine every detail of this complex query plan.

13 Conclusions

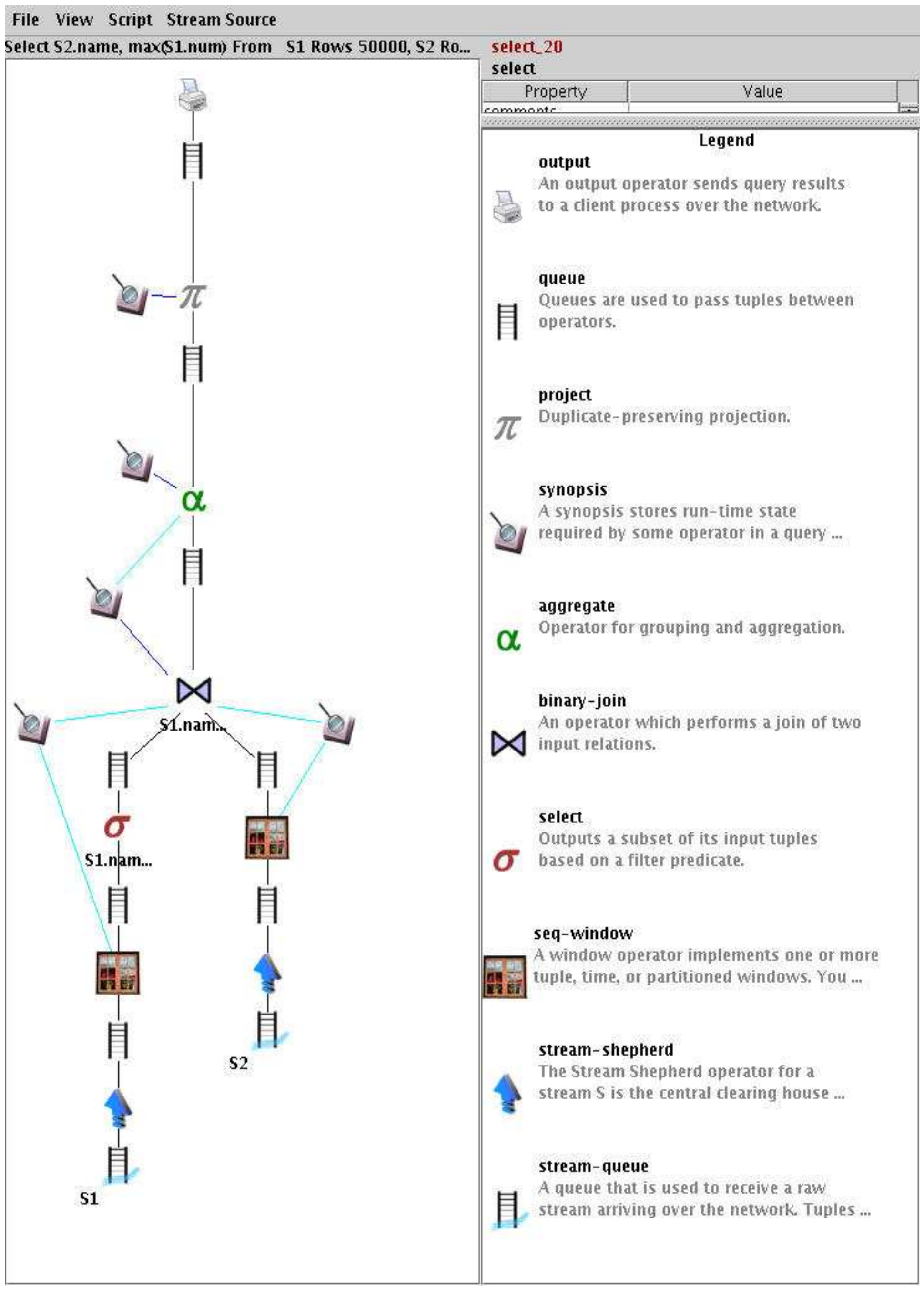
This paper specified the CQL language for continuous queries over data streams, including a formal abstract semantics on which CQL is based. Most of the CQL language is operational in the STREAM prototype Data Stream Management System, including the "linear road" benchmark used as examples throughout this paper. This paper also describes the structure and implementation of query execution plans for CQL in the STREAM system.

