03

Continuous Query Language

Notice

- **Author**
  - José Júlio Alferes (jja@fct.unl.pt)
  - João Moura Pires (jmp@fct.unl.pt)

- **This material can be freely used for personal or academic purposes without any previous authorization from the author, provided that this notice is maintained/kept.**

- **For commercial purposes the use of any part of this material requires the previous authorization from the author(s).**

Bibliography

- **Many examples are extracted and adapted from:**
  - Arvind Arasu, Shivnath Babu, Jennifer Widom: The CQL continuous query language: semantic foundations and query execution. VLDB J. 15(2): 121-142 (2006)

Table of Contents

CQL Basic Concepts

Continuous Query Language

- **CQL is a declarative query language for DSMS**
  - It was developed in 2006 for STREAM, the **St**anford St**re**am Dat**a** **M**anagement System
  - It is an extension of SQL, tailored to deal with streams, besides dealing with relations
  - It is part of Oracle's system since version 11g (in Oracle CEP)
    - You may try CQL in Oracle,
    - or by installing the (deprecated) STREAM prototype

Continuous Query Language

- **CQL is a declarative query language for DSMS**
  - It was developed in 2006 for STREAM, the **St**anford St**re**am Dat**a** **M**anagement System
  - It is an extension of SQL, tailored to deal with streams, besides dealing with relations
  - It is part of Oracle's system since version 11g (in Oracle CEP)
    - You may try CQL in Oracle,
    - or by installing the (deprecated) STREAM prototype
- **We will cover**
  - CQL motivation and design principles
  - CQL formal abstract semantics
  - CQL language, and usage examples
  - CQL query plans and execution (and other implementation issues)

Continuous Query Language

- **CQL is a declarative query language for DSMS**
  - It was developed in 2006 for STREAM, the **St**anford St**re**am Dat**a** **M**anagement System
  - It is an extension of SQL, tailored to deal with streams, besides dealing with relations
  - It is part of Oracle's system since version 11g (in Oracle CEP)
    - You may try CQL in Oracle,
    - or by installing the (deprecated) STREAM prototype
- **We will cover**
  - CQL motivation and design principles
  - CQL formal abstract semantics
  - CQL language, and usage examples
  - CQL query plans and execution (and other implementation issues)

Context

Continuous Query Language

- **CQL is a declarative query language for DSMS**
  - It was developed in 2006 for STREAM, the **St**anford St**re**am Dat**a** **M**anagement System
  - It is an extension of SQL, tailored to deal with streams, besides dealing with relations
  - It is part of Oracle's system since version 11g (in Oracle CEP)
    - You may try CQL in Oracle,
    - or by installing the (deprecated) STREAM prototype
- **We will cover**
  - CQL motivation and design principles — Context
  - CQL formal abstract semantics — Relational algebra
  - CQL language, and usage examples
  - CQL query plans and execution (and other implementation issues)

Continuous Query Language

- **CQL is a declarative query language for DSMS**
  - It was developed in 2006 for STREAM, the **St**anford St**re**am Dat**a** **M**anagement System
  - It is an extension of SQL, tailored to deal with streams, besides dealing with relations
  - It is part of Oracle's system since version 11g (in Oracle CEP)
    - You may try CQL in Oracle,
    - or by installing the (deprecated) STREAM prototype
- **We will cover**
  - CQL motivation and design principles — Context
  - CQL formal abstract semantics — Relational algebra
  - CQL language, and usage examples — SQL and its usage
  - CQL query plans and execution (and other implementation issues)

CQL Basic Concepts

CQL Basic Concepts

- **Two main data types:**
  - **Relations**
  - **Streams**

CQL Basic Concepts

- **Two main data types:**
  - **Relations**
  - **Streams**
- **Queries can return:**
  - **a relation** based on other relations (as in SQL)
  - **a stream** based on other streams and relations (extends SQL)

CQL Basic Concepts

- **Two main data types:**
  - **Relations**
  - **Streams**
- **Queries can return:**
  - **a relation** based on other relations (as in SQL)
  - **a stream** based on other streams and relations (extends SQL)
- **This is done with an algebra that has 3 kinds of operators**
  - ***stream-to-relation***, ***relation-to-relation***, ***relation-to-stream***

CQL Basic Concepts

- **Two main data types:**
  - **Relations**
  - **Streams**
- **Queries can return:**
  - **a relation** based on other relations (as in SQL)
  - **a stream** based on other streams and relations (extends SQL)
- **This is done with an algebra that has 3 kinds of operators**
  - ***stream-to-relation***, ***relation-to-relation***, ***relation-to-stream***
- **There are no *stream-to-stream* operators, and this makes things closer to SQL**
  - Can be obtained by combining *stream-to-relation - relation-to-relation - relation-to-stream*

A motivating example

- **Simplified version of the Linear Road Benchmark**

A motivating example

- **Simplified version of the Linear Road Benchmark**

A motivating example

- **Simplified version of the Linear Road Benchmark**
- **A system that computes variable tolling in a highway**
  - The tolls depend on the distance travelled in the highway (as usual), but also on traffic conditions
    - The more congested the highway is, the more toll is charged (in order to discourage cars to go to congested highways)

A motivating example

- **Simplified version of the Linear Road Benchmark**
- **A system that computes variable tolling in a highway**
  - The tolls depend on the distance travelled in the highway (as usual), but also on traffic conditions
    - The more congested the highway is, the more toll is charged (in order to discourage cars to go to congested highways)
- **Each car has a sensor that relays its position and speed every 30s, while in the highway**
  - This information is continuously passed to the system, as a stream of data

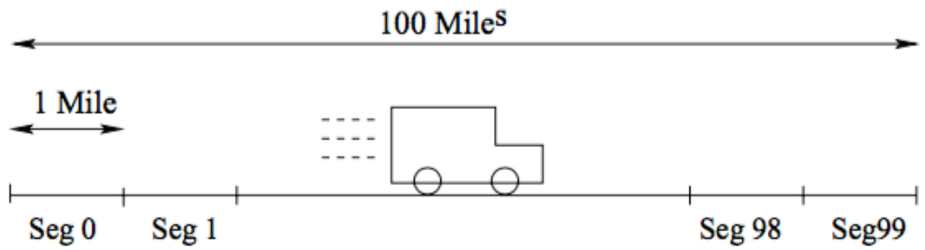
A motivating example

- **Simplified version of the Linear Road Benchmark**
- **A system that computes variable tolling in a highway**
  - The tolls depend on the distance travelled in the highway (as usual), but also on traffic conditions
    - The more congested the highway is, the more toll is charged (in order to discourage cars to go to congested highways)
- **Each car has a sensor that relays its position and speed every 30s, while in the highway**
  - This information is continuously passed to the system, as a stream of data
- **The system computes tolls in real-time, and continuously outputs a stream with the prices to pay by each car, upon their exit**

Simplified Linear Road (details)

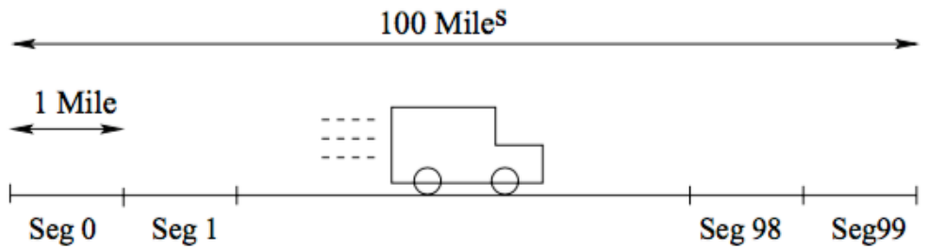
- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.

Simplified Linear Road (details)



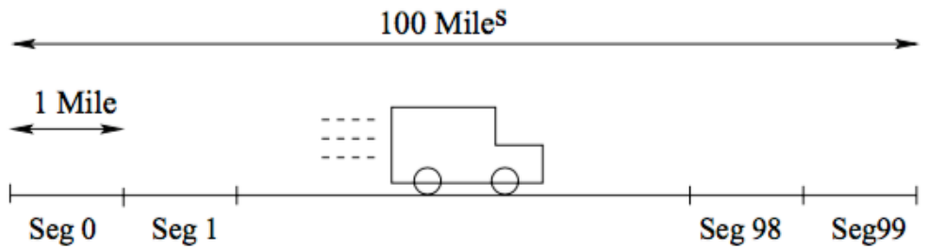
- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.

Simplified Linear Road (details)



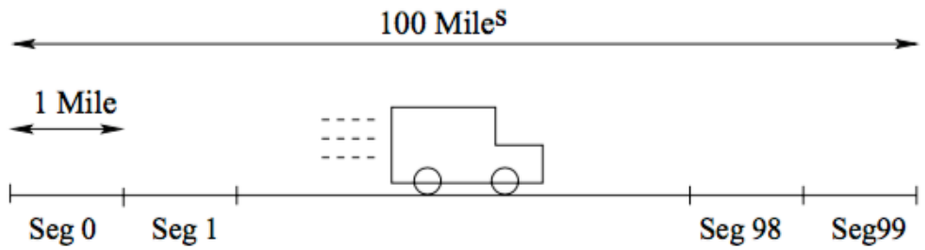
- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.
- **The input stream has tuples with carID, position, and speed**
  - The position is given as the distance in number of feet from the beginning of the road; the speed in miles per hour

Simplified Linear Road (details)



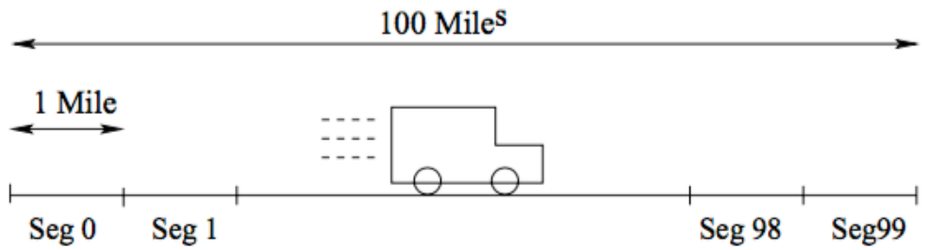
- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.
- **The input stream has tuples with carID, position, and speed**
  - The position is given as the distance in number of feet from the beginning of the road; the speed in miles per hour
- **The output stream has tuples with carID and price to pay for a segment,** any time a car enters that segment

Simplified Linear Road (details)



- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.
- **The input stream has tuples with carID, position, and speed**
  - The position is given as the distance in number of feet from the beginning of the road; the speed in miles per hour
- **The output stream has tuples with carID and price to pay for a segment,** any time a car enters that segment
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment. A segment is **congested** if the **average speed of all cars in the last 5 minutes** < than 40 Miles/h.

Simplified Linear Road (details)



- The road is **divided into 100 segments** (of 1 Mile each), possibly with an entrance at the beginning of each segment, and/or an exit at the end.
- **The input stream has tuples with carID, position, and speed**
  - The position is given as the distance in number of feet from the beginning of the road; the speed in miles per hour
- **The output stream has tuples with carID and price to pay for a segment,** any time a car enters that segment
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment. A segment is **congested** if the **average speed of all cars in the last 5 minutes** < than 40 Miles/h.
- Note that all of this changes dynamically, and the **continuous query must compute the tolls at all times**

Streams and Relations

- There is a notion of **schemas**, which is exactly as in relational DBs (**a set of named attributes**)
- There is also a notion of **timestamps** (or time instants) which is simply a **discrete totally ordered set** T (e.g. nonnegative integers, or datetime)

Streams and Relations

- There is a notion of **schemas**, which is exactly as in relational DBs (**a set of named attributes**)
- There is also a notion of **timestamps** (or time instants) which is simply a **discrete totally ordered set** T (e.g. nonnegative integers, or datetime)
- A **stream *S*** is a possible infinite **bag of elements <s, *t*>** where ***s*** is a tuple in the schema of **S**, and $t \in$ T is the timestamp of the element.

Streams and Relations

- There is a notion of **schemas**, which is exactly as in relational DBs (**a set of named attributes**)
- There is also a notion of **timestamps** (or time instants) which is simply a **discrete totally ordered set** T (e.g. nonnegative integers, or datetime)
- A **stream *S*** is a possible infinite **bag of elements <s, *t*>** where ***s*** is a tuple in the schema of **S**, and $t \in T$ is the timestamp of the element.
  - I.e. **stream are timestamped sequences of tuples**

Streams and Relations

- There is a notion of **schemas**, which is exactly as in relational DBs (**a set of named attributes**)
- There is also a notion of **timestamps** (or time instants) which is simply a **discrete totally ordered set** T (e.g. nonnegative integers, or datetime)
- A **stream *S*** is a possible infinite **bag of elements <s, *t*>** where ***s*** is a tuple in the schema of **S**, and $t \in T$ is the timestamp of the element.
  - I.e. **stream are timestamped sequences of tuples**
- **A relation *R* is a mapping from each time instant in T to a finite bag of tuples in the schema of *R***

Instantaneous Relations and Relations

- **Instantaneous relation *R(t)* is the bag of tuples of *R* at time *t***
  - This exactly corresponds to a relation in DBs

Instantaneous Relations and Relations

- **Instantaneous relation *R(t)* is the bag of tuples of *R* at time *t***
  - This exactly corresponds to a relation in DBs

- **Note that these relations are an extension of relations in DBs**
  - In DBs a relation is a (permanent) bag of tuples - it does not take into account the changes in relations
  - DSMS must take into account that relations change over time

Instantaneous Relations and Relations

- **Instantaneous relation *R(t)* is the bag of tuples of *R* at time *t***
  - This exactly corresponds to a relation in DBs

- **Note that these relations are an extension of relations in DBs**
  - In DBs a relation is a (permanent) bag of tuples - it does not take into account the changes in relations
  - DSMS must take into account that relations change over time

- **A Relation is a set of Instantaneous relation *R(t)***

Instantaneous Relations and Relations

- ***S* up to *t*** denotes the bag of elements in stream S with timestamps ≤ *t*
  - ***S*** up to $t = \{\langle s,t'\rangle \in S : t' \leq t\}$

Instantaneous Relations and Relations

- ***S* up to *t*** denotes the bag of elements in stream S with timestamps ≤ *t*
  - ***S*** up to $t = \{\langle s,t'\rangle \in S : t' \leq t\}$
- **S at *t*** denotes the bag of elements of S with timestamp *t*
  - ***S*** at $t = \{\langle s,t'\rangle \in S : t' = t\}$

Instantaneous Relations and Relations

- ***S* up to *t*** denotes the bag of elements in stream S with timestamps ≤ *t*
  - ***S*** up to $t = \{\langle s,t'\rangle \in S : t' \leq t\}$
- **S at *t*** denotes the bag of elements of S with timestamp *t*
  - ***S*** at $t = \{\langle s,t'\rangle \in S : t' = t\}$

- ***R* up to *t*** denotes the collection of instantaneous relations $R(0), \ldots, R(t)$

Instantaneous Relations and Relations

- ***S* up to *t*** denotes the bag of elements in stream S with timestamps ≤ *t*
  - ***S*** up to $t = \{\langle s,t'\rangle \in S : t' \leq t\}$
- **S at *t*** denotes the bag of elements of S with timestamp *t*
  - ***S*** at $t = \{\langle s,t'\rangle \in S : t' = t\}$

- ***R* up to *t*** denotes the collection of instantaneous relations $R(0), \ldots, R(t)$
- **R at *t*** denotes the instantaneous relation $R(t)$.

Linear Road Example

- The Linear Road application there is just **one base stream** containing vehicle speed and position measurements, with schema:

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

Linear Road Example

- The Linear Road application there is just **one base stream** containing vehicle speed and position measurements, with schema:

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

- 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

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

Linear Road Example

- The Linear Road application there is just **one base stream** containing vehicle speed and position measurements, with schema:

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

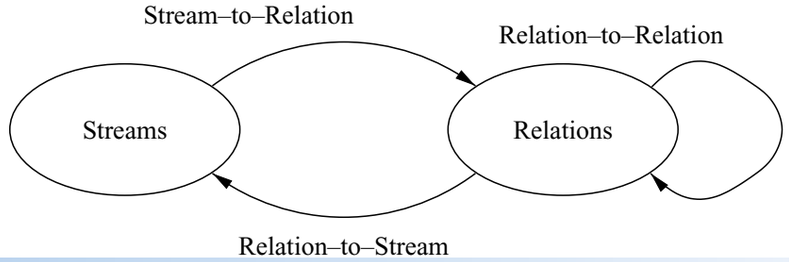
- The Linear Road application contains **no base relations**, but several **derived relations** are useful in toll computation. For example, the toll for a congested segment depends on the current number of vehicles in the segment

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

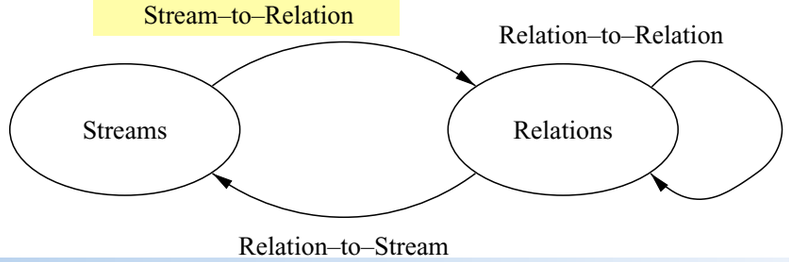
- At time *t*, `SegVolRel(t)` contains the count of vehicles (`numVehicles`) in each highway segment (`segNo`) as of time t.

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.

