

The CQL Continuous Query Language: Semantic Foundations and Query Execution

by Arvind Arasu, Shivnath Babu, and Jennifer Widom

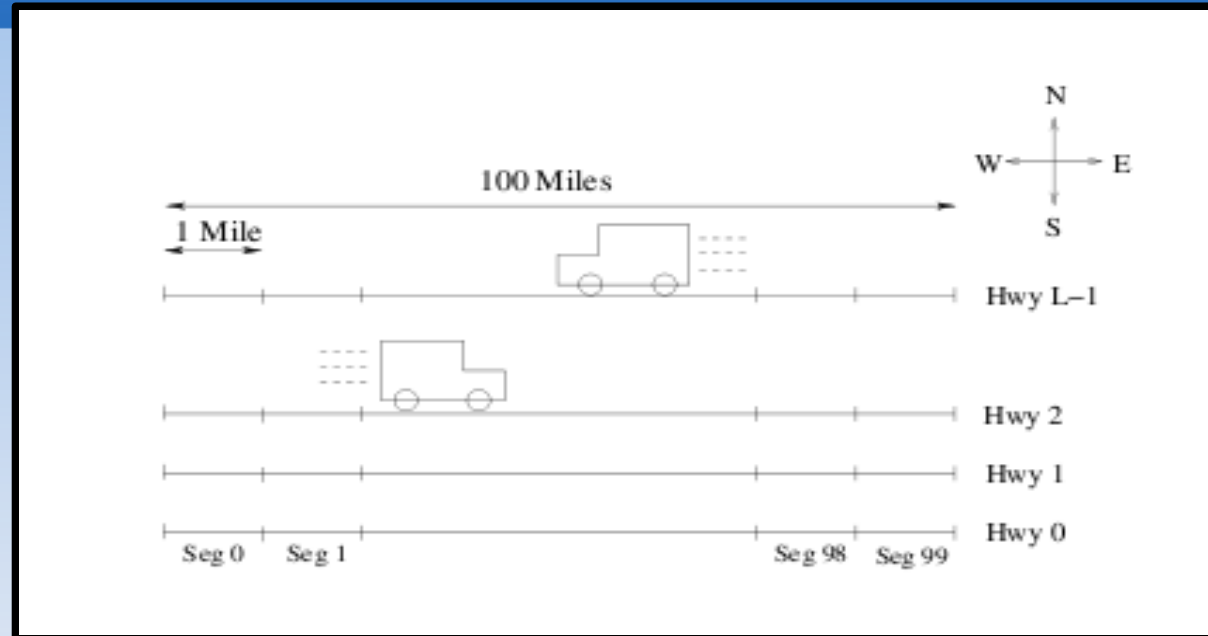
presented by Michael Mohler

Goals for CQL

- 1) Exploit well-understood relational semantics
- 2) Simple tasks should be compact and easy to write
- 3) Query plans should be modular and pluggable
- 4) Execution model should efficiently handle relations and streams
- 5) Architecture should be easily adaptable to allow for easy experimentation

Linear Road Example

There are L bidirectional highways numbered $0, \dots, L-1$. Each highway is 100 miles long and runs east-west. Each highway is composed of 100 1-mile segments.



For each car on the highway system, a sensor reports its current position, speed (in mph), and vehicleID to a central facility. This occurs once every 30 seconds.

Tolls are calculated for this highway system based upon the congestion of the highway segments. Specifically, a toll is given to a car whenever the car enters a congested segment and the amount of the toll is defined by:

$$\text{toll} = \text{basetoll} * (\text{numVehicles} - 150)^2$$

Streams

Definition 4.1 (Stream): A stream is a (possibly infinite) bag of elements $\{s,t\}$, where s is a tuple belonging to the schema of S , and $t \in T$ is the timestamp of the element.

The timestamp is NOT a part of the schema of S , and there MAY be duplicates depending on the implementation.