The STREAM system is available for experimentation over the Internet: For each user, a dedicated server is started on a machine at Stanford and a client is started on the user's machine. Through the graphical interface as depicted in Figures 6 and 7, users may register streams and continuous queries, and view the streamed (or stored) results. Users may also inspect and alter query plans, and may perform visual system monitoring through "introspection": Query components write statistics (such as throughput, selectivity, etc.) onto a special *system stream*. Graphical system monitors obtain their plotted values by registering standard CQL queries on the special system stream.

Please try out the system, available from the STREAM home page at `http://www-db.stanford.edu/stream/`.

Acknowledgments

We are especially grateful to Keith Ito for his significant contributions to implementing the STREAM prototype. We are also grateful to STREAM participants Brian Babcock, John Cieslewicz, Mayur Datar, Rajeev Motwani, Itaru Nishizawa,


File View Script Stream Source
Select S2.name, max(S1.num) From S1 Rows 50000, S2 Ro...
select_20
select
Property
Value
comments
S1.nam...
S1.nam...
S1
S2

Legend

output
An output operator sends query results to a client process over the network.

queue
Queues are used to pass tuples between operators.

project
Duplicate-preserving projection.

synopsis
A synopsis stores run-time state required by some operator in a query ...

aggregate
Operator for grouping and aggregation.

binary-join
An operator which performs a join of two input relations.

select
Outputs a subset of its input tuples based on a filter predicate.

seq-window
A window operator implements one or more tuple, time, or partitioned windows. You ...

stream-shepherd
The Stream Shepherd operator for a stream S is the central clearing house ...

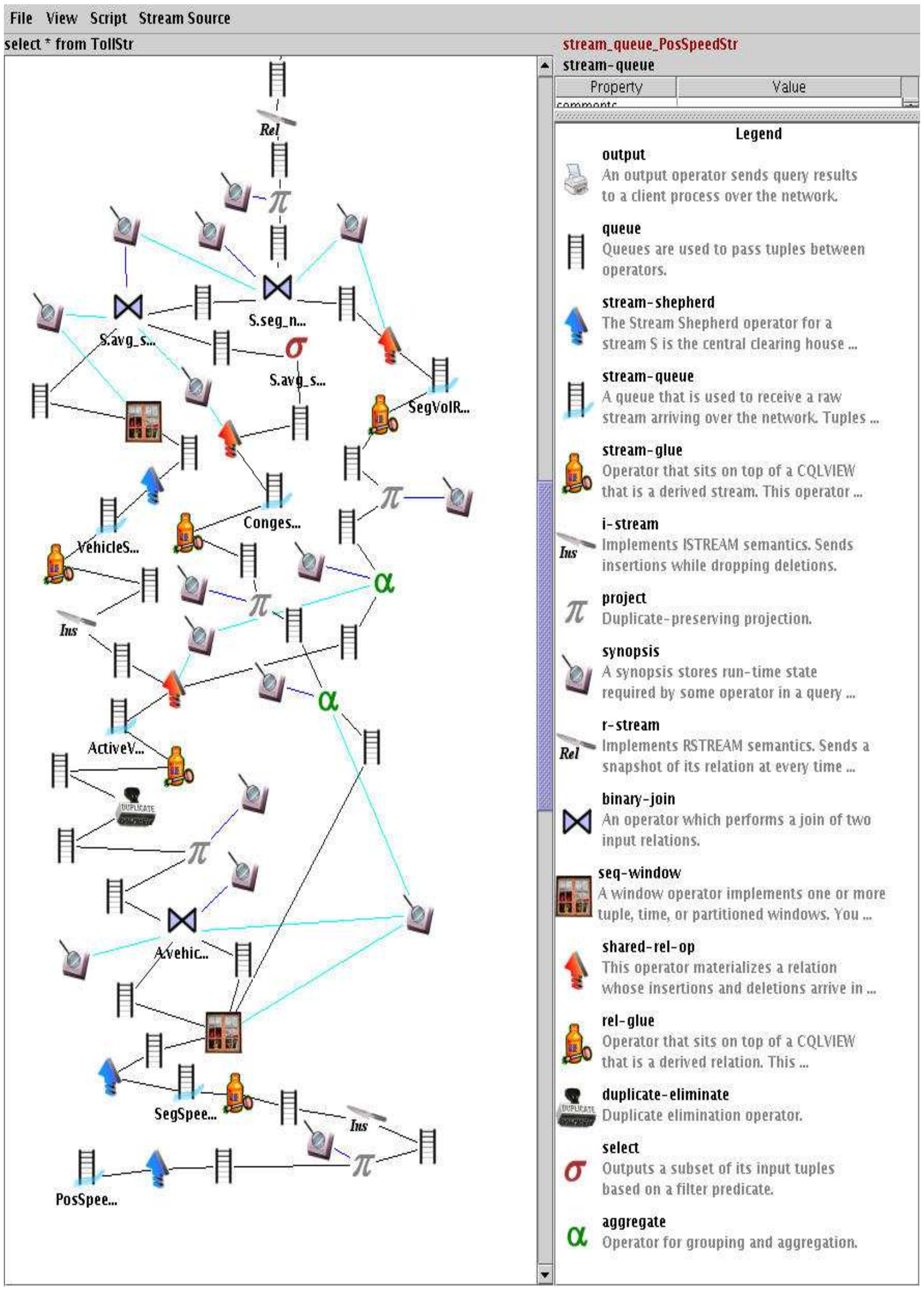
stream-queue
A queue that is used to receive a raw stream arriving over the network. Tuples ...