Abstract operators

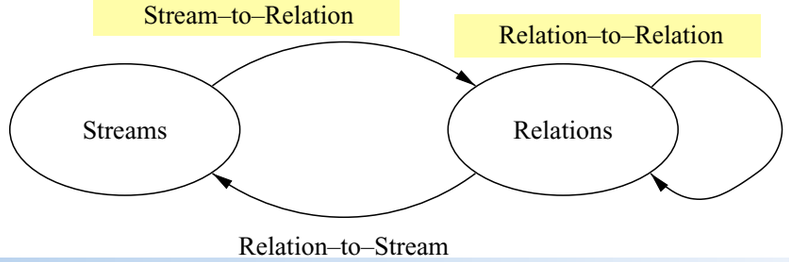


Abstract operators



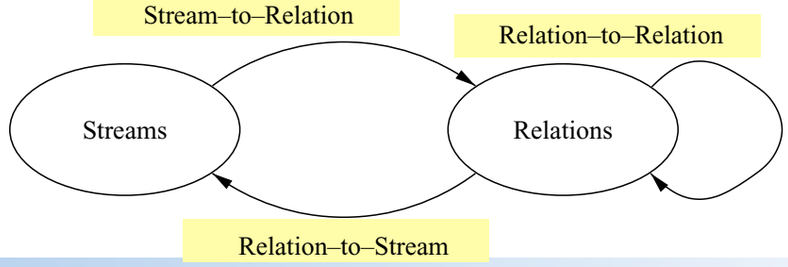
- **Stream-to-relation (s2r) operators**
  - Receive a stream *S* and produce a relation *R* with the same schema of *S*
  - At any instant *t*, *R(t)* must be computable from *S up to t*
    - (i.e. from the tuples of *S* with timestamp ≤ *t*)

Abstract operators



- **Stream-to-relation (s2r) operators**
  - Receive a stream *S* and produce a relation *R* with the same schema of *S*
  - At any instant *t*, *R(t)* must be computable from *S up to t*
    - (i.e. from the tuples of *S* with timestamp ≤ *t*)
- **Relation-to-relation (r2r) operators**
  - Receive one or more relations $R_1 \ldots R_n$ and produce a relation *R*
  - At any instant *t*, $R(t)$ must be computable from $R_1(t) \ldots R_n(t)$

Abstract operators



- **Stream-to-relation (s2r) operators**
  - Receive a stream *S* and produce a relation *R* with the same schema of *S*
  - At any instant *t*, *R(t)* must be computable from *S up to t*
    - (i.e. from the tuples of *S* with timestamp ≤ *t*)
- **Relation-to-relation (r2r) operators**
  - Receive one or more relations $R_1 \ldots R_n$ and produce a relation *R*
  - At any instant *t*, *R(t)* must be computable from $R_1(t) \ldots R_n(t)$
- **Relation-to-stream (r2s) operators**
  - Receive one relation *R* and produce a stream *S*
  - At any instant *t*, *S at t* must be computable from *R(t)*
    - (where *S at t* is the bag of all tuples of S with timestamp *t*)

Continuous Semantics

- A **query *Q*** is a well typed algebraic expression (or **tree**) formed with the previously mentioned operators.

Continuous Semantics

- A **query *Q*** is a well typed algebraic expression (or **tree**) formed with the previously mentioned operators.
- Suppose the set of **all inputs** to the innermost (leaf) operators of Q are **streams** S_1, . . . , S_n ($n \geq 0$) and **relations** R_1,..., R_m ($m \geq 0$)

Continuous Semantics

- A **query *Q*** is a well typed algebraic expression (or **tree**) formed with the previously mentioned operators.
- Suppose the set of **all inputs** to the innermost (leaf) operators of Q are **streams** S_1, . . . , S_n ($n \geq 0$) and **relations** R_1,..., R_m ($m \geq 0$)

- The **result of continuous query Q at a time t** , which denotes the result of Q once all inputs up to *t* are “available”. There are two cases:
  - If the outermost operator of *Q* is a r2s operator producing *S*
  - If the outermost operator is a s2r or r2r operator producing R

Continuous Semantics

- The **result of continuous query Q at a time t** , which denotes the result of Q once all inputs up to *t* are “available”. There are two cases:
- **If the outermost operator of *Q* is a r2s operator producing *S***
  - the result of *Q* at *t* is ***S up to t***
  - it is produced by recursively applying the operators in *Q* to the leaf streams and relations *up to t*

Continuous Semantics

- The **result of continuous query Q at a time t** , which denotes the result of Q once all inputs up to *t* are “available”. There are two cases:
- **If the outermost operator of *Q* is a r2s operator producing *S***
  - the result of *Q* at *t* is ***S up to t***
  - it is produced by recursively applying the operators in *Q* to the leaf streams and relations *up to t*
- **If the outermost operator is a s2r or r2r operator producing *R***
  - the result of *Q* at *t* is ***R(t)***,
  - it is produced by recursively applying the operators in *Q* to the leaf streams and relations *up to t*

Continuous Queries

- **How to operationally understand the continuous semantics?**

Continuous Queries

- **How to operationally understand the continuous semantics?**
  - Let time advance in T

Continuous Queries

- **How to operationally understand the continuous semantics?**
  - Let time advance in T
  - At time $t \in$ T, **all inputs up to *t* are processed**
    - This requires that, after processing the outputs at time t, no further input with timestamp *t* can arrive
    - *heartbeats* are timestamps *t* that prevent the system from receiving further input with timestamps ≤ *t*

Continuous Queries

- **How to operationally understand the continuous semantics?**
  - Let time advance in T
  - At time $t \in T$, **all inputs up to *t* are processed**
    - This requires that, after processing the outputs at time t, no further input with timestamp *t* can arrive
    - *heartbeats* are timestamps *t* that prevent the system from receiving further input with timestamps ≤ *t*
  - If the query produces a **stream**, the continuous query **emits new results with timestamp *t***

Continuous Queries

- **How to operationally understand the continuous semantics?**
  - Let time advance in T
  - At time $t \in$ T, **all inputs up to *t* are processed**
    - This requires that, after processing the outputs at time t, no further input with timestamp *t* can arrive
    - *heartbeats* are timestamps *t* that prevent the system from receiving further input with timestamps ≤ *t*
  - If the query produces a **stream**, the continuous query **emits new results with timestamp *t***
  - If the query produces a **relation** *R*, it **updates the relation from *R(t-1)* to *R(t)***

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*

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

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

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

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

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

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

Time	S	LastRow
0	$\langle(a_0),0\rangle$	(a_0)
1	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$	(a_1)
2	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$	(a_2)
3	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$	(a_3)
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)
⋮	⋮	⋮

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*
- `Filter` is r2r and filters way all a_i where *i* is odd

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

Time	S	LastRow
0	$\langle(a_0),0\rangle$	(a_0)
1	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$	(a_1)
2	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$	(a_2)
3	$\langle(a_0),0\rangle$ $\langle(a_1),1\rangle$ $\langle(a_2),2\rangle$ $\langle(a_3),3\rangle$	(a_3)
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)
⋮	⋮	⋮

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*
- `Filter` is r2r and filters way all a_i where *i* is odd

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

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

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*
- `Filter` is r2r and filters way all a_i where *i* is odd
- `Istream` is r2s that “streams” every **new** tuple in *R*

CQL

```
Select Istream(*)
From S [Rows 1]
Where <filter>.
```

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

Continuous Semantics Example

- Query: `Istream(Filter(LastRow(S))`
- *S* has a_i every instant *i*
- `LastRow` is s2r that at time *t* produces a relation with the last tuples that arrived in *S*
- `Filter` is r2r and filters way all a_i where *i* is odd
- `Istream` is r2s that “streams” every **new** tuple in *R*

CQL

```
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)	$\langle(a_0),0\rangle$ $\langle(a_2),2\rangle$ $\langle(a_4),4\rangle$
⋮	⋮	⋮	⋮	⋮

`SegVolRel (segNo, numVehicles)`

Requirements for processing a query

- In the Linear Road application the sequence of operators producing derived relation **`SegVolRel`** conceptually produces, **at every time instant *t*,** the instantaneous relation **`SegVolRel(t)`** containing the current number of vehicles in each segment.

`SegVolRel (segNo, numVehicles)`

Requirements for processing a query

- In the Linear Road application the sequence of operators producing derived relation **`SegVolRel`** conceptually produces, **at every time instant *t*,** the instantaneous relation **`SegVolRel(t)`** containing the current number of vehicles in each segment.
- In a DSMS implementing this semantics, `SegVolRel(t)` **cannot be produced until it is known that all elements on input stream**

`PosSpeedStr(vehicleId,speed,xPos)`

with **timestamp ≤ t have been received** !

`SegVolRel (segNo, numVehicles)`

Requirements for processing a query

- In the Linear Road application the sequence of operators producing derived relation **`SegVolRel`** conceptually produces, **at every time instant *t*,** the instantaneous relation **`SegVolRel(t)`** containing the current number of vehicles in each segment.
- In a DSMS implementing this semantics, `SegVolRel(t)` **cannot be produced until it is known that all elements on input stream**

`PosSpeedStr(vehicleId,speed,xPos)`

with **timestamp ≤ t have been received** !

- Furthermore, once they have been received, **there may be additional lag before the relation is actually updated** due to query processing time.

CQL language

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**

- **General Idea**

  - Support a **large class** of relation-to-relation operators that **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**.

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**

- **General Idea**

  - Support a **large class** of relation-to-relation operators that **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 !

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**
- **r2r operators are those of relational algebra**
  - Formally, they have to be adapted to time-varying relations, but that is easy:
  - If $\oplus_r$ is a traditional operator (e.g. $\bowtie$, Π, σ, …) over $R_1, \ldots, R_n$ then $\oplus$ in CQL produces a time-varying relations R, such that at each t, $R(t) = \oplus(R_1(t), \ldots, R_n(t))$

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**
- **r2r operators are those of relational algebra**
  - Formally, they have to be adapted to time-varying relations, but that is easy:
  - If $\oplus_r$ is a traditional operator (e.g. $\bowtie$, Π, σ, …) over $R_1, \ldots, R_n$ then $\oplus$ in CQL produces a time-varying relations R, such that at each t, $R(t) = \oplus(R_1(t), \ldots, R_n(t))$
- **For writing the queries, CQL uses an extension of SQL syntax**
  - By now, it should be more than clear to you how to switch from SQL syntax to relational algebra, and back!

CQL language

- **CQL is obtained by a precise instantiation of the abstract operators**
- **r2r operators are those of relational algebra**
  - Formally, they have to be adapted to time-varying relations, but that is easy:
  - If $\oplus_r$ is a traditional operator (e.g. $\bowtie$, Π, σ, …) over $R_1, \ldots, R_n$ then $\oplus$ in CQL produces a time-varying relations R, such that at each t, $R(t) = \oplus(R_1(t), \ldots, R_n(t))$
- **For writing the queries, CQL uses an extension of SQL syntax**
  - By now, it should be more than clear to you how to switch from SQL syntax to relational algebra, and back!
- **Incorporating user-defined** procedures, aggregates, and windows, is **straightforward in CQL**, at least from the semantics perspective.

CQL's **s2r** operators

CQL's **s2r** operators

- **They are all based on the concept of sliding window over a stream**
  - at any point *t* **a window is an instantaneous relation *R(t)* that contains a historical snapshot** of a finite portion of the stream
    - Note that in the relation the timestamps disappear

CQL's **s2r** operators

- **They are all based on the concept of sliding window over a stream**
  - at any point *t* **a window is an instantaneous relation *R(t)* that contains a historical snapshot** of a finite portion of the stream
    - Note that in the relation the timestamps disappear
- **4 types of sliding window s2r operators**
  - **time-based** (e.g. last minute)
  - **tuple-based** (e.g. last five tuples)
  - **partitioned** (similar to `group by` in SQL)
  - **with slide parameter** (e.g. last minute, picked every minute)

Time-based windows

- **`S [Range T]` takes a stream *S* and a time-interval *T*, and produces the relation *R* such that**

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

Time-based windows

- **`S [Range T]` takes a stream *S* and a time-interval *T*, and produces the relation *R* such that**

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

  - Produces a relation with all tuples of stream *S* with timestamps in the last *T* time
    - E.g. `S [Range 30s]` produces a relation with all tuples from S in the last 30 seconds

- **Note that the tuples in *R* do not have timestamps. They are "time-flat"!**

Time-based windows

- **`S [Range T]` takes a stream *S* and a time-interval *T*, and produces the relation *R* such that**

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

  - Produces a relation with all tuples of stream *S* with timestamps in the last *T* time
    - E.g. `S [Range 30s]` produces a relation with all tuples from S in the last 30 seconds

- **Note that the tuples in *R* do not have timestamps. They are "time-flat"!**

- **Special cases:**

  - *T=0* (write `S [Now]`)
    - at any time *t*, *R(t)* has the tuples that appeared in the stream at *t*, possibly empty
  - *T=∞* (write `S [Range Unbounded]`)
    - at any time *R*, has all the tuples that ever appeared in *S*, so far

Example

- Let **`PosSpeedStr`** be the **input stream** of the Linear Road

Example

- Let **`PosSpeedStr`** be the **input stream** of the Linear Road
- $\sigma_{speed \geq 60}$(`PosSpeedStr [Range 30s]`) is the relation that, at any instant *t,* has all the tuples `(carID,position,speed)` of cars detected in the 30 seconds just before *t*, with speed higher than 60

Example

- Let **`PosSpeedStr`** be the **input stream** of the Linear Road
- $\sigma_{speed \geq 60}$(`PosSpeedStr [Range 30s]`) is the relation that, at any instant *t,* has all the tuples `(carID,position,speed)` of cars detected in the 30 seconds just before *t*, with speed higher than 60
- **In SQL-like syntax:**

```
select *
from PosSpeedStr [Range 30s]
where speed ≥ 60;
```

Example

- Let **`PosSpeedStr`** be the **input stream** of the Linear Road
- $\sigma_{speed \geq 60}$(`PosSpeedStr [Range 30s]`) is the relation that, at any instant *t,* has all the tuples `(carID,position,speed)` of cars detected in the 30 seconds just before *t*, with speed higher than 60
- **In SQL-like syntax:**

```
select *
from PosSpeedStr [Range 30s]
where speed ≥ 60;
```

- **The ids of cars that were ever detected with a speed higher than 200**

```
select distinct carID
from PosSpeedStr [Range Unbounded]
where speed ≥ 200;
```

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***
    - if S has less than N tuples with timestamps ≤ t, then R(t) has all tuples in *S*

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***
    - if S has less than N tuples with timestamps ≤ t, then R(t) has all tuples in *S*
    - if there are ties, then the *N* tuples are determined arbitrarily

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***
    - if S has less than N tuples with timestamps ≤ t, then R(t) has all tuples in *S*
    - if there are ties, then the *N* tuples are determined arbitrarily
  - Produces a relation with the last *N* tuples of stream *S*

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***
    - if S has less than N tuples with timestamps ≤ t, then R(t) has all tuples in *S*
    - if there are ties, then the *N* tuples are determined arbitrarily
  - Produces a relation with the last *N* tuples of stream *S*
- **E.g. `PosSpeedStr [Rows 1]` denotes, at any instant, the latest position-speed measurement**

Tuple-based windows

- **`S [Rows N]` takes a stream *S* and an integer *N*, and produces the relation *R* such that**
  - *R(t)* contains exactly the *N* tuples in stream *S* with **highest timestamps ≤ *t***
    - if S has less than N tuples with timestamps ≤ t, then R(t) has all tuples in *S*
    - if there are ties, then the *N* tuples are determined arbitrarily
  - Produces a relation with the last *N* tuples of stream *S*
- **E.g. `PosSpeedStr [Rows 1]` denotes, at any instant, the latest position-speed measurement**
- **Special case:**
  - *N*=∞ (write `S [Rows Unbounded]`)
    - It is equivalent to `S [Range Unbounded]`

Partitioned windows

- **What if we wanted to have *the last measurement for each car*?**
  - similar to *group by* each carID and then getting the last measurement for each group

Partitioned windows

- **What if we wanted to have *the last measurement for each car*?**
  - similar to *group by* each carID and then getting the last measurement for each group
- **`S [Partition By A1, …, An Rows N]` takes a stream *S*, a set of attributes $A_1, \ldots, A_n$, and an integer *N*, and produces the relation *R* where:**
  - a tuple s with values $a_1 \ldots a_n$ in attributes $A_1, \ldots, A_n$ belongs to *R(t)* iff there exists $\langle s,t' \rangle \in$ *S*, with $t' \leq t$, such that t' is among the *N* largest timestamps of elements in S with those values in those attributes

Partitioned windows

- **What if we wanted to have *the last measurement for each car*?**
  - similar to *group by* each carID and then getting the last measurement for each group
- **`S [Partition By A1, …, An Rows N]` takes a stream *S*, a set of attributes $A_1, \ldots, A_n$, and an integer *N*, and produces the relation *R* where:**
  - a tuple s with values $a_1 \ldots a_n$ in attributes $A_1, \ldots, A_n$ belongs to $R(t)$ iff there exists $\langle s,t' \rangle \in S$, with $t' \leq t$, such that t' is among the N largest timestamps of elements in S with those values in those attributes
- **E.g. `PosSpeedStr [Partition by carID Rows 1]`**

Partitioned windows

- **What if we wanted to have *the last measurement for each car*?**
  - similar to *group by* each carID and then getting the last measurement for each group
- **`S [Partition By A1, …, An Rows N]` takes a stream *S*, a set of attributes $A_1, \ldots, A_n$, and an integer *N*, and produces the relation *R* where:**
  - a tuple s with values $a_1 \ldots a_n$ in attributes $A_1, \ldots, A_n$ belongs to $R(t)$ iff there exists $\langle s,t' \rangle \in$ *S*, with $t' \leq t$, such that t' is among the *N* largest timestamps of elements in S with those values in those attributes
- **E.g. `PosSpeedStr [Partition by carID Rows 1]`**
  - denotes, at any instant, the latest position-speed measurement of each car

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
  - A sliding windows that only changes once every minute, and, in a change, takes everything from the last minute

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
  - A sliding windows that only changes once every minute, and, in a change, takes everything from the last minute
- **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
  - The slide parameter is a **positive integer** for row-based and partitioned windows

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
  - A sliding windows that only changes once every minute, and, in a change, takes everything from the last minute
- **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
  - The slide parameter is 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]

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

where t_{start} is the largest multiple of L smaller than t

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

where t_{start} is the largest multiple of L smaller than t

- So it gets the elements with a time-stamp between t_{start} and t_{end} = max(t_{start} - T, 0)

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

where t_{start} is the largest multiple of L smaller than t

- So it gets the elements with a time-stamp between t_{start} and t_{end} = max(t_{start} - T, 0)
- **E.g. `PosSpeedStr [Range 30s Slide 1min]`**
  - every minute, get the measurements from the last clock half minute
    - up to second 59, it outputs nothing
    - from time 60s to 119s, contains the measurement made in the 2nd half minute (from second 31 to 60)
    - from time 120s to 179s, contains the measurements made in the 4th half minute

Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

where t_{start} is the largest multiple of L smaller than t

- So it gets the elements with a time-stamp between t_{start} and t_{end} = max(t_{start} - T, 0)
- **E.g. `PosSpeedStr [Range 30s Slide 1min]`**
  - every minute, get the measurements from the last clock half minute
    - up to second 59, it outputs nothing
    - from time 60s to 119s, contains the measurement made in the 2nd half minute (from second 31 to 60)
    - from time 120s to 179s, contains the measurements made in the 4th half minute

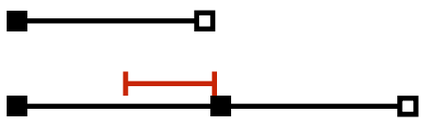
Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

where t_{start} is the largest multiple of L smaller than t

- So it gets the elements with a time-stamp between t_{start} and t_{end} = max(t_{start} - T, 0)
- **E.g. `PosSpeedStr [Range 30s Slide 1min]`**
  - every minute, get the measurements from the last clock half minute
    - up to second 59, it outputs nothing
    - from time 60s to 119s, contains the measurement made in the 2nd half minute (from second 31 to 60)
    - from time 120s to 179s, contains the measurements made in the 4th half minute



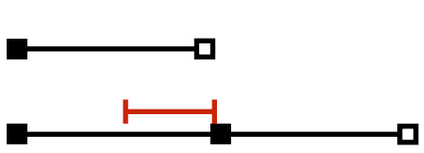
Windows with a slide parameter

- **What if we only want to look at the measures once every minute?**
- **`S [Range T Slide L]` produces relation *R*, where:**

$$R(t) = \begin{cases} \{\} \text{ if } t < L - 1 \\ \{s | \langle s, t' \rangle \in S \wedge max(t_{start} - T, 0) \leq t' \leq t_{start}\} \text{ otherwise} \end{cases}$$

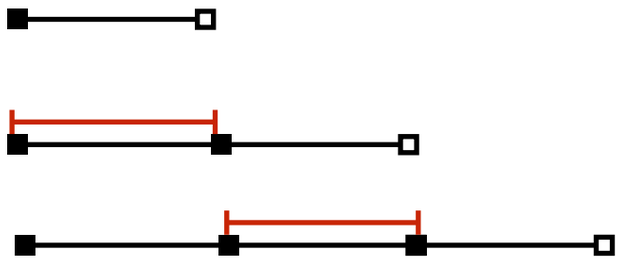
where t_{start} is the largest multiple of L smaller than t

- So it gets the elements with a time-stamp between t_{start} and t_{end} = max(t_{start} - T, 0)
- **E.g. `PosSpeedStr [Range 30s Slide 1min]`**
  - every minute, get the measurements from the last clock half minute
    - up to second 59, it outputs nothing
    - from time 60s to 119s, contains the measurement made in the 2nd half minute (from second 31 to 60)
    - from time 120s to 179s, contains the measurements made in the 4th half minute



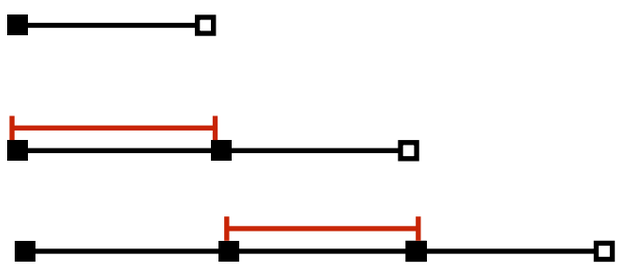
Windows with a slide parameter

- **T**his type of window in which **L = T** has been referred to as a **tumbling** window



Windows with a slide parameter

- This type of window in which **L = T** has been referred to as a **tumbling** window



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

CQL's **r2s** operators

- **They take a varying-time relation *R*, and build a stream *S* based on the changes that occur in *R* over the time**
- **Three r2s operators**

CQL's **r2s** operators

- **They take a varying-time relation *R*, and build a stream *S* based on the changes that occur in *R* over the time**
- **Three r2s operators**
  - **Istream** (*insert stream*): the stream "registers" the **insertions** made in the relation

CQL's **r2s** operators

- **They take a varying-time relation *R*, and build a stream *S* based on the changes that occur in *R* over the time**
- **Three r2s operators**
  - **Istream** (*insert stream*): the stream "registers" the **insertions** made in the relation
  - **Dstream** (*delete stream*): "registers" the **deletions**

CQL's **r2s** operators

- **They take a varying-time relation *R*, and build a stream *S* based on the changes that occur in *R* over the time**
- **Three r2s operators**
  - **Istream** (*insert stream*): the stream "registers" the **insertions** made in the relation
  - **Dstream** (*delete stream*): "registers" the **deletions**
  - **Rstream** (*relation stream*): at each instant, outputs to the stream **all the tuples** that are in the relation

Insert stream

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

- I.e. S has <*s,t*> whenever *s* is in *R(t)* and was not in *R(t-1)*

Insert stream

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

- I.e. S has <*s,t*> whenever *s* is in *R(t)* and was not in *R(t-1)*

- **Example: Output in a stream the id and position of a car whenever that car is detected with a speed greater than 150**