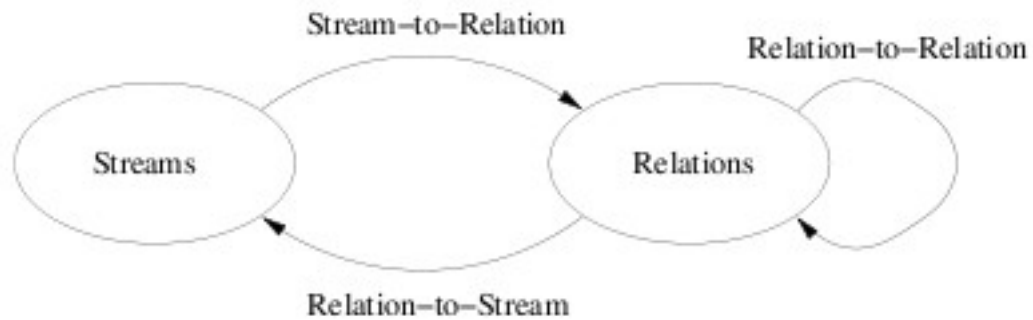
Base streams are source data streams, and derived streams are produced by operators in a query.

Relations

Definition 4.2 (Relation): A relation R is a mapping from 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 $t \in T$ denoted $R(t)$.

This differs from the standard definition of relation in that it is explicitly sensitive to time.

Operators



There are three types of query operators.

Stream-to-relation operators take a stream S as input and produce a relation R with the same schema as S .

Relation-to-Relation operators take one or more relations R_1, \dots, R_n as input and produce a relation R as output.

Relation-to-stream operators take a relation R as input and produce a stream S as output with the same schema as R .

Note: Stream-to-stream operators (filters) are not defined in the syntax and can only be performed by converting to an intermediary relation.

Stream-to-Relation Operators

Stream-to-Relation operators convert a portion of a stream (within a given window) into a relation.

The windows can be:

- Time-based
- Tuple-based
- Partitioned

Time-based sliding windows

For a stream S , a time-based window can be generated using the following syntax:

$$S \text{ [Range } T]$$

where T is some time (e.g. 30 seconds, 1 hour, 100 clock cycles). Specifically, the window produces a relation at time t containing all tuples of S with timestamp $\geq \max\{t - T, 0\}$.

Two special cases are defined as shortcuts:

$S \text{ [Now]} \rightarrow T=0$ (all tuples with $t=T$ are returned)

$S \text{ [Range Unbounded]} \rightarrow T=\infty$ (all tuples are returned)

Note: The syntax for T is not explicitly defined by the authors.

Tuple-based windows

A tuple-based window of S is created using the following syntax:

$$S \text{ [Rows } N\text{]}$$

which returns the N most recent tuples in S .

Tuple-based windows provide a shortcut `[Rows Unbounded]` which returns all tuples in S .

Note: If the stream contains fewer than N tuples, all tuples are returned. If timestamps are non-unique, ties are broken arbitrarily, making tuple-based windows non-deterministic.

Partitioned Windows

Partitioned windows simulate the “Group by” semantics of SQL. Consider the following command:

S [Partition By A1, ..., Ak, Rows N]

where A_1, \dots, A_k represents a subset of the features in the schema of S. This command returns the N most recent tuples for each combination of A_1, \dots, A_k .

Relation-to-Relation Operators

Relation-to-Relation operators closely mirror traditional SQL queries. Anywhere a traditional relation is referenced in an SQL query, a relation can be referenced in CQL.

Some relation-to-relation operators in CQL:

- Select
- Binary-join
- Union
- Intersect
- Aggregate

Relation-to-Stream Operators

CQL has three Relation-to-Stream operators:

- **Istream**: At time t , Istream contains all tuples of relation R that are in $R(t)$ but not in $R(t-1)$.
- **Dstream**: contains all tuples of R that are in $R(t-1)$ but not in $R(t)$.
- **Rstream**: contains all tuples in $R(t)$.

Note: Istream is most often used with Unbounded windows; Rstream is most often used with Now windows. Filtering can be accomplished equivalently using either method.