Fig. 6 Plan for aggregate query over join

Utkarsh Srivastava, and others who have influenced the development of the CQL language and STREAM query processor.

References

1. A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, I. Nishizawa, J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, page 665, June 2003. Demonstration description.
2. A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. *ACM Trans. on Database Systems*, 29(1):162–194, Mar. 2004.
3. A. Arasu, S. Babu, and J. Widom. CQL: A language for continuous queries over streams and relations. In *9th Intl. Workshop on Database Programming Languages*, pages 1–11, Sept. 2003.
4. A. Arasu, M. Cherniak, et al. Linear road: A stream data management benchmark. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 480–491, Sept. 2004.
5. A. Arasu and J. Widom. A denotational semantics for continuous queries over streams and relations. *SIGMOD Record*,



select * from TollStr

stream_queue_PosSpeedStr

stream-queue

Legend

output An output operator sends query results to a client process over the network.

queue Queues are used to pass tuples between operators.

stream-shepherd The Stream Shepherd operator for a stream S is the central clearing house ...

stream-queue A queue that is used to receive a raw stream arriving over the network. Tuples ...

stream-glue Operator that sits on top of a CQLVIEW that is a derived stream. This operator ...

i-stream Implements ISTREAM semantics. Sends insertions while dropping deletions.

project Duplicate-preserving projection.

synopsis A synopsis stores run-time state required by some operator in a query ...

r-stream Implements RSTREAM semantics. Sends a snapshot of its relation at every time ...

binary-join An operator which performs a join of two input relations.

seq-window A window operator implements one or more tuple, time, or partitioned windows. You ...

shared-rel-op This operator materializes a relation whose insertions and deletions arrive in ...

rel-glue Operator that sits on top of a CQLVIEW that is a derived relation. This ...

duplicate-eliminate Duplicate elimination operator.

select Outputs a subset of its input tuples based on a filter predicate.

aggregate Operator for grouping and aggregation.

Fig. 7 Plan for `TollStr` query

33(3):6–12, Sept. 2004.

6. A. Arasu and J. Widom. Resource sharing in continuous sliding-window aggregates. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 336–347, Sept. 2004.
7. B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain : Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 253–264, June 2003.
8. B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proc. of the 21st ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 1–16, June 2002.
9. B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *Proc. of the 20th Intl. Conf. on Data Engineering*, pages 350–361, Mar. 2004.
10. S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 407–418, June 2004.
11. S. Babu, K. Munagala, J. Widom, and R. Motwani. Adaptive caching for continuous queries. In *Proc. of the 21st Intl. Conf. on Data Engineering*, 2005.
12. S. Babu, U. Srivastava, and J. Widom. Exploiting k-constraints to reduce memory overhead in continuous queries over data