```
select Istream(carID, position)
from PosSpeedStr [Range Unbounded]
where speed ≥ 150;
```

Insert stream

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

- I.e. S has <*s,t*> whenever *s* is in *R(t)* and was not in *R(t-1)*

- **Example: Output in a stream the id and position of a car whenever that car is detected with a speed greater than 150**

```
select Istream(carID, position)
from PosSpeedStr [Range Unbounded]
where speed ≥ 150;
```

- Note that, this query has as **input a stream** and as **output another stream**, and continuously produces the output upon what it receives from the input - i.e., it is a continuous query

Delete stream

$$\texttt{Dstream}(R) = \bigcup_{t \geq 0} ((R(t-1) - R(t)) \times \{t\})$$

- I.e. S has <*s,t*> whenever *s* was in *R(t-1)* and is no longer in *R(t)*

Delete stream

$$\texttt{Dstream}(R) = \bigcup_{t \geq 0} ((R(t-1) - R(t)) \times \{t\})$$

- I.e. S has <*s,t*> whenever *s* was in *R(t-1)* and is no longer in *R(t)*

- **Example: Output in a stream the ids of cars that exit the highway**

```
select Dstream(carID)
from PosSpeedStr [Range 30s];
```

- An id of a car is output in the stream whenever the car reported its position and speed before 30 seconds ago, and in the last 30 seconds did not send anything

Relation stream

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

- I.e. S has <*s,t*> whenever *s* is *R(t)* - at any instant, outputs to the stream the whole content of the relation

Relation stream

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

- I.e. S has <*s,t*> whenever *s* is *R(t)* - at any instant, outputs to the stream the whole content of the relation

- **Example: What does this query compute?**

```
select Rstream(carID, position)
from PosSpeedStr [Now]
where speed ≥ 150;
```

Relation stream

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

- I.e. S has <*s,t*> whenever *s* is *R(t)* - at any instant, outputs to the stream the whole content of the relation

- **Example: What does this query compute?**

```
select Rstream(carID, position)
from PosSpeedStr [Now]
where speed ≥ 150;
```

- **Exactly the same as this other one!**

```
select Istream(carID, position)
from PosSpeedStr [Range Unbounded]
where speed ≥ 150;
```

- **We will see that this is a general equivalence result. This is the general form of a stream filter**

Shortcuts

- **To makes things easier to write, there are some shortcut and default assumptions (though this is not usual in SQL-like languages)**

  1. When a stream is referred in a place where a relation is expected (e.g. in the from clause) assume an unbounded range
  2. If one has a monotonic query based on streams, assume that the result should be a stream, and use the Istream operator

Shortcuts

- **To makes things easier to write, there are some shortcut and default assumptions (though this is not usual in SQL-like languages)**

  1. When a stream is referred in a place where a relation is expected (e.g. in the from clause) assume an unbounded range
  2. If one has a monotonic query based on streams, assume that the result should be a stream, and use the Istream operator

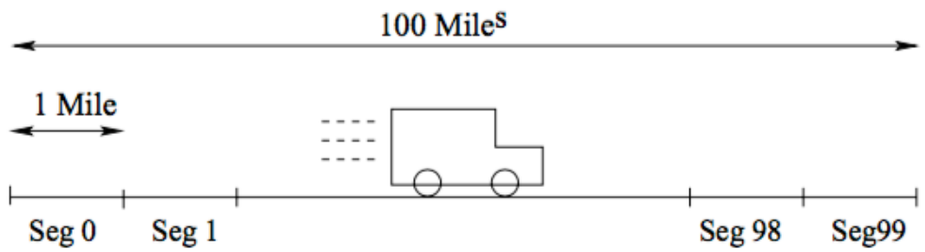
- **Example: With this, one can simply write**

```
select carID, position
from PosSpeedStr
where speed ≥ 150;
```

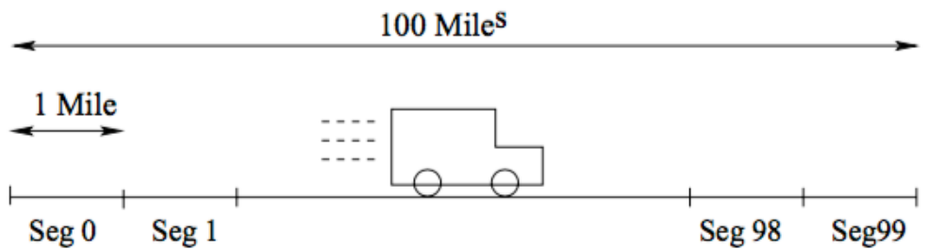
  - to denote

```
select Istream(carID, position)
from PosSpeedStr [Range Unbounded]
where speed ≥ 150;
```

Simplified Linear Road (details)

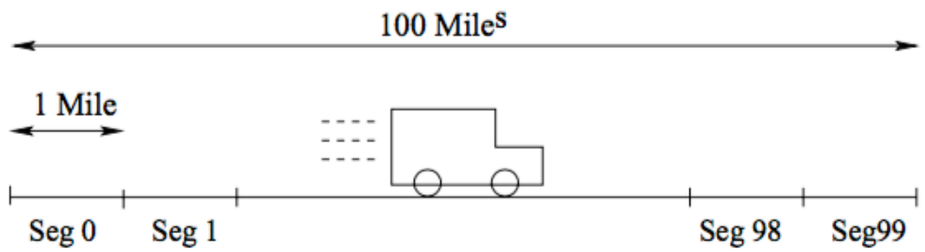


Simplified Linear Road (details)



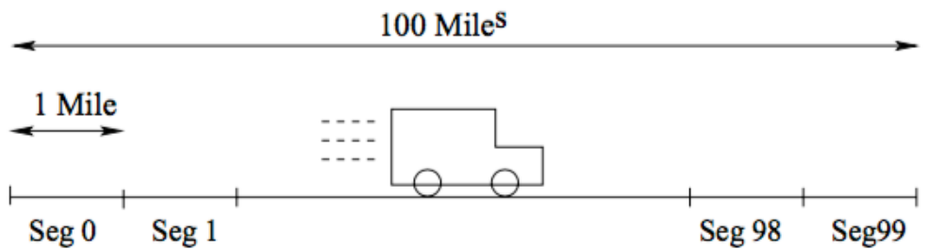
- **The road is divided into 100 segments (of Mile each)**

Simplified Linear Road (details)



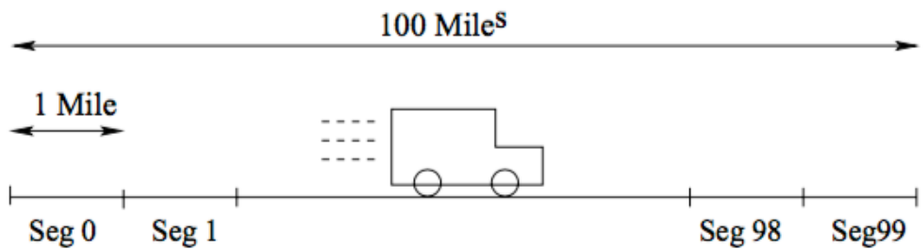
- **The road is divided into 100 segments (of Mile each)**
- **The input stream has tuples with carID, position, and speed, every 30s**
  - The position given as the distance in feet from the beginning of the road

Simplified Linear Road (details)



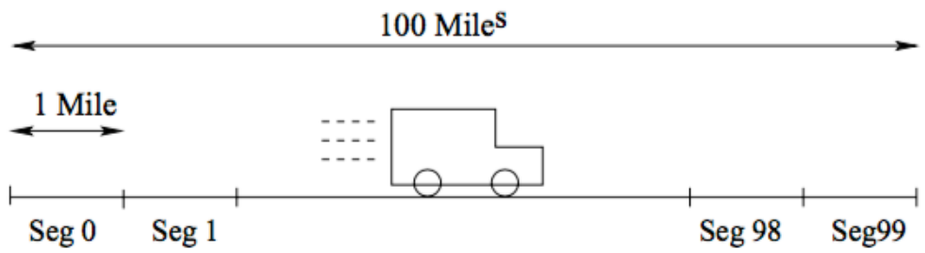
- **The road is divided into 100 segments (of Mile each)**
- **The input stream has tuples with carID, position, and speed, every 30s**
  - The position given as the distance in feet from the beginning of the road
- **The output stream has tuples with carID and price to pay for a segment, any time a car enters that segment**

Simplified Linear Road (details)



- **The road is divided into 100 segments (of Mile each)**
- **The input stream has tuples with carID, position, and speed, every 30s**
  - The position given as the distance in feet from the beginning of the road
- **The output stream has tuples with carID and price to pay for a segment, any time a car enters that segment**
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment
  - a seg. is congested if the average speed of all cars in the last 5 minutes is less than 40 Miles/h

Simplified Linear Road (details)



- **The road is divided into 100 segments (of Mile each)**
- **The input stream has tuples with carID, position, and speed, every 30s**
  - The position given as the distance in feet from the beginning of the road
- **The output stream has tuples with carID and price to pay for a segment, any time a car enters that segment**
- **The total price to pay for a segment is as follows:**
  - a non-congested segment pays nothing
  - a congested segment pays $2 \times (numCars-50)^2$, where *numCars* is the number of cars in the segment at the moment
  - a seg. is congested if the average speed of all cars in the last 5 minutes is less than 40 Miles/h
- **Lets implement this in CQL!**

Linear Road in CQL

5280 feet in a Mile

- **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

Linear Road in CQL

5280 feet in a Mile

- **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

Linear Road in CQL

- **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

- **Define `ActCarSegRel` as the relation with the current segments of cars in the highway**

```
select carID, segNo
from SegSpeedStr [Range 30s];
```

Linear Road in CQL

S ■ **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

R ■ **Define `ActCarSegRel` as the relation with the current segments of cars in the highway**

```
select carID, segNo
from SegSpeedStr [Range 30s];
```

Linear Road in CQL

5280 feet in a Mile

S ■ **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

R ■ **Define `ActCarSegRel` as the relation with the current segments of cars in the highway**

```
select carID, segNo
from SegSpeedStr [Range 30s];
```

■ **Define the stream `CarSegEntryStr` signalling cars that enter the highway**

```
select Istream(*)
from ActCarSegRel;
```

Linear Road in CQL

S ■ **Start by defining a query `SegSpeedStr` that in the input stream replaces the position by the number of the segment**

```
select Istream(carID, speed, position/5280 as segNo)
from PosSpeedStr [Range Unbounded];
```

R ■ **Define `ActCarSegRel` as the relation with the current segments of cars in the highway**

```
select carID, segNo
from SegSpeedStr [Range 30s];
```

S ■ **Define the stream `CarSegEntryStr` signalling cars that enter the highway**

```
select Istream(*)
from ActCarSegRel;
```

Linear Road in CQL

- **Define `CongSegRel` as the relation containing, at any instant, this list of all congested segments**

Linear Road in CQL

- **Define `CongSegRel` as the relation containing, at any instant, this list of all congested segments**

a seg. is congested if the **average speed** of all cars in the **last 5 minutes** is less than 40 Miles/h

Linear Road in CQL

- **Define `CongSegRel` as the relation containing, at any instant, this list of all congested segments**

a seg. is congested if the **average speed** of all cars in the **last 5 minutes** is less than 40 Miles/h

```
select segNo
from SegSpeedStr [Range 5min]          (carID, speed, segNo)
group by segNo
having avg(speed) < 40;
```

Linear Road in CQL

- **Define `CongSegRel` as the relation containing, at any instant, this list of all congested segments**

a seg. is congested if the **average speed** of all cars in the **last 5 minutes** is less than 40 Miles/h

```
select segNo
from SegSpeedStr [Range 5min]                    (carID, speed, segNo)
group by segNo
having avg(speed) < 40;
```

- **Define `SegVolRel` as the relation with the current number of cars in each segment**

Linear Road in CQL

- **Define `CongSegRel` as the relation containing, at any instant, this list of all congested segments**

a seg. is congested if the **average speed** of all cars in the **last 5 minutes** is less than 40 Miles/h

```
select segNo
from SegSpeedStr [Range 5min]                    (carID, speed, segNo)
group by segNo
having avg(speed) < 40;
```

- **Define `SegVolRel` as the relation with the current number of cars in each segment**