Default behavior of CQL

Some notes about shortcuts and default behavior of CQL:

Default Windows: If a stream is referenced (as in a FROM clause) with no window defined, an Unbounded window is applied by default.

Default R-to-S operators: Istream is automatically provided for monotonic relations.

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

The above query provides an unbounded window on PosSpeedStr and an Istream by default.

Queries for Linear Road Example

- PosSpeedStr(vehicleID,speed,xPos,dir,hwy) => base stream yields raw data
- SegSpeedStr(vehicleID,speed,segNo,dir,hwy) => converts raw position into segment
- ActiveVehicleSegRel(vehicleID,segNo,dir,hwy) => determines current segment for every active vehicle at time T [i.e. has a reading in the last 30s]
- VehicleSegEntryStr(vehicleID,segNo,dir,hwy) => stream indicating a vehicle has entered a segment
- CongestedSegRel(segNo,dir,hwy) => determines which segments are congested at time T
- SegVolRel(segNo,dir,hwy,numVehicles) => determines the number of vehicles in each segment at time T
- TollStr(vehicleID,toll) => stream indicating the toll given to a vehicle upon entering a congested segment

SegSpeedStr

PosSpeedStr(vehicleID,speed,xPos,dir,hwy) => raw data stream

SegSpeedStr(vehicleID,speed,segNo,dir,hwy) => converts raw position into segment

SegSpeedStr:

```
SELECT vehicleID, speed, xPos/5280 as segNo, dir, hwy  
FROM PosSpeedStr;
```

Note: Input stream is automatically converted to a relation using default [Range Unbounded] operator. Output is automatically converted to a stream using default istream(*) operator. The paper incorrectly lists the number of yards per mile (1760) instead of feet per mile (5280).

Transform: Stream-Relation-Stream (i.e. Filter)

ActiveVehicleSegRel

SegSpeedStr(vehicleID,speed,segNo,dir,hwy) => converts raw position into segment

ActiveVehicleSegRel(vehicleID,segNo,dir,hwy) => determines current segment for every active vehicle at time T [i.e. has a reading in the last 30s]

ActiveVehicleSegRel:

```
SELECT Distinct A.vehicleID, A.segNo, A.dir, A.hwy
FROM SegSpeedStr [Range 30 sec] as A,
     SegSpeedStr [Partition by vehicleID, Rows 1] as B
WHERE A.vehicleID = B.vehicleID;
```

Note: [Partition] operator is used here to ensure only one vehicleID exists in relation. [Range] operator indicates which are active.

Transform: Stream-Relation-Join

VehicleSegEntryStr

ActiveVehicleSegRel(vehicleID,segNo,dir,hwy) => determines current segment for every active vehicle at time T [i.e. has a reading in the last 30s]

VehicleSegEntryStr(vehicleID,segNo,dir,hwy) => stream indicating a vehicle has entered a segment

VehicleSegEntryStr:

```
SELECT istream(*)  
FROM ActiveVehicleSegRel;
```

Note: The ActiveVehicleSegRel relation triggers this stream whenever a vehicle enters a new segment.

Transform: Relation-Stream

CongestedSegRel

SegSpeedStr(vehicleID,speed,segNo,dir,hwy) => converts raw position into segment

CongestedSegRel(segNo,dir,hwy) => determines which segments are congested at time T

CongestedSegRel:

```
SELECT segNo, dir, hwy
FROM SegSpeedStr [Range 5 mins]
GROUP BY segNo, dir, hwy
HAVING Avg(speed) < 40;
```

Note: For every (segNo,dir,hwy) tuple, determine the average vehicle speed over the past 5 mins. If the average is less than 40, the segment is congested.

Transform: Stream-Relation-Relation

SegVolRel

`ActiveVehicleSegRel(vehicleID,segNo,dir,hwy)` => determines current segment for every active vehicle at time T [i.e. has a reading in the last 30s]

`SegVolRel(segNo,dir,hwy,numVehicles)` => determines the number of vehicles in each segment at time T

SegVolRel:

```
SELECT segNo, dir, hwy, count(vehicleID) as numVehicles
FROM ActiveVehicleSegRel
GROUP BY segNo, dir, hwy;
```

Note: Counts the number of vehicles in a given (segNo, dir, hwy) tuple at query time.