streams. *ACM Trans. on Database Systems*, 31(3), Sept. 2004.
13. S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004. Demonstration description.
14. Daniel Barbará. The characterization of continuous queries. *Intl. Journal of Cooperative Information Systems*, 8(4):295–323, Dec. 1999.
15. B. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.
16. D. Carney, U. Centintemel, et al. Monitoring streams - a new class of data management applications. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 215–226, Aug. 2002.
17. D. Carney, U. Centintemel, et al. Operator scheduling in a data stream manager. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pages 838–849, Sept. 2003.
18. S. Chandrasekharan, O. Cooper, et al. TelegraphCQ: Continuous datafbw processing for an uncertain world. In *Proc. of the 1st Conf. on Innovative Data Systems Research*, pages 269–280, Jan. 2003.
19. S. Chandrasekharan and M. J. Franklin. Streaming queries over streaming data. In *Proc. of the 28th Intl. Conf. on Very Large Data Bases*, pages 203–214, Aug. 2002.
20. 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.
21. C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, June 2003.
22. A. Das, J. Gehrke, and M. Reidewald. Approximate join processing over data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 40–51, June 2003.
23. A. Dobra, M. N. 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, June 2002.
24. J. Gehrke. Special issue on data stream processing. *IEEE Computer Society Bulletin of the Technical Comm. on Data Engg.*, 26(1), Mar. 2003.
25. L. Golab and M. T. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.
26. A. Gupta and I. S. Mumick. Maintenance of materialized views: Problems, techniques, and applications. *IEEE Computer Society Bulletin of the Technical Comm. on Data Engg.*, 18(2):3–18, June 1995.
27. A. Gupta, I. S. Mumick, and V. S. Subramanian. Maintaining views incrementally. In *Proc. of the1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 157–166, May 1993.
28. M. A. Hammad, M. J. Franklin, W. G. Aref, and A. K. Elmagarmid. Scheduling for shared window joins over data streams. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pages 297–308, Sept. 2003.
29. H. V. Jagadish, I. S. Mumick, and A. Silberschatz. View maintenance issues for the chronicle data model. In *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 113–124, May 1995.
30. Y.-N. Law, H. Wang, and C. Zaniolo. Query languages and data models for database sequences and data streams. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 492–503, Sept. 2004.
31. L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engg.*, 11(4):610–628, Aug. 1999.
32. S. Madden, M. A. Shah, J. M. 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.
33. R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the 1st Conf. on Innovative Data Systems Research*, pages 245–256, Jan. 2003.
34. 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.
35. G. Özsoyoglu and R. T. Snodgrass. Temporal and real-time databases: A survey. *IEEE Trans. on Knowledge and Data Engg.*, 7(4):513–532, Aug. 1995.
36. P. Seshadri, M. Livny, and R. Ramakrishnan. SEQ: A model for sequence databases. In *Proc. of the 11th Intl. Conf. on Data Engineering*, pages 232–239, Mar. 1995.
37. SQR – A Stream Query Repository. `http://www-db.stanford.edu/stream/sqr/`.
38. U. Srivastava and J. Widom. Flexible time management in data stream systems. In *Proc. of the 23rd ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems*, pages 263–274, June 2004.
39. U. Srivastava and J. Widom. Memory-limited execution of windowed stream joins. In *Proc. of the 30th Intl. Conf. on Very Large Data Bases*, pages 324–335, Sept. 2004.
40. Stanford Stream Data Management Project. `http://www-db.stanford.edu/stream`.
41. M. Sullivan. Tribeca: A stream database manager for network traffi c analysis. In *Proc. of the 22nd Intl. Conf. on Very Large Data Bases*, page 594, Sept. 1996.
42. N. Tatbul, U. Cetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, pages 309–320, Sept. 2003.
43. D. B. Terry, D. Goldberg, D. Nichols, and B. M. 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.
44. 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/`.
45. S. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pages 285–296, Sept. 2003.
46. J. Vitter. Random sampling with a reservoir. *ACM Trans. on Mathematical Software*, 11(1):37–57, Mar. 1985.
47. H. Wang and C. Zaniolo. ATLaS: A native extension of sql for data mining. In *Proc. of the 3rd SIAM Intl. Conf. on Data Mining*, May 2003.
48. H. Wang, C. Zaniolo, and C. Luo. ATLaS: A small but complete sql extension for data mining and data streams. In *Proc. of the 29th Intl. Conf. on Very Large Data Bases*, pages 1113–1116, Sept. 2003. Demonstration description.
49. J. Widom and S. Ceri, editors. *Active database systems: triggers and rules for advanced database processing*. Morgan Kaufmann, San Francisco, CA, 1996.