```
select segNo, count(carID) as numCars
from ActCarSegRel                                (segNo, numCars)
group by segNo
```

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

- **SegSpeedStr - (carID, speed, segNo)**

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

- **SegSpeedStr - (carID, speed, segNo)**
- **ActCarSegRel - current segments of cars - (carID, speed, segNo) – Last 30s**

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

- **SegSpeedStr - (carID, speed, segNo)**
- **ActCarSegRel - current segments of cars - (carID, speed, segNo) – Last 30s**
- **CarSegEntryStr - signalling cars that enter the highway - (carID, speed, segNo)**

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

- **SegSpeedStr - (carID, speed, segNo)**
- **ActCarSegRel - current segments of cars - (carID, speed, segNo) – Last 30s**
- **CarSegEntryStr - signalling cars that enter the highway - (carID, speed, segNo)**
- **CongestedSegRel - at any instant, this list of all congested segments - (segNo)**

Linear Road in CQL

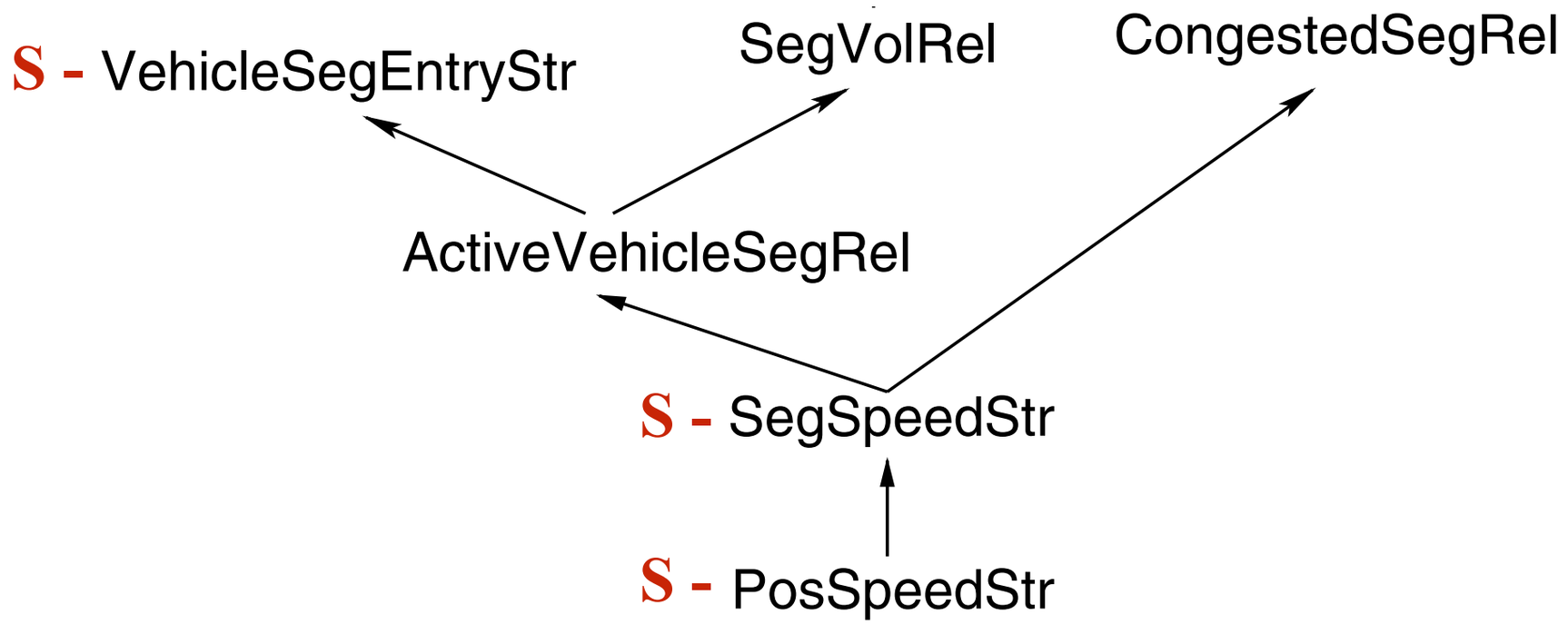
PosSpeedStr(carID, speed, segNo)

- **SegSpeedStr - (carID, speed, segNo)**
- **ActCarSegRel - current segments of cars - (carID, speed, segNo) – Last 30s**
- **CarSegEntryStr - signalling cars that enter the highway - (carID, speed, segNo)**
- **CongestedSegRel - at any instant, this list of all congested segments - (segNo)**
- **SegVolRel - the current number of cars in each segment - (segNo, numCars)**

Linear Road in CQL

PosSpeedStr(carID, speed, segNo)

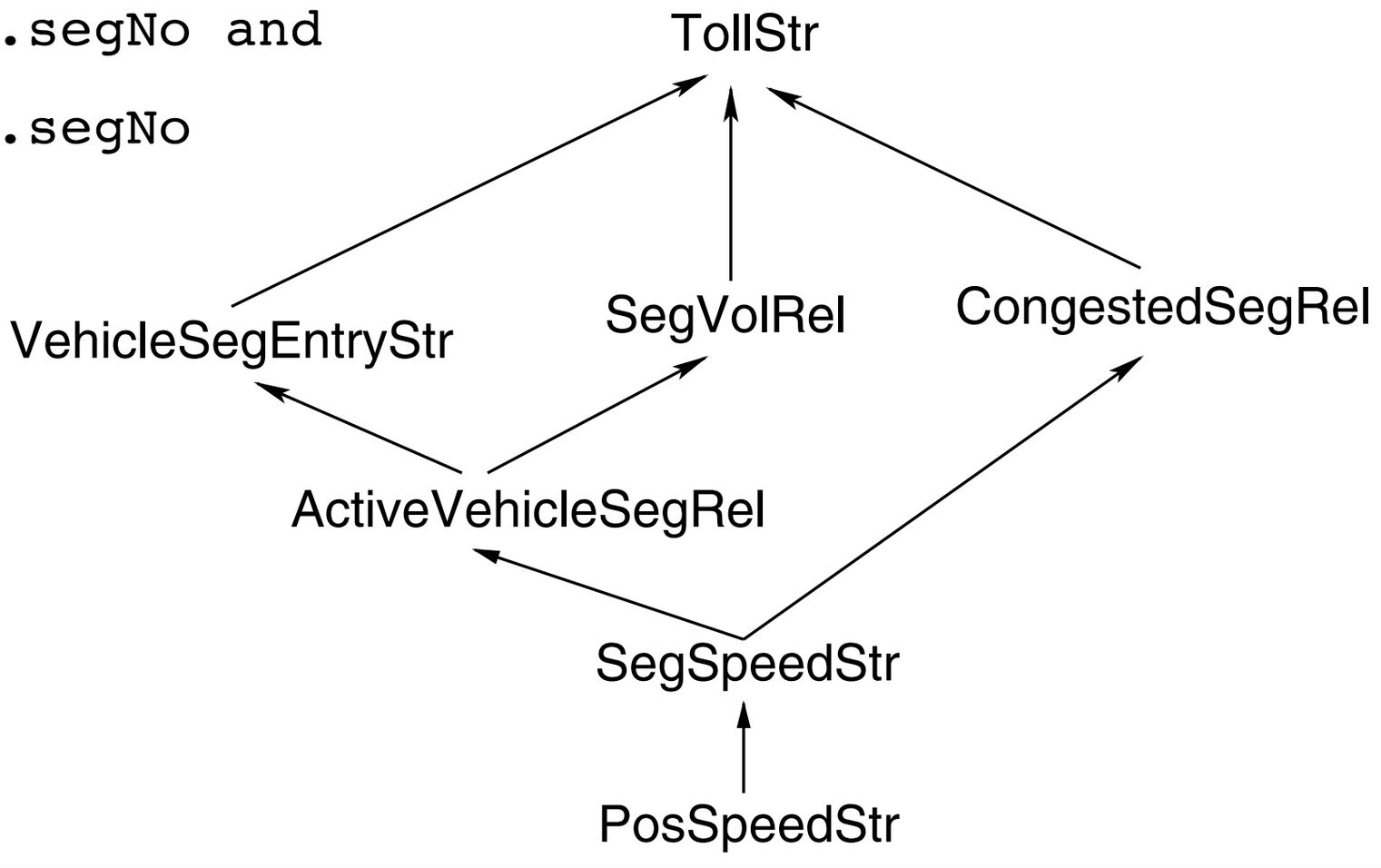
- **SegSpeedStr - (carID, speed, segNo)**
- **ActCarSegRel - current segments of cars - (carID, speed, segNo) – Last 30s**
- **CarSegEntryStr - signalling cars that enter the highway - (carID, speed, segNo)**
- **CongestedSegRel - at any instant, this list of all congested segments - (segNo)**
- **SegVolRel - the current number of cars in each segment - (segNo, numCars)**



Linear Road in CQL

- **The query that outputs the desired stream is:**

```
select Rstream(E.carID,2*(V.numCars-50)**2)
from CarSegEntryStr [Now] as E,
     CongestedSegRel as C, SegVolRel as V
where E.segNo = C.segNo and
      C.segNo = V.segNo
```



The complete query (just for the sake of completeness)

```
select Rstream(E.carID,2*(V.numCars-50)**2)
from
   ( select Istream(*)
     from
     ( select carID, segNo
       from
       ( select Istream(carID, speed,position/1000 as segNo) from PosSpeedStr [Range Unbounded]
       ) [Range 30s]
     )
   ) [Now] as E,
   ( select segNo
     from
       ( select Istream(*)
         from
         ( select Istream(carID, speed,position/1000 as segNo) from PosSpeedStr [Range Unbounded]
         ) [Range 30s]
       ) [Range 5min]
     group by segNo
     having avg(speed) < 40
   ) as C,
   ( select segNo, count(carID) as numCar
     from (select carID, segNo
           from (select Istream(carID, speed,position/1000 as segNo) from PosSpeedStr [Range Unbounded]
                )[Range 30s] )
     group by segNo
   ) as V
where E.segNo = C.segNo and C.segNo = V.segNo;
```

Some variants

- **Additional fixed price per segment (when segment is congested)**

```
select Rstream(E.carID,P.price+2*(V.numCars-50)**2)
from CarSegEntryStr [Now] as E,
     CongSegRel as C, SegVolRel as V, SegPrice as P
where E.segNo = C.segNo and
      C.segNo = V.segNo and V.segNo = P.segNo;
```

```
select Rstream(E.carID,2*(V.numCars-50)**2)
from CarSegEntryStr [Now] as E,
     CongestedSegRel as C, SegVolRel as V
where E.segNo = C.segNo and
      C.segNo = V.segNo
```

Some variants

- **Charging the customer: having the prices being output upon exit of the cars**

```
select Rstream(A.NIF,P.price+2*(V.numCars-50)**2)
from CarSegEntryStr [Now] as E,
     CongSegRel as C, SegVolRel as V,
     SegPrice as P, Cars, Accounts as A
where E.segNo = C.segNo and
      C.segNo = V.segNo and V.segNo = P.segNo
      E.carID = Cars.ID and Cars.owner = A.client;
```

Another variant

- **Fixed price per segment (charged always), plus variable toll:**

```
select Rstream(E.carID,P.price+2*(V.numCars-50)**2)
from
   (CarSegEntryStr [Now] E inner join SegPrice S
       on(E.segNo = S.segNo))
   left outer join
   (CongSegRel C inner join SegVolRel V
       on(C.segNo = V.segNo))
on(E.segNo = C.segNo)
```

Other variants

- **If a cars is detected in segment N and then in segment N+2, within a time frame of 30 minutes, and in-between it is never detected in segment N+1, then signal a possible failure**

Other variants

- **If a cars is detected in segment N and then in segment N+2, within a time frame of 30 minutes, and in-between it is never detected in segment N+1, then signal a possible failure**
  - This may be problematic…

Other variants

- **If a cars is detected in segment N and then in segment N+2, within a time frame of 30 minutes, and in-between it is never detected in segment N+1, then signal a possible failure**
  - This may be problematic…
- **If a car is detected in a segment N, and the last two times it was detected was in segment N+1, then warn the police (it may be driving in the wrong way**

Other variants

- **If a cars is detected in segment N and then in segment N+2, within a time frame of 30 minutes, and in-between it is never detected in segment N+1, then signal a possible failure**
  - This may be problematic…
- **If a car is detected in a segment N, and the last two times it was detected was in segment N+1, then warn the police (it may be driving in the wrong way**
  - Again this may be problematic
  - Note that when one applies a s2r operator, the relation loses the timestamps

Some common queries

- **Stream filters:**
  - Either use
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`

Some common queries

- **Stream filters:**
  - Either use
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`
  - Note the **Istream-Unbounded** window combination is the **default** for streams whose window specification is omitted

Some common queries

- **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**
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`

Some common queries

- **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**
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`
  - But these two ways may have different meanings!

Some common queries

- **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**
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`
  - But these two ways may have different meanings!

- **Example: Get prices for purchased items:**

```
select Rstream(item.id, prices.price)
from item [Now], prices
where item.id=prices.id;
```

```
select Istream(item.id, prices.price)
from item [Range Unbounded], prices
where item.id=prices.id;
```

- **Differences?**

Some common queries

- **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**
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`
  - But these two ways may have different meanings!

- **Example: Get prices for purchased items:**

```
select Rstream(item.id, prices.price)
from item [Now], prices
where item.id=prices.id;
```

```
select Istream(item.id, prices.price)
from item [Range Unbounded], prices
where item.id=prices.id;
```

- **Differences?**
  - The 1st outputs an item with price, whenever a new item arrives

Some common queries

- **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**
    - `Rstream` and `[Now]`, or
    - `Istream` and `[Range Unbounded]`
  - But these two ways may have different meanings!

- **Example: Get prices for purchased items:**

```
select Rstream(item.id, prices.price)
from item [Now], prices
where item.id=prices.id;
```

```
select Istream(item.id, prices.price)
from item [Range Unbounded], prices
where item.id=prices.id;
```

- **Differences?**
  - The 1st outputs an item with price, whenever a new item arrives
  - The 2nd does that plus, whenever a price changes, it outputs all the items purchased so far, with the new price

Some common queries

- **Sliding windows joins:**

```
select Istream(*)

from S1 [Rows 5], S2 [Rows 10]

where S1.A=S2.A;
```

Some common queries

- **Sliding windows joins:**

```
select Istream(*)

from S1 [Rows 5], S2 [Rows 10]

where S1.A=S2.A;
```

  - A new tuple is produced whenever a tuple in the last 5 elements of S1 matches with one in the last 10 elements of S2

Some common queries

- **Sliding windows joins:**

```
select Istream(*)

from S1 [Rows 5], S2 [Rows 10]

where S1.A=S2.A;
```

  - A new tuple is produced whenever a tuple in the last 5 elements of S1 matches with one in the last 10 elements of S2

- **But if both S1 and S2 can have duplicates in attribute A, it may work differently!**

Some common queries

- **Sliding windows joins:**

```
select Istream(*)

from S1 [Rows 5], S2 [Rows 10]

where S1.A=S2.A;
```

  - A new tuple is produced whenever a tuple in the last 5 elements of S1 matches with one in the last 10 elements of S2

- **But if both S1 and S2 can have duplicates in attribute A, it may work differently!**

  - Suppose a new tuple of S2 is identical to the tuple 10 positions earlier

Some common queries

- **Sliding windows joins:**

```
select Istream(*)

from S1 [Rows 5], S2 [Rows 10]

where S1.A=S2.A;
```

  - A new tuple is produced whenever a tuple in the last 5 elements of S1 matches with one in the last 10 elements of S2

- **But if both S1 and S2 can have duplicates in attribute A, it may work differently!**
  - 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.

Stream aggregations

Aggregation produces relations, not streams, in CQL since **aggregations** are **relation-to-relation operators**

- **Stream the value of aggregation whenever it changes:**

```
select Istream(Count(*))

from PosSpeedStr [Range 1min];
```

  - Counts the measurement over the previous minute, and outputs the count to the stream whenever it changes

Stream aggregations

Aggregation produces relations, not streams, in CQL since **aggregations** are **relation-to-relation operators**

- **Stream the value of aggregation whenever it changes:**

```
select Istream(Count(*))

from PosSpeedStr [Range 1min];
```

  - Counts the measurement over the previous minute, and outputs the count to the stream whenever it changes

- **Stream the value of aggregation periodically:**

```
select Istream(Count(*))

from PosSpeedStr [Range 1min Slide 1min];
```

  - Outputs the number of measurements over the last minute, once a minute

Stream aggregations

Aggregation produces relations, not streams, in CQL since **aggregations** are **relation-to-relation operators**

- **Stream the value of aggregation whenever it changes:**

```
select Istream(Count(*))

from PosSpeedStr [Range 1min];
```

  - Counts the measurement over the previous minute, and outputs the count to the stream whenever it changes

- **Stream the value of aggregation periodically:**

```
select Istream(Count(*))

from PosSpeedStr [Range 1min Slide 1min];
```

  - Outputs the number of measurements over the last minute, once a minute
  - If the count does not change, then it output nothing!
    - It is possible, with a more complex query, also to output in that case.

Some equivalences

L is any **select-list**, *S* is any **stream** (including a subquery producing a stream), and *C* is any **condition**.

Some equivalences

L is any **select-list**, *S* is any **stream** (including a subquery producing a stream), and *C* is any **condition**.

- **Window reduction - the two are equivalent**

```
select Istream(L)
from S [Range Unbounded]
where Cond;
```

```
select Rstream(L)
from S [Now]
where Cond;
```

  - the execution of the query on the right should be more efficient (it doesn't require buffering of stream results)

Some equivalences

L is any **select-list**, *S* is any **stream** (including a subquery producing a stream), and *C* is any **condition**.

- **Window reduction - the two are equivalent**

```
select Istream(L)
from S [Range Unbounded]
where Cond;
```

```
select Rstream(L)
from S [Now]
where Cond;
```

  - the execution of the query on the right should be more efficient (it doesn't require buffering of stream results)

- **Filter-Window commutativity - the two are equivalent**