Transform: Relation-Relation

TollStr

VehicleSegEntryStr(vehicleID,segNo,dir,hwy) => stream indicating a vehicle has entered a segment

CongestedSegRel(segNo,dir,hwy) => determines which segments are congested at time T

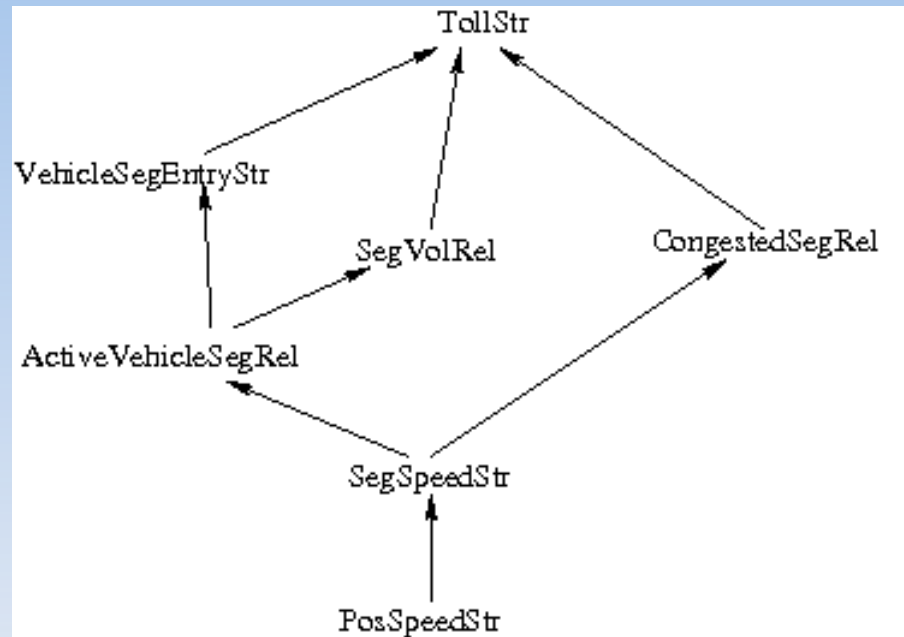
SegVolRel(segNo,dir,hwy,numVehicles) => determines the number of vehicles in each segment at time T

TollStr(vehicleID,toll) => stream indicating the toll given to a vehicle upon entering a congested segment

TollStr:

```
SELECT A.vehicleID,  
       basetoll*(C.numVehicles-150)*(C.numVehicles-150) as toll  
FROM VehicleSegEntryStr [Now] as A,  
     CongestedSegRel as B,  
     SegVolRel as C,  
WHERE A.segNo=B.segNo and A.segNo=C.segNo and  
       A.dir=B.dir and A.dir=C.dir and A.hwy=B.hwy and A.hwy=C.hwy;
```

TollStr (cont)



Note: TollStr joins two relations and a stream. Ultimately all input data comes from the single stream source PosSpeedStr.

Time Management

A real-world consideration is how time is handled in a continuous query system.

The semantics of CQL require a discrete, ordered time domain. Queries need to know when time t has passed so that they can be certain that they will not encounter another tuple with timestamp t . Sensor latency may be problematic.

Heartbeats serve as an indication to individual operators that a certain time has passed. Heartbeats are provided in three ways:

- By the system clock itself, if the system produces timestamps
- Sentinels in the source stream(s)
- By analyzing the network environment and determining an upper bound D on delay, the system can produce heartbeats for $t-D$.

Equivalence

For efficiency purposes, it may be beneficial for the system to automatically produce an equivalent but more-efficient version of a query. The authors provide two examples.

Example 1:

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

instead of

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

The [Range Unbounded] operator has to buffer the entire input stream, but Istream is only concerned with new data. [Now] can discard stream tuples once they are used.