```
(select L from S where C) [Range T]
```

```
select L from S [Range T] where C
```

  - the execution of the 1st query should be more efficient, because it filters before applying the window
    - If the system uses a query strategy that materialises the window, materialising only the filtered tuples requires less steady-state memory

On CQL implementation

On CQL implementation

- **CQL was first implement in the prototype system STREAM**
  - Currently there are other implementations (e.g. Oracle)
- **One of the nice things about declarative languages is that they hide quite a lot of the implementation details**
  - We can use them without having any idea of how things are implemented

On CQL implementation

- **CQL was first implement in the prototype system STREAM**
  - Currently there are other implementations (e.g. Oracle)
- **One of the nice things about declarative languages is that they hide quite a lot of the implementation details**
  - We can use them without having any idea of how things are implemented
- **But, in the end, the declarative language must be implemented!**
  - If one wants to use the system efficiently, it is good to have an idea of how things are implemented, and run

CQL Query Plans

- **When a CQL query is registered, a query plan is generated**

CQL Query Plans

- **When a CQL query is registered, a query plan is generated**
  - The query plan is a **tree/graph** with **operators**, roughly corresponding to the operators in the query's algebraic expression (pretty much as in DBMSs)

CQL Query Plans

- **When a CQL query is registered, a query plan is generated**
  - The query plan is a **tree/graph** with **operators**, roughly corresponding to the operators in the query's algebraic expression (pretty much as in DBMSs)
  - The first plan is generated taking into account several simple heuristics
    - But while running, with concrete data, it may be adapted

CQL Query Plans

- **When a CQL query is registered, a query plan is generated**
  - The query plan is a **tree/graph** with **operators**, roughly corresponding to the operators in the query's algebraic expression (pretty much as in DBMSs)
  - The first plan is generated taking into account several simple heuristics
    - But while running, with concrete data, it may be adapted
  - A **new query plan is then merged with existing ones**
    - This way, **parts of the plan that are common to more than one query are only (continuously) executed once**
    - At least the parts of incoming streams, are usually common

CQL Query Execution

- **Query execution strategy is based mostly on *incremental processing*:**

CQL Query Execution

- **Query execution strategy is based mostly on *incremental processing*:**
  - **s2r** operators transform **new items in the input stream** into changes (**insertions and deletions**) in the **output relation**

CQL Query Execution

- **Query execution strategy is based mostly on *incremental processing*:**
  - **s2r** operators transform **new items in the input stream** into changes (**insertions and deletions**) in the **output relation**
  - **r2r** operators transform **changes** in the input relations into **changes** in the output relation.

CQL Query Execution

- **Query execution strategy is based mostly on *incremental processing*:**
  - **s2r** operators transform **new items in the input stream** into changes (**insertions and deletions**) in the **output relation**
  - **r2r** operators transform **changes** in the input relations into **changes** in the output relation.
  - **r2s** operators transform changes in the input relations into new items in the output stream

CQL Query Execution

- **Query execution strategy is based mostly on *incremental processing*:**
  - **s2r** operators transform **new items in the input stream** into changes (**insertions and deletions**) in the **output relation**
  - **r2r** operators transform **changes** in the input relations into **changes** in the output relation.
  - **r2s** operators transform changes in the input relations into new items in the output stream
- **We need components associated to operators, to:**
  - store **intermediate changes** - inserts and deletes (***queues***)
  - store **intermediate results** - e.g. with materialised windows (***synopses***)

Internal representation

- **Both streams and relations are internally represented as sequences of *tagged tuples***

Internal representation

- **Both streams and relations are internally represented as sequences of *tagged tuples***
- **A tagged tuple is a tuple plus a tag with:**
  - a timestamp
  - an indication whether it is an insertion of a deletion

Internal representation

- **Both streams and relations are internally represented as sequences of *tagged tuples***
- **A tagged tuple is a tuple plus a tag with:**
  - a timestamp
  - an indication whether it is an insertion of a deletion
- **Streams can only have insertion tags**

Internal representation

- **Both streams and relations are internally represented as sequences of *tagged tuples***
- **A tagged tuple is a tuple plus a tag with:**
  - a timestamp
  - an indication whether it is an insertion of a deletion
- **Streams can only have insertion tags**
- **Relation can have both.**
  - Relations are thus represented as a set of insert/delete operations that give rise to the current state of the relation

Queues and Synopses

- ***A queue* connects two operators O_i and O_o**
  - At any time, contains a sequence of tagged tuples
  - The queue buffers the insertions and deletions output by O_i until they are processed by O_o

Queues and Synopses

- ***A queue* connects two operators O_i and O_o**
  - At any time, contains a sequence of tagged tuples
  - The queue buffers the insertions and deletions output by O_i until they are processed by O_o
- ***A synopsis* is associated to one operator**
  - It stores an intermediate state needed by the operator
    - E.g. when joining two windowed streams, the join needs, at any time, to have the contents of the whole two windows
  - Synopses mostly materialise the content of relations resulting from windows, or store intermediate summaries (aggregations) of data

Queues and Synopses

- ***A queue* connects two operators O_i and O_o**
  - At any time, contains a sequence of tagged tuples
  - The queue buffers the insertions and deletions output by O_i until they are processed by O_o
- ***A synopsis* is associated to one operator**
  - It stores an intermediate state needed by the operator
    - E.g. when joining two windowed streams, the join needs, at any time, to have the contents of the whole two windows
  - Synopses mostly materialise the content of relations resulting from windows, or store intermediate summaries (aggregations) of data
- **Two operators are always connected by a queue, but many operators need no synopsis**

Example plan

- **Q1**

```
select B, max(A)
from S1 [Rows 50000]
group by B;
```

- **Q2**

```
select Istream(*)
from S1 [Rows 40000]
     S2 [Range 600s]
where S1.A = S2.A;
```

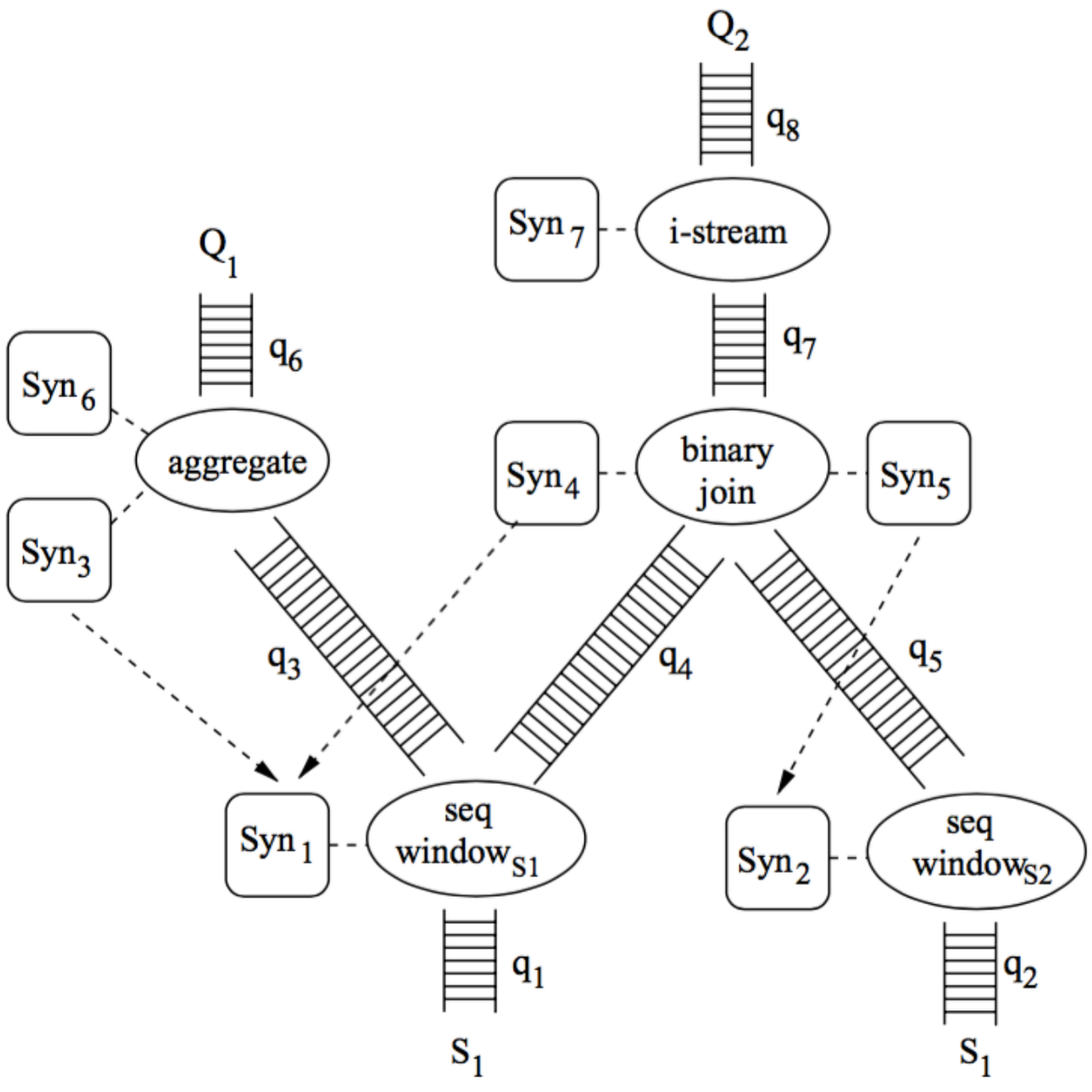
Example plan

- **Q1**

```
select B, max(A)

from S1 [Rows 50000]

group by B;
```

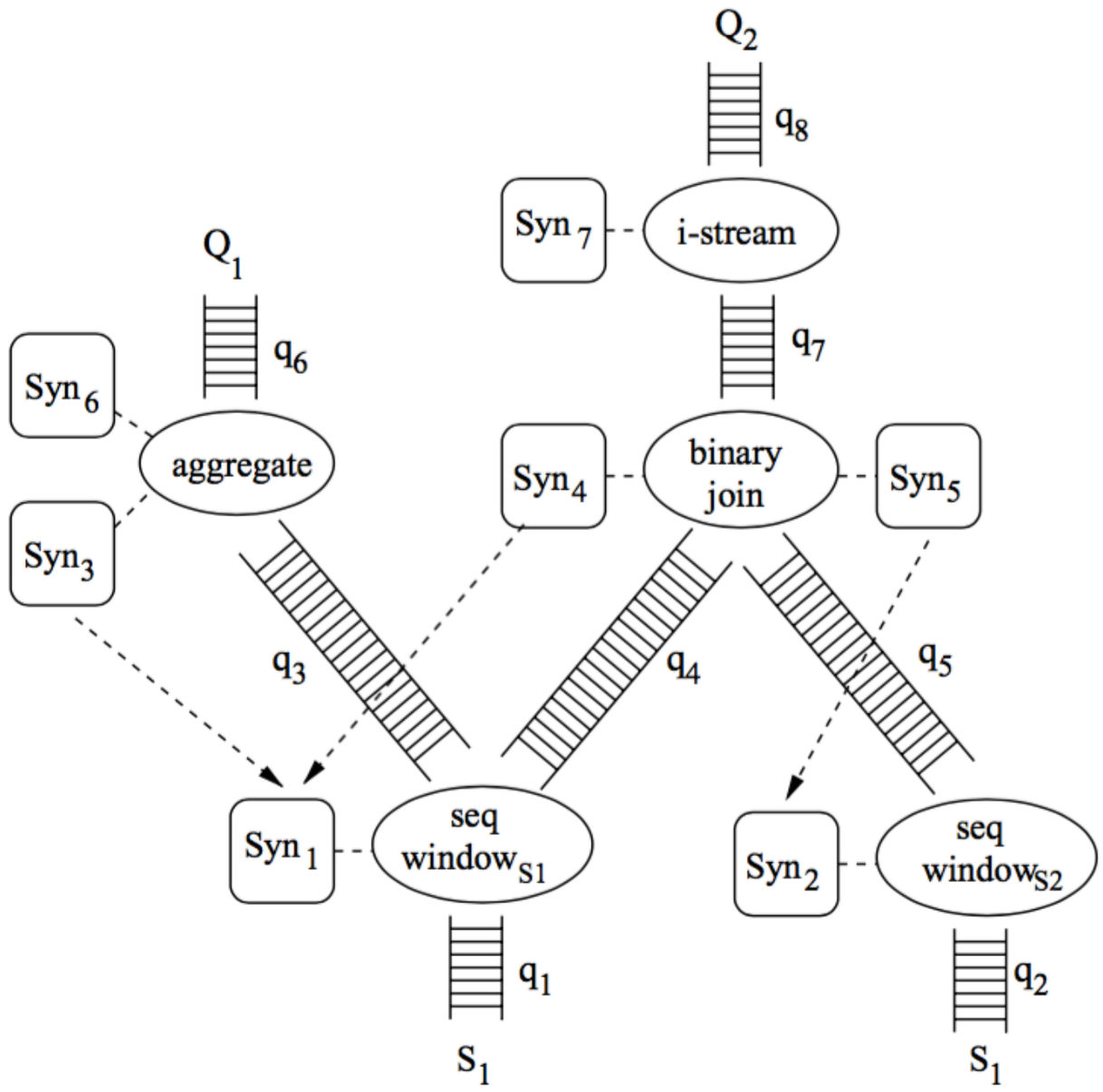
- **Q2**

```
select Istream(*)

from S1 [Rows 40000]

     S2 [Range 600s]

where S1.A = S2.A;
```



Example plan

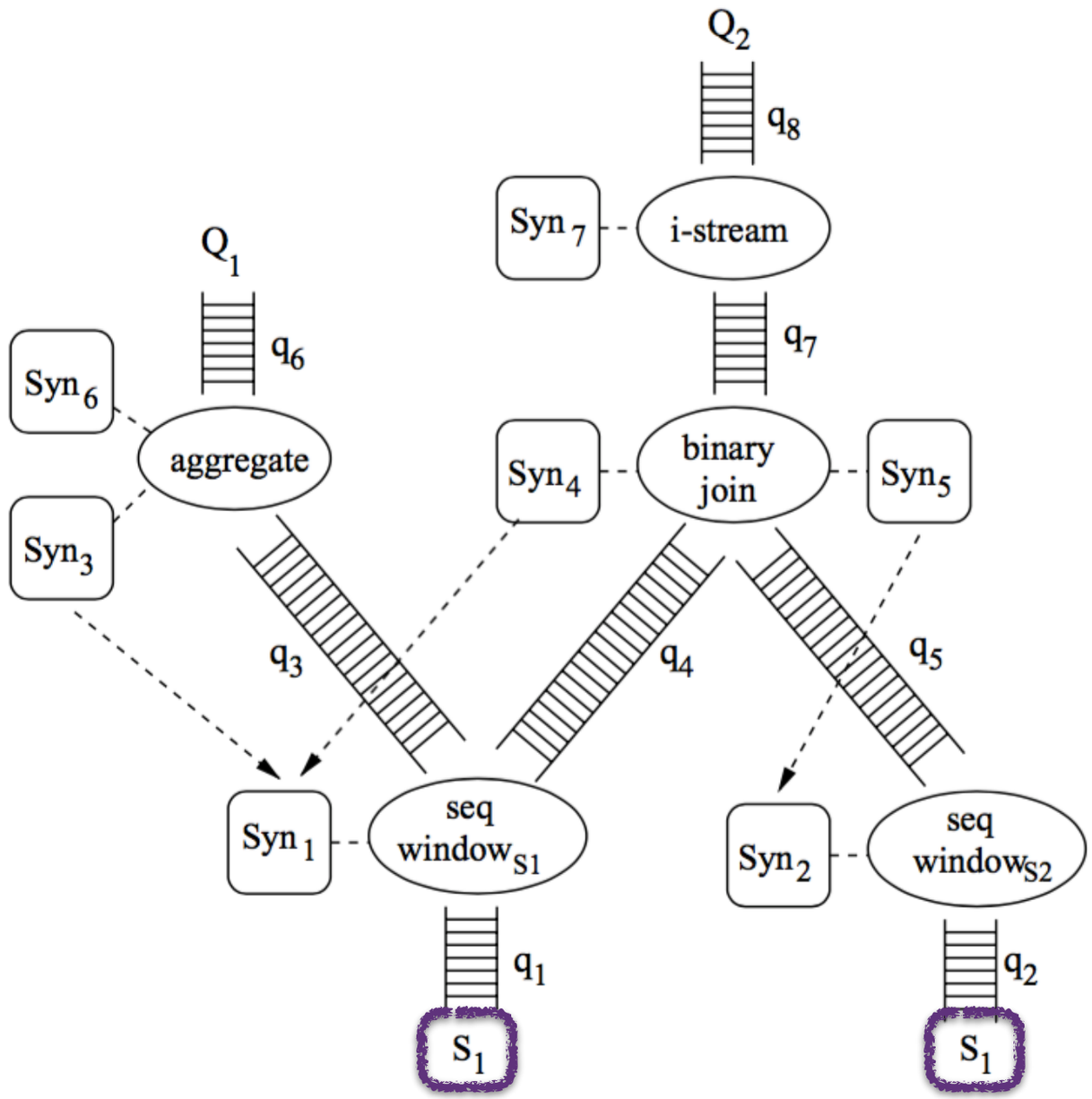
- **This plan contains**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

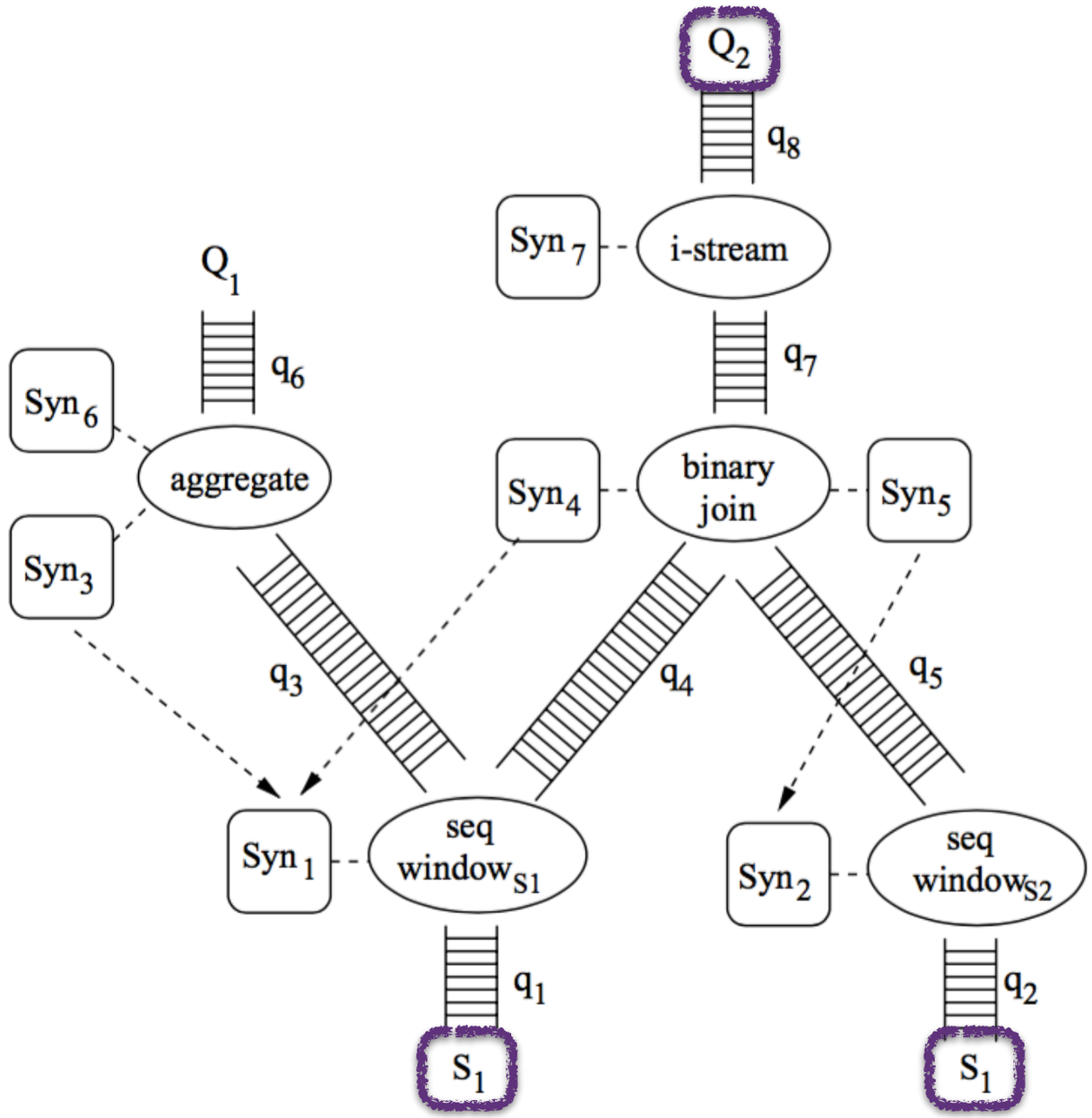
- **This plan contains**
  - 2 input streams




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

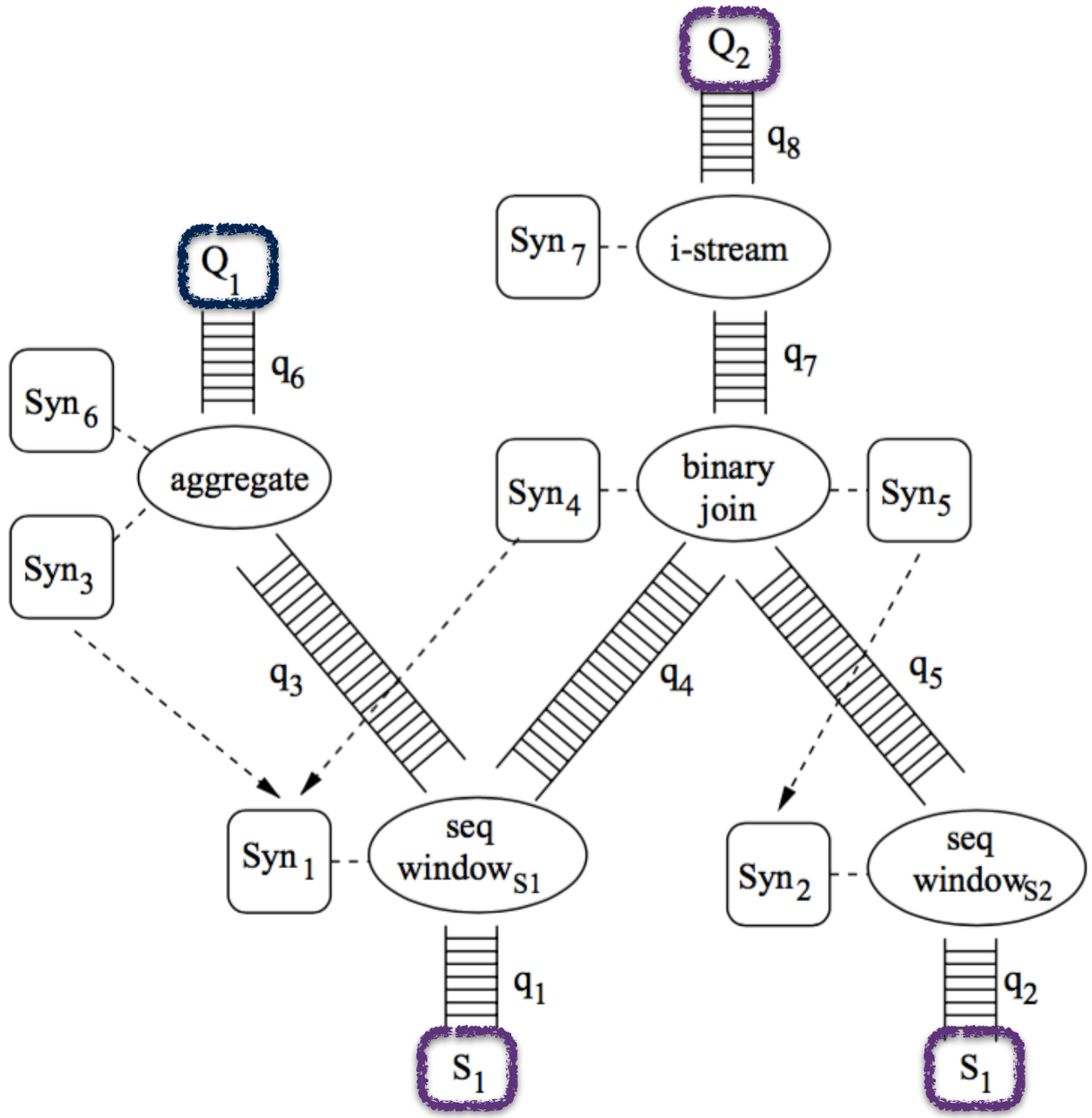
- **This plan contains**
  - 2 input streams
  - 1 output stream




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

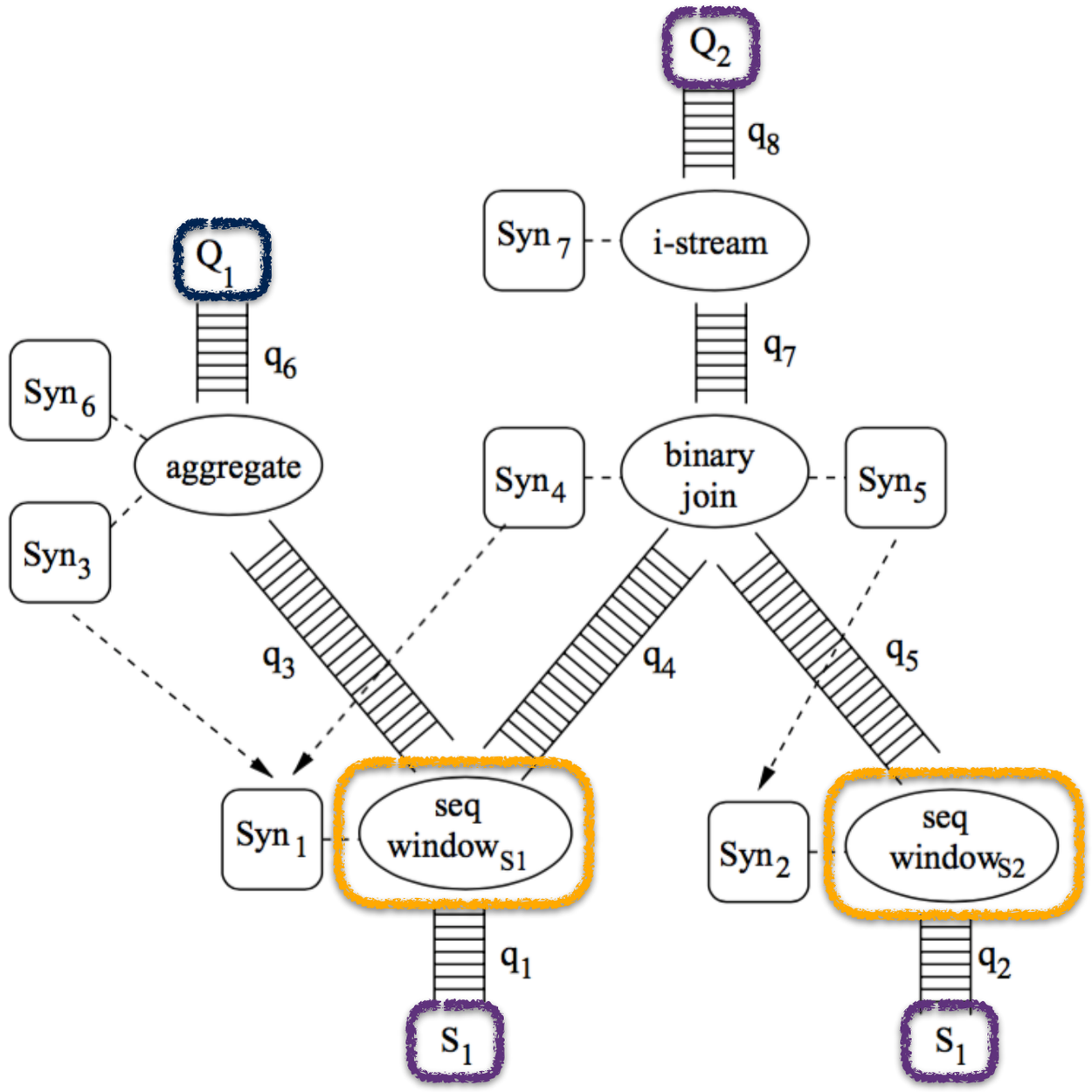
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

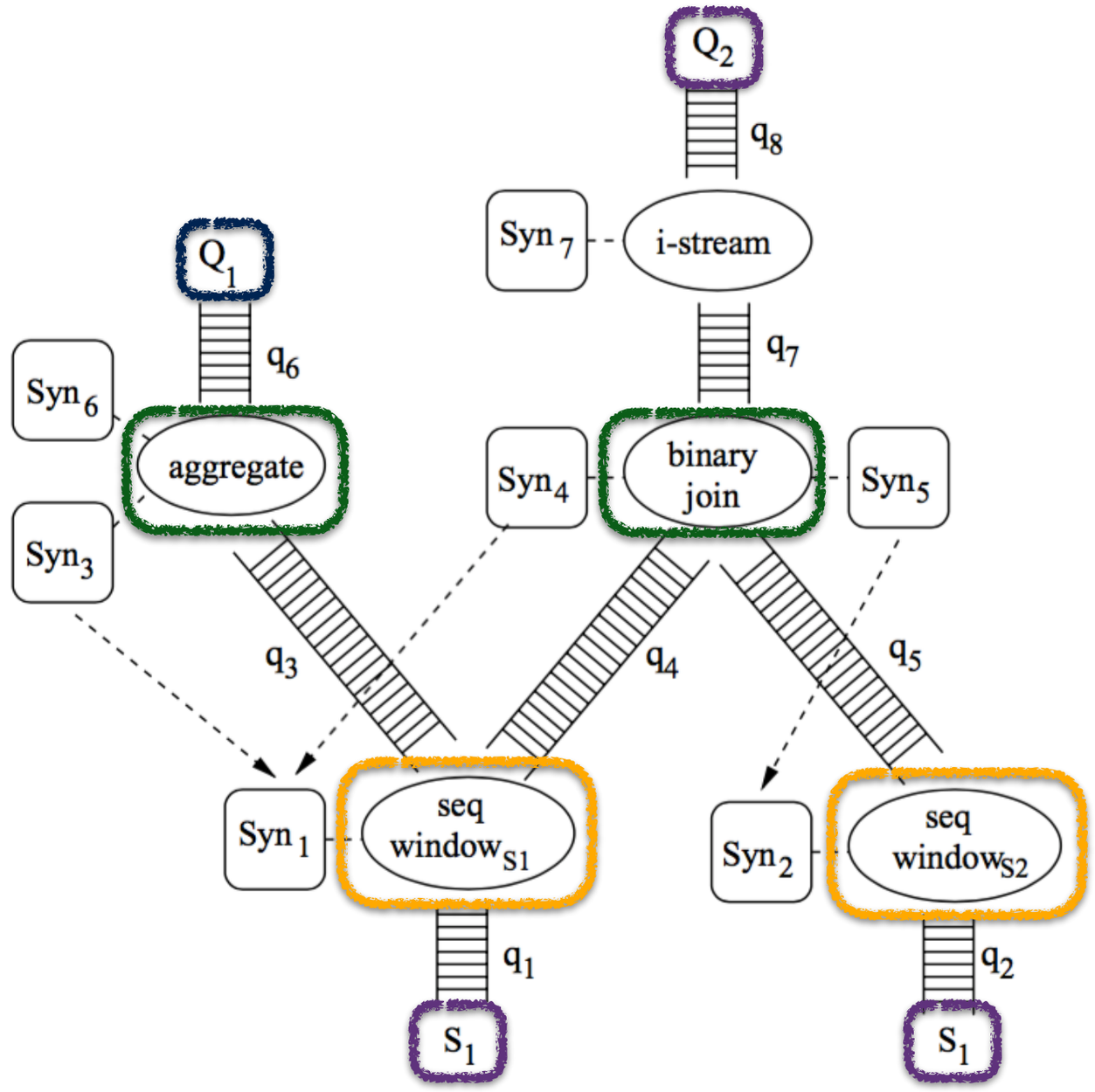
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation
  - 2 s2r operators




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

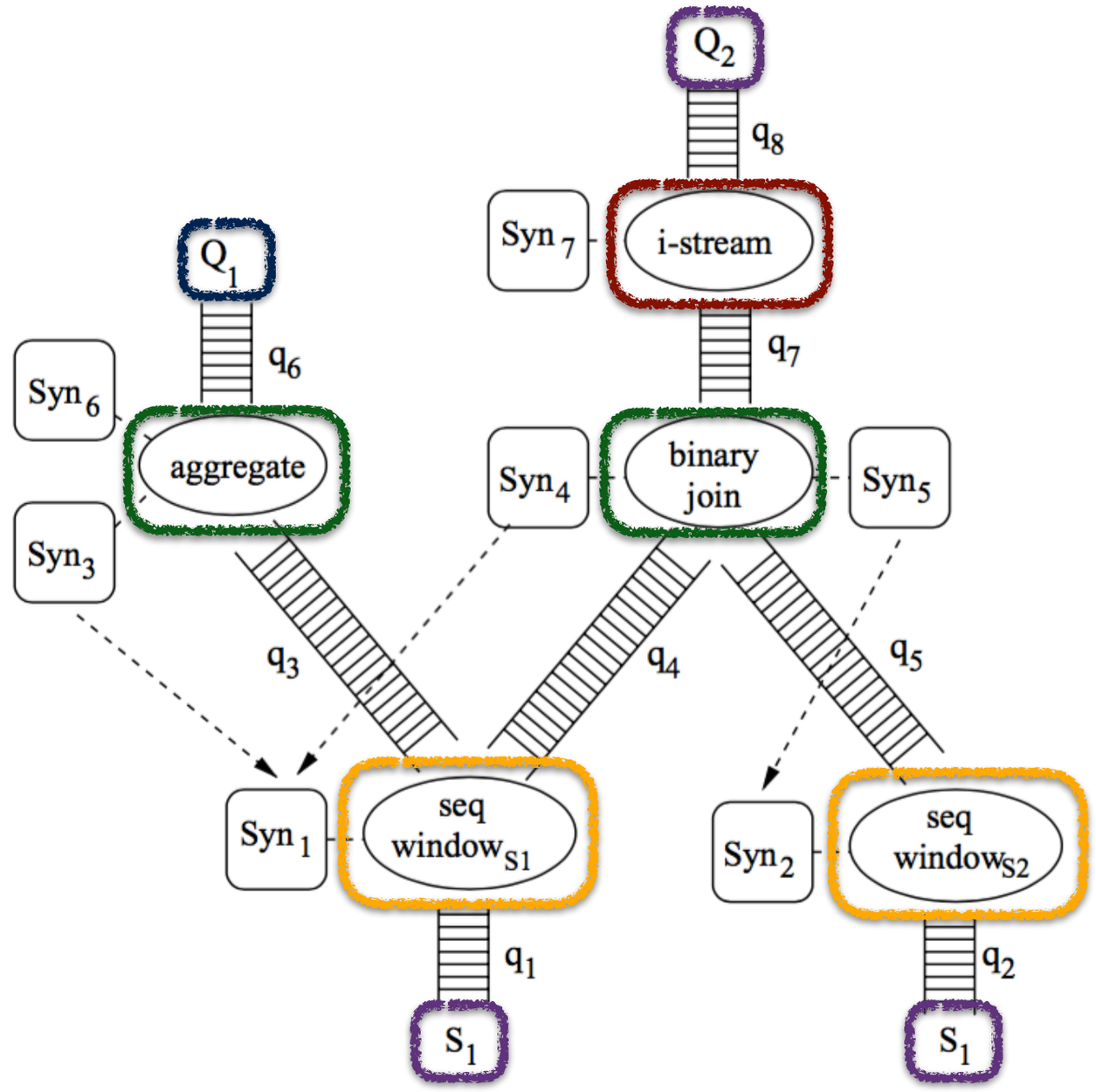
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation
  - 2 s2r operators
  - 2 r2r operators




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

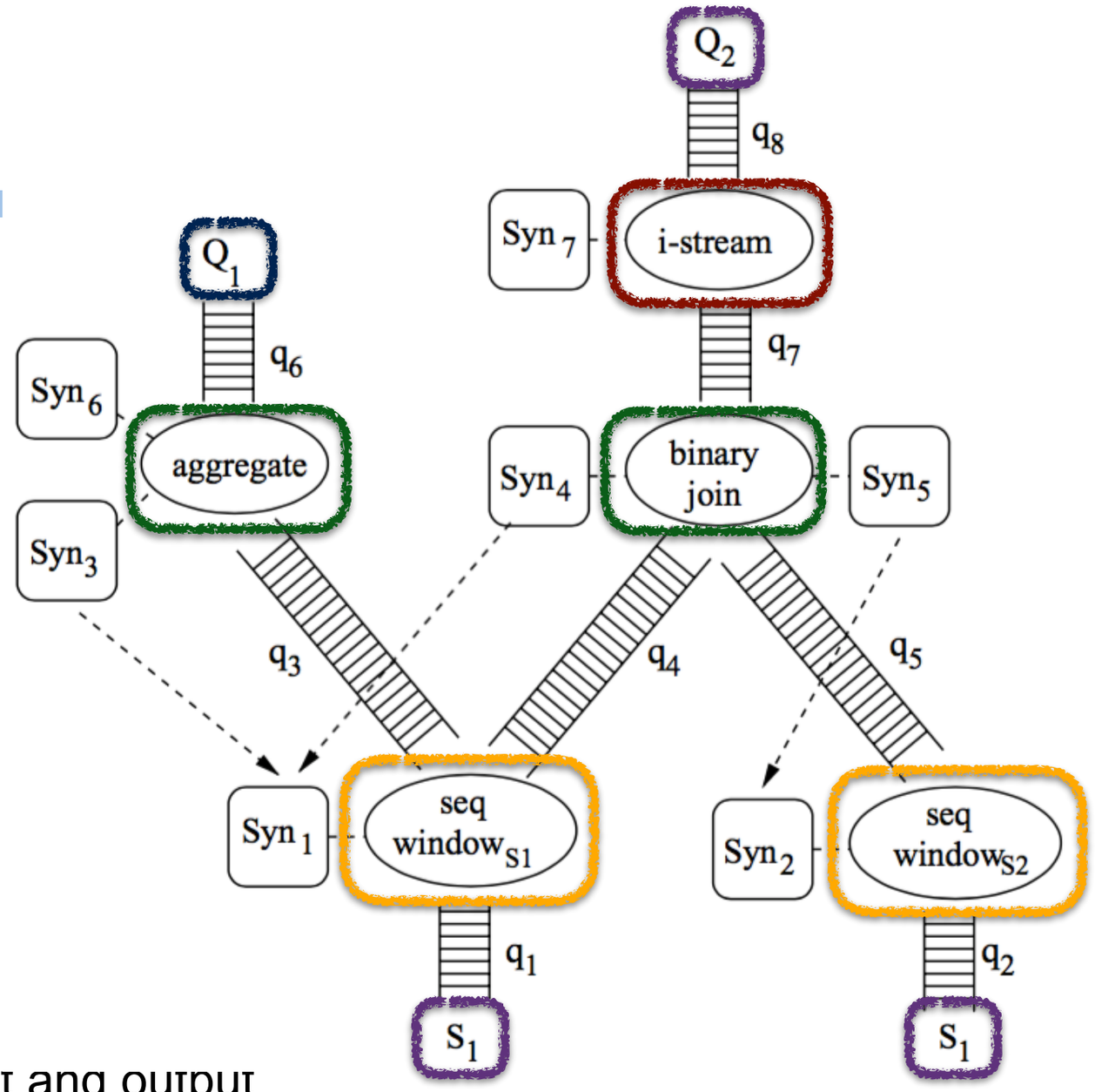
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation
  - 2 s2r operators
  - 2 r2r operators
  - 1 r2s operator




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

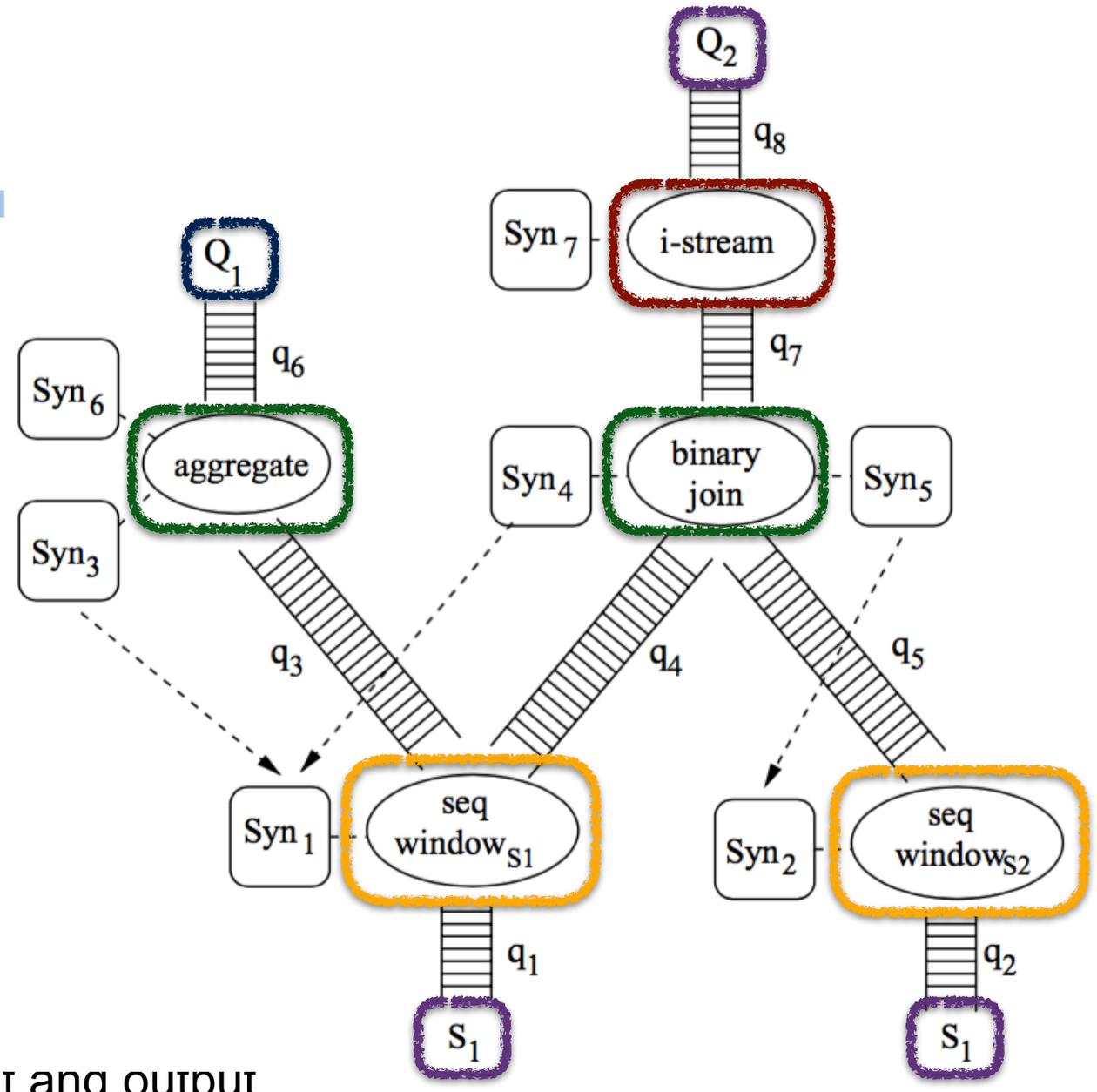