Equivalence (cont)

Example 2:

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

instead of

Select L From S [Range T] Where C

By performing the Select operation before the Window operation, potentially far fewer tuples need to be stored in the Window materialization.

Note: This cannot be done with a tuple-based window because performing SELECT first may alter the number of tuples needed by the window.

Implementation: Data

In CQL's internal representation, relations and streams are represented in the same way – as a set of tagged tuples.

The difference lies in the fact that a tuple can be deleted from a relation, but it will always be part of a stream.

Each tuple in a relation/stream is therefore tagged with an Insert/Delete flag, and a timestamp (indicating when it was inserted/deleted).

Note: Stream tuples are always flagged “insert.”

Implem.: Operators and Queues

Operators:

- read data from one or more input queues (selecting the queue with the earliest timestamp first)
- processes the input, and
- writes its output to one or more output queues

Note: Operators must produce output in non-decreasing order of timestamp.

Queues:

- connect an input operator to an output operator
- serve as an input buffer for the output operator
- represents a portion of a stream or relation

Implem.: Synopses

Synopses:

- store the intermediate state needed by continuous query plans
- is “owned” by a single operator and stores what is needed for that operator to function

All data is stored in synopses. Queues contain only references to tuple data and tags.

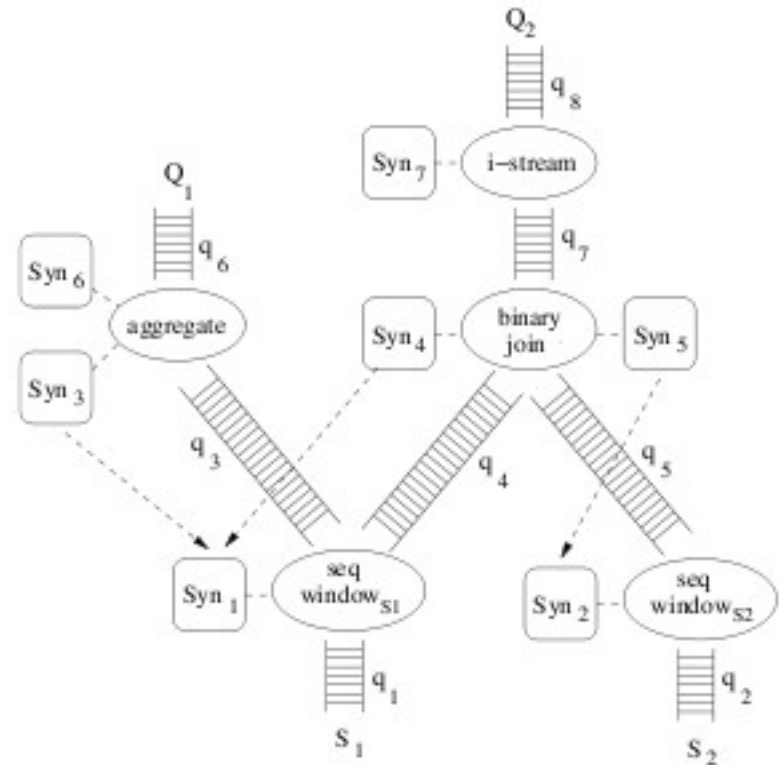
Note: Operators like selection and duplicate-preserving union do not need intermediate states and so do not require synopses.

Query Plan

The image to the right shows the query plan for the following set of queries:

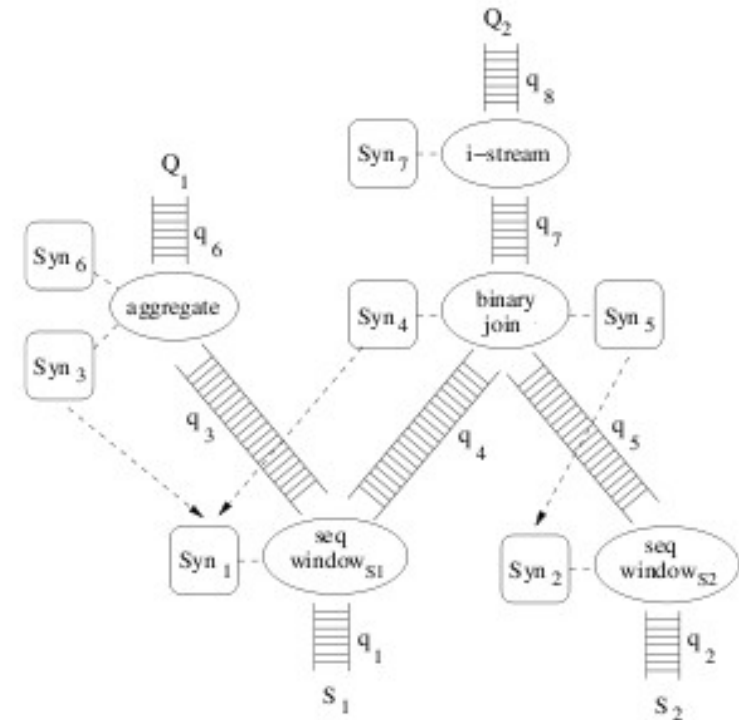
Q1: Select B, max(A)
From S1 [Rows 50000]
Group by B

Q2: Select Istream(*)
From S1 [Rows 40000], S2 [Range 600 Seconds]
Where S1.A = S2.A.



Query Plan: seq-window(s1)

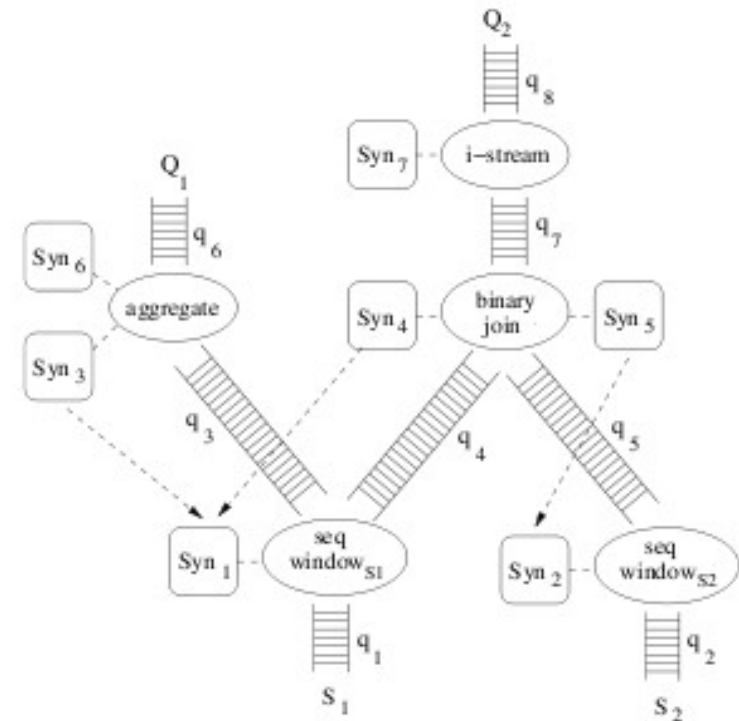
Tuple-Sliding Window:
seq-window(s1) handles the
50,000 most recent tuples for
Q1 and the 40,000 most recent
for Q2.



The synopsis Syn₁ contains the larger of the two windows to maximize data sharing. Output is written to both q₃ and q₄

Query Plan: seq-window(s2)