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation
  - 2 s2r operators
  - 2 r2r operators
  - 1 r2s operator
  - 8 queues connecting the operators, input and output




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

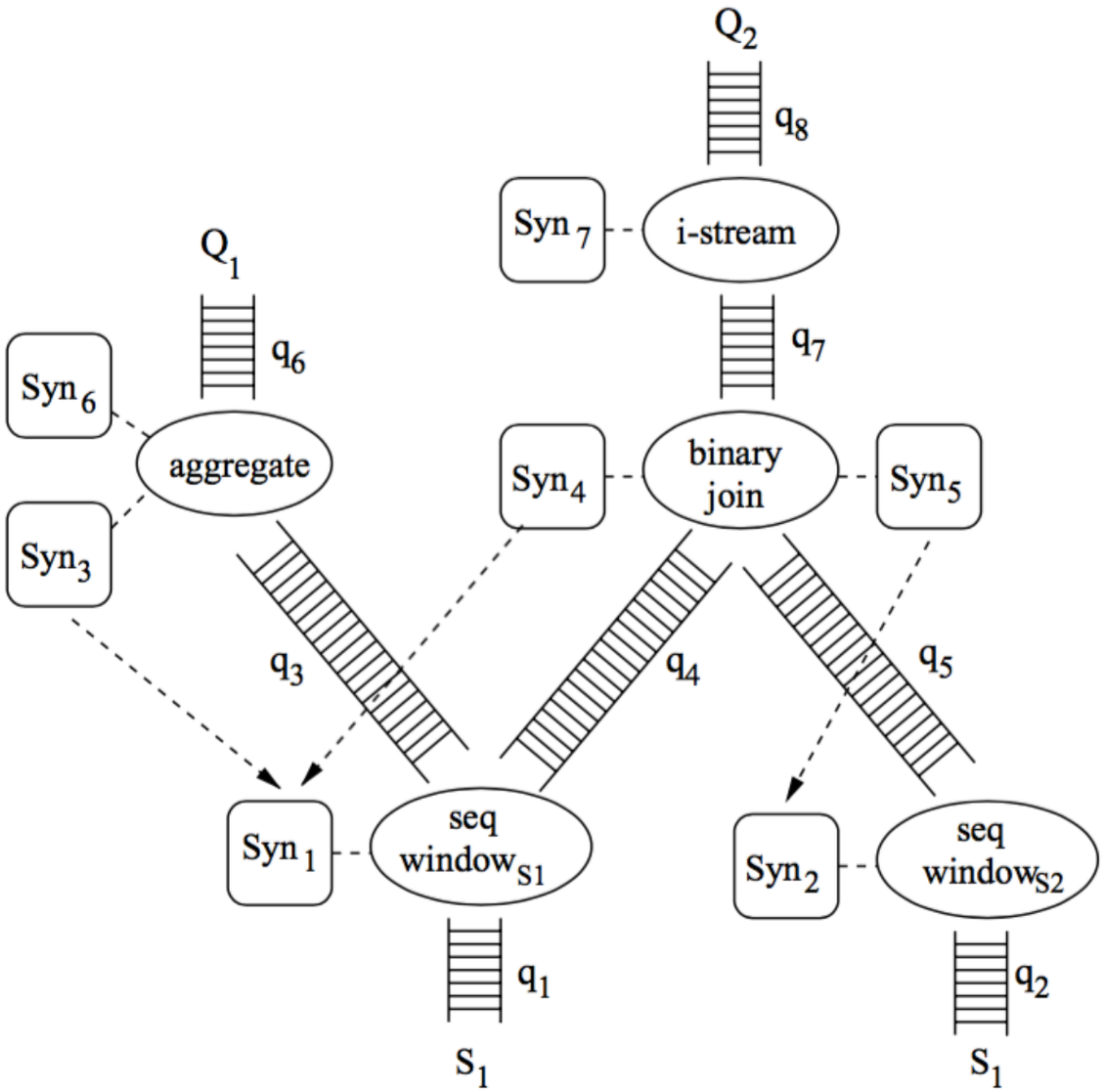
- **This plan contains**
  - 2 input streams
  - 1 output stream
  - 1 output relation
  - 2 s2r operators
  - 2 r2r operators
  - 1 r2s operator
  - 8 queues connecting the operators, input and output
  - 7 synopses associated to operators




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

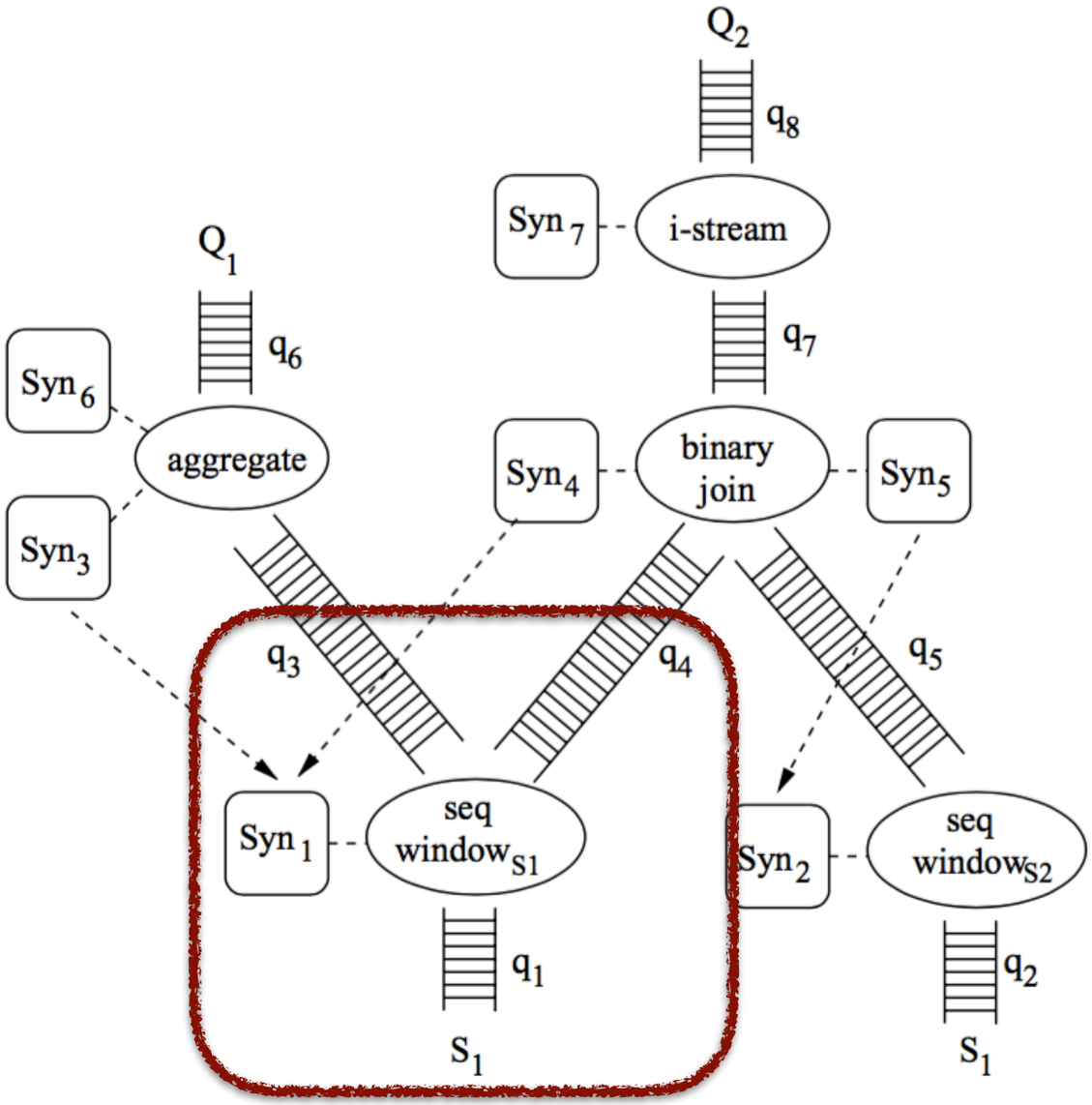
- **Sliding window operator seq-window$_{S1}$**
  - Reads stream S_1's tuples
  - Outputs *insertions and deletions in the window* in **both q_3 and q_4**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

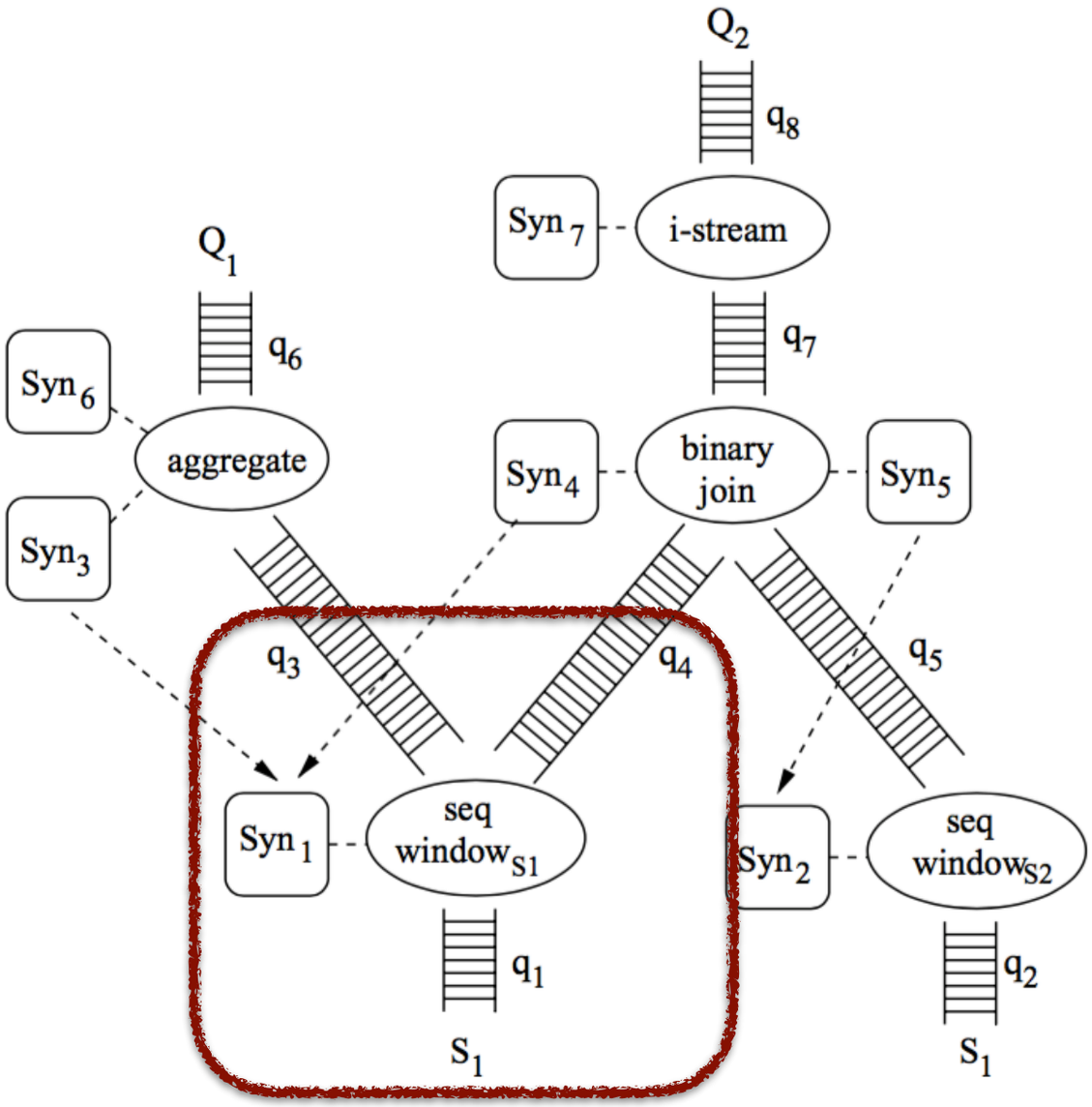
- **Sliding window operator seq-window$_{S1}$**
  - Reads stream S_1's tuples
  - Outputs *insertions and deletions in the window* in **both q_3 and q_4**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

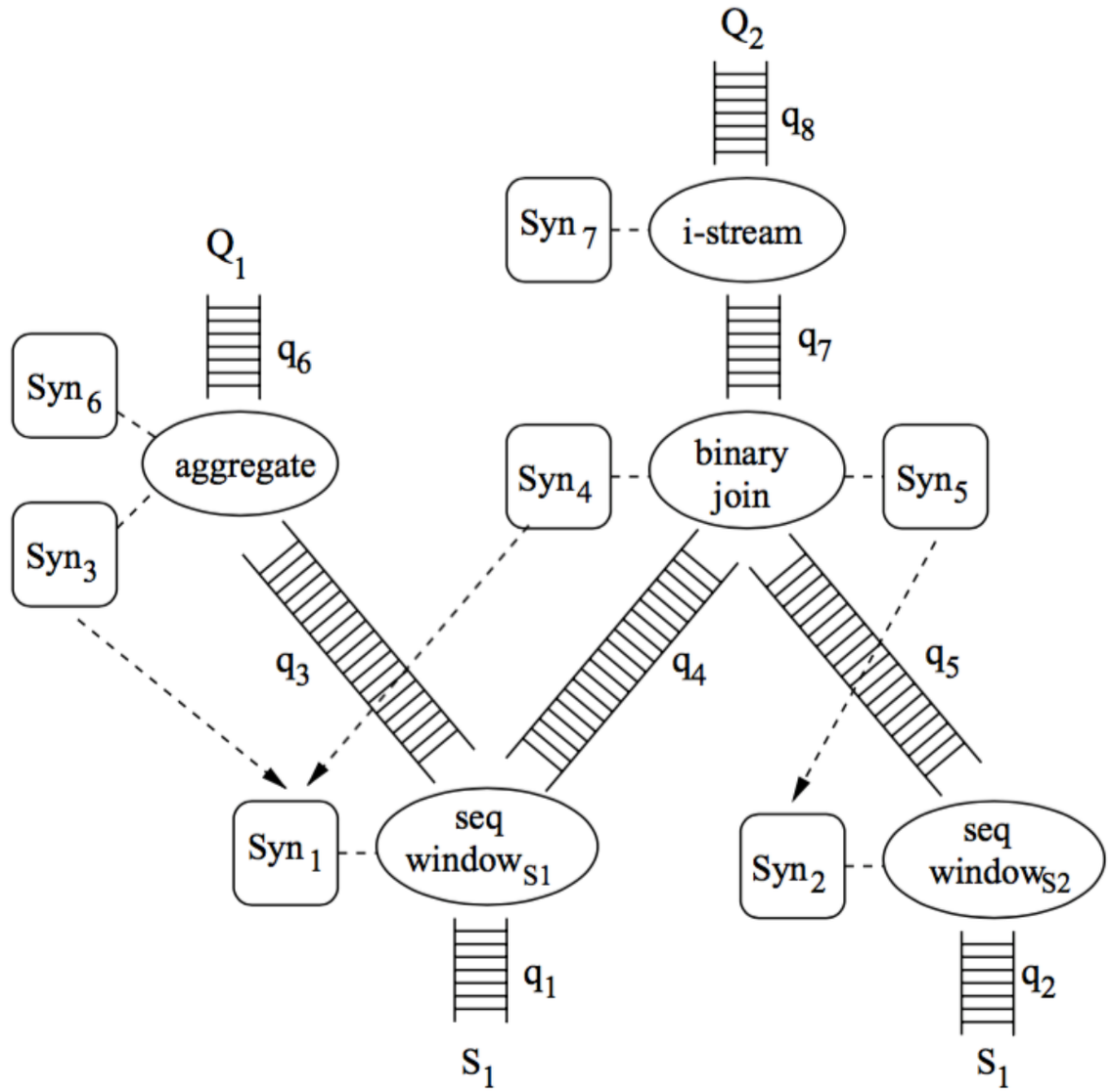
Example plan

- **Sliding window operator seq-window$_{S1}$**
  - Reads stream S_1's tuples