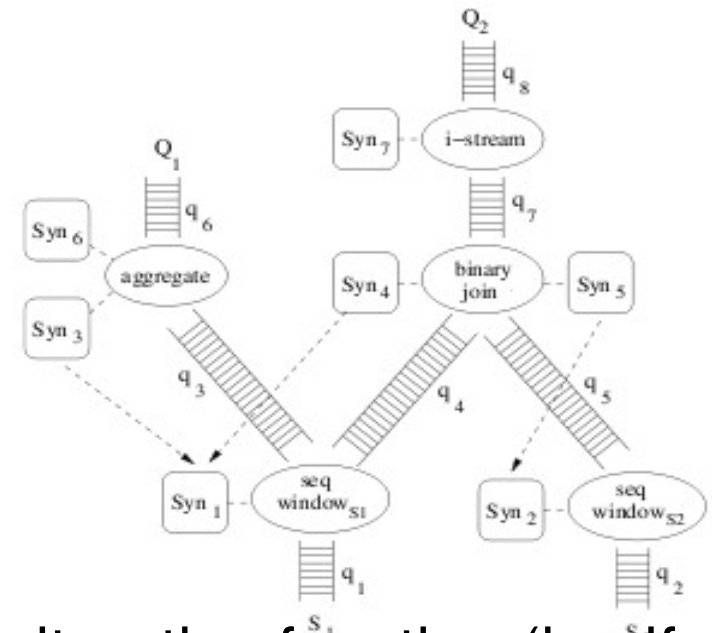
Time-Sliding Window:
seq-window(s2) handles all tuples on S2 from the last 10 minutes.



When an input tuple arrives from the stream at time T , seq-window(s2) stores it in Syn₂, writes an insertion record to q₅ for time T , and writes a deletion record for time $T+600$ which will be written to q₅ at time $T+600$.

Query Plan: aggregate

Aggregate $\max(S1.A)$:
For each value of $S1.B$, this operator finds the maximum value of $S1.A$ in the 50,000 tuple sliding window.

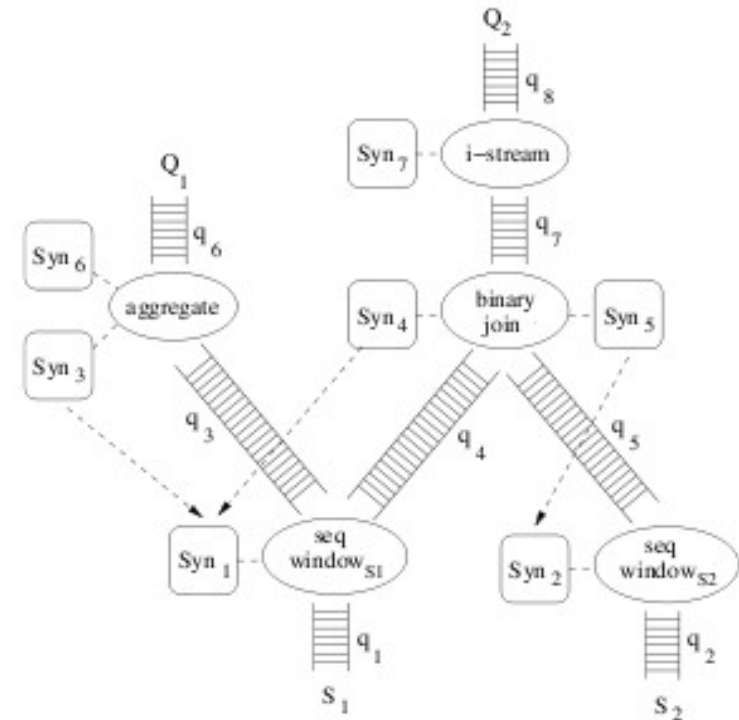


Since $\max(S1.A)$ over a window is not an iterative function (i.e. if a tuple leaves the window, it is impossible to reconstruct a new maximum without re-analyzing all tuples in the window) it requires data stored in **Syn₃**.

Syn₃ is a stub which refers back to **Syn₁**. This operator shares data with the window operator to reduce unnecessary copying. **Syn₆** stores the current state of the relation. Insertions and deletions are written to the output queue q_6 .

Query Plan: binary-join

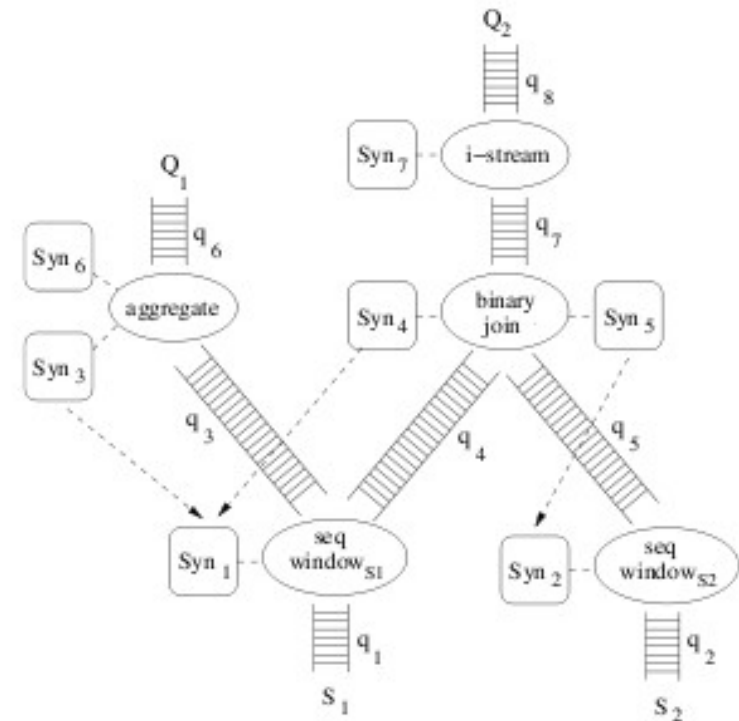
Binary-join: combines the 40,000 most recent tuples from S1 with any tuples from S2 with timestamp $\geq t-600$. Outputs tuples such that $S1.A=S2.A$.



Binary-join reads data from its two input streams by processing the data with the older timestamp. The tuple is compared to all data in the other stream's synopses, and matches are written out to q7. Syn5 refers back to Syn2, and Syn4 refers to Syn1. Note that Syn4 requires a window of 40,000 while Syn1 contains a larger window (of 50,000).

Query Plan: i-stream

Istream: converts the relation produced by binary-join into a stream by writing the set of $R(t)-R(t-1)$.



I-stream requires synopsis Syn7 because it is possible for a tuple to be deleted and added at the same timestamp. If this were not the case, deletion-tagged tuples could simple be ignored and all insertion-tagged tuples could be passed through to q₈.

Operator List

Name	Operator Type	Description
<code>seq-window</code>	stream-to-relation	Implements time-based, tuple-based, and partitioned windows
<code>select</code>	relation-to-relation	Filters tuples based on predicate(s)
<code>project</code>	relation-to-relation	Duplicate-preserving projection
<code>binary-join</code>	relation-to-relation	Joins two input relations
<code>mjoin</code>	relation-to-relation	Multiway join from [VNB03]
<code>union</code>	relation-to-relation	Bag union
<code>except</code>	relation-to-relation	Bag difference
<code>intersect</code>	relation-to-relation	Bag intersection
<code>antijoin</code>	relation-to-relation	Antijoin of two input relations
<code>aggregate</code>	relation-to-relation	Performs grouping and aggregation
<code>duplicate-eliminate</code>	relation-to-relation	Performs duplicate elimination
<code>i-stream</code>	relation-to-stream	Implements <code>Istream</code> semantics
<code>d-stream</code>	relation-to-stream	Implements <code>Dstream</code> semantics
<code>r-stream</code>	relation-to-stream	Implements <code>Rstream</code> semantics
<code>stream-shepherd</code>	system operator	Handles input streams arriving over the network
<code>stream-sample</code>	system operator	Samples specified fraction of tuples
<code>stream-glue</code>	system operator	Adapter for merging a stream-producing view into a plan
<code>rel-glue</code>	system operator	Adapter for merging a relation-producing view into a plan
<code>shared-rel-op</code>	system operator	Materializes a relation for sharing
<code>output</code>	system operator	Sends results to remote clients

System Operators

stream-shepherd: manages the input streams by converting them to the proper internal format and writing to the input queues. This operator may also be responsible for issuing heartbeats, load shedding, and reordering.

stream-sample: handles system-managed load shedding by sampling at a given rate.