  - Outputs *insertions and deletions in the window* in **both q_3 and q_4**
- **Synopsis Syn$_1$ stores the last 50.000 tuples that arrived in Stream 1**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

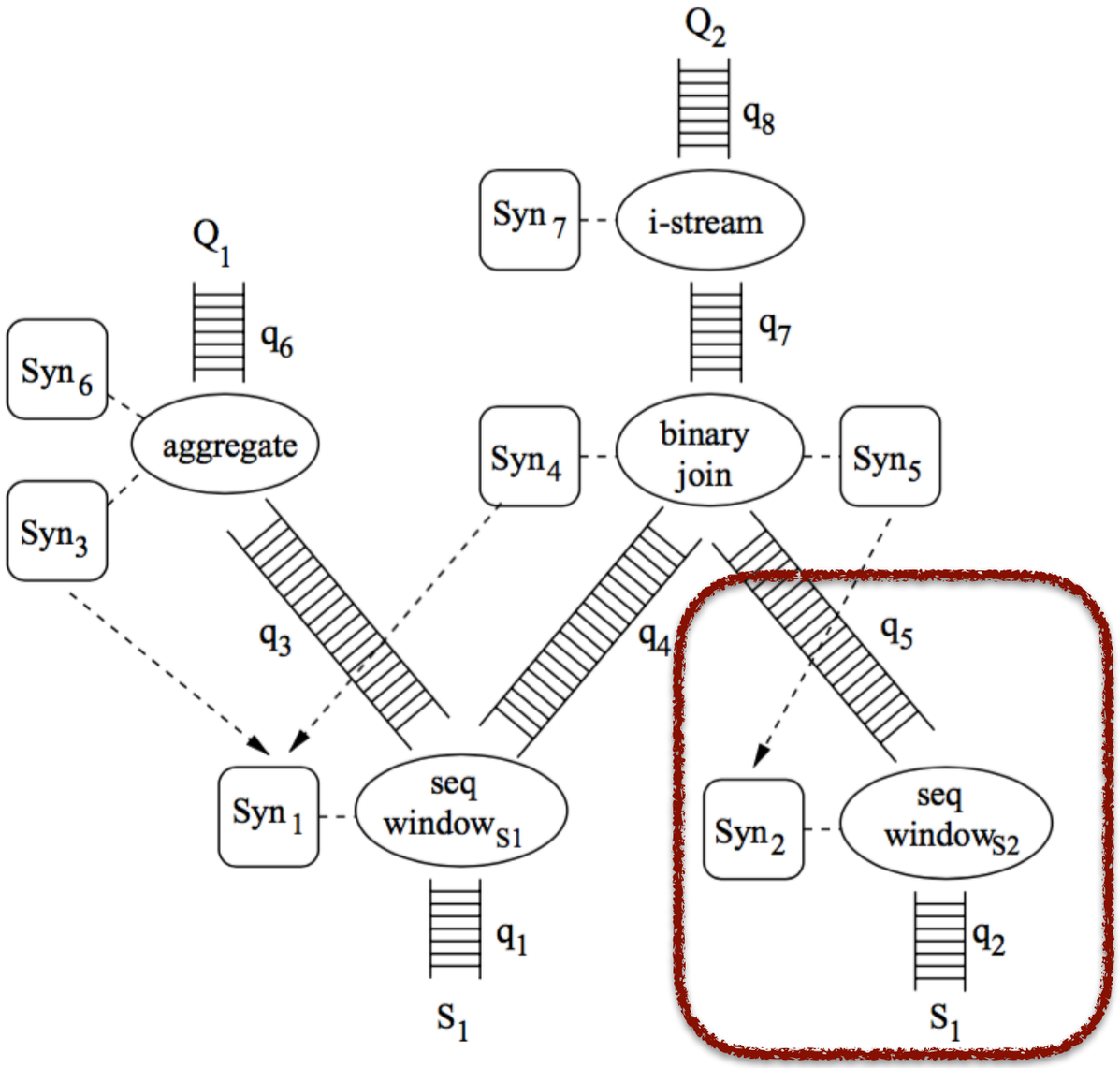
Example plan




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

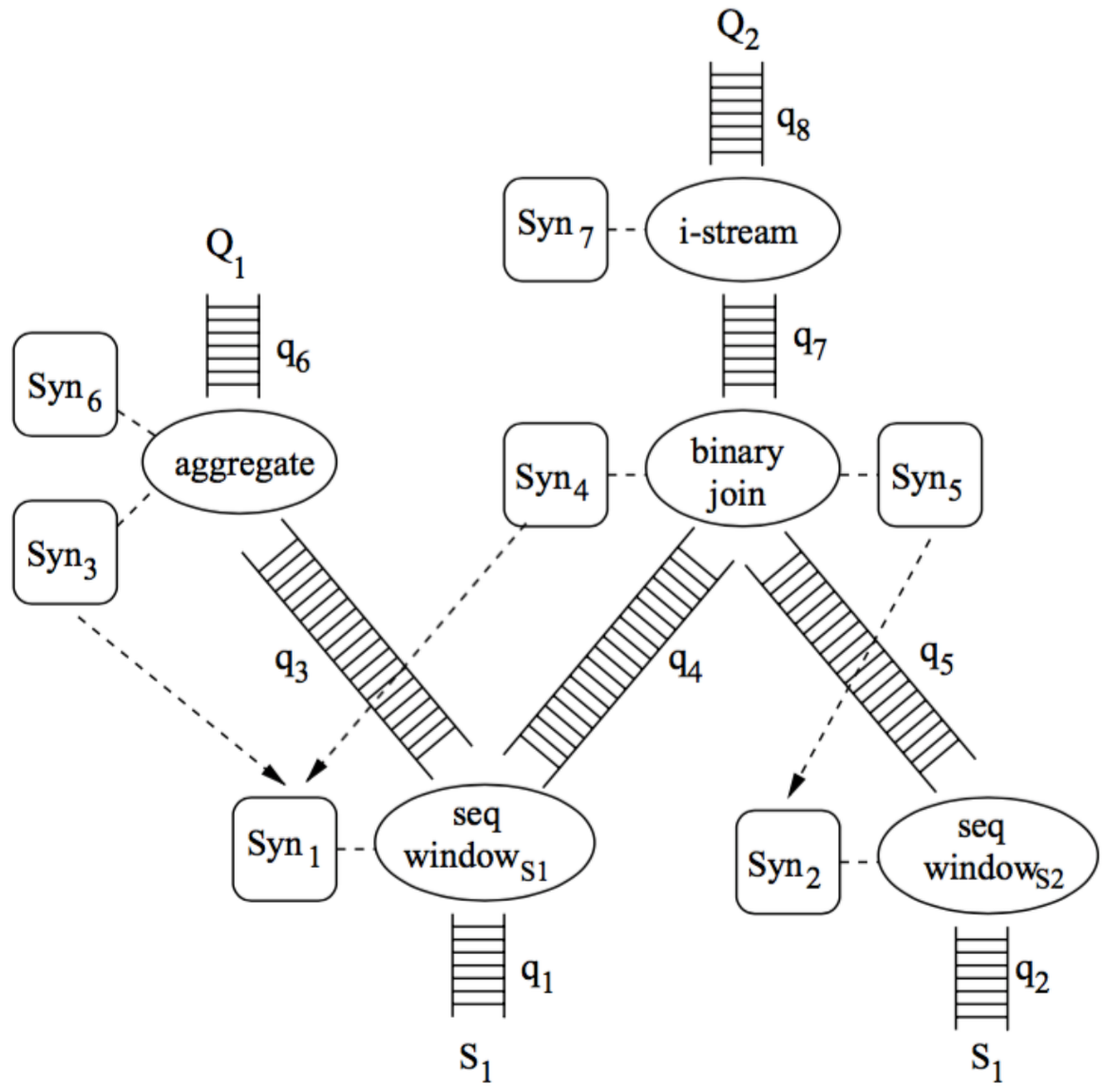
- **Sliding window operator seq-window$_{S2}$**
  - Reads stream S_2's tuples
  - Processes them to know the ones that arrived in the last 10 minutes
  - Outputs in q_5, the insertions and deletions in the window
- **Synopsis Syn$_2$ stores the items that arrived at S2 in the last 10 minutes**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

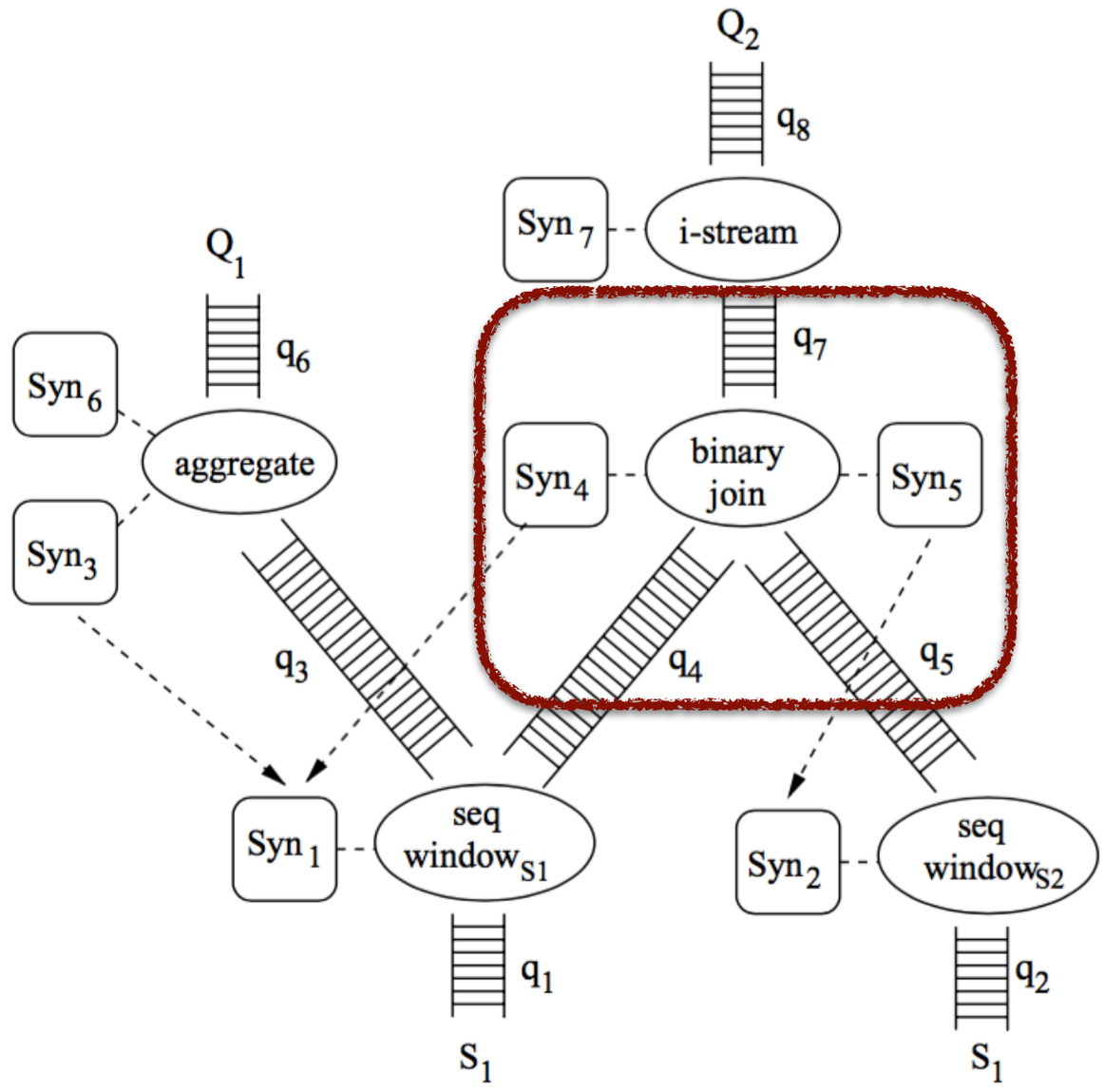
- **operator binary-join**
  - Incrementally computes the join of two relations
    - Uses insertions and deletions in q_4 and q_5, to produce insertions and deletions in q_7




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

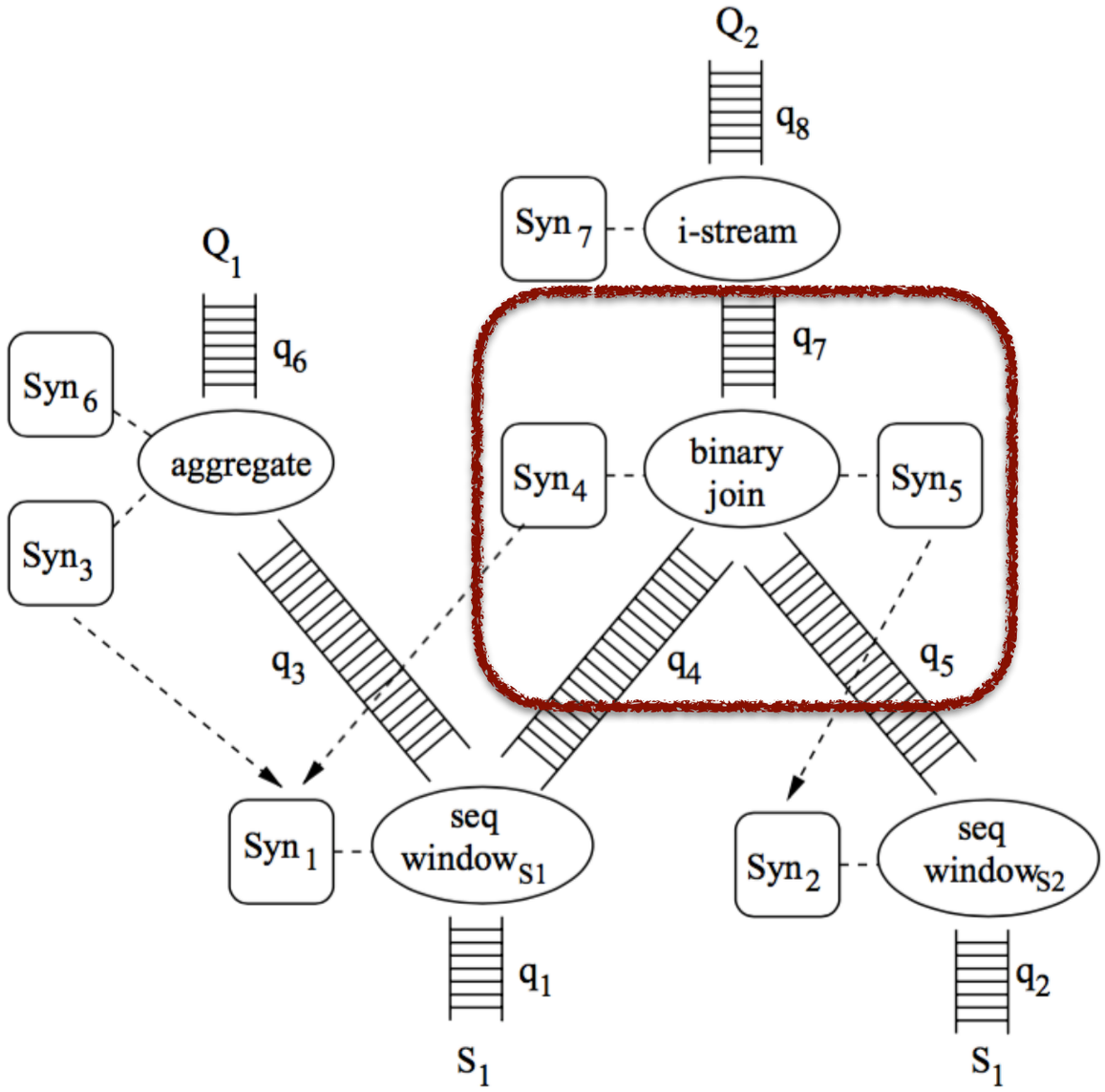
- **operator binary-join**
  - Incrementally computes the join of two relations
    - Uses insertions and deletions in q_4 and q_5, to produce insertions and deletions in q_7




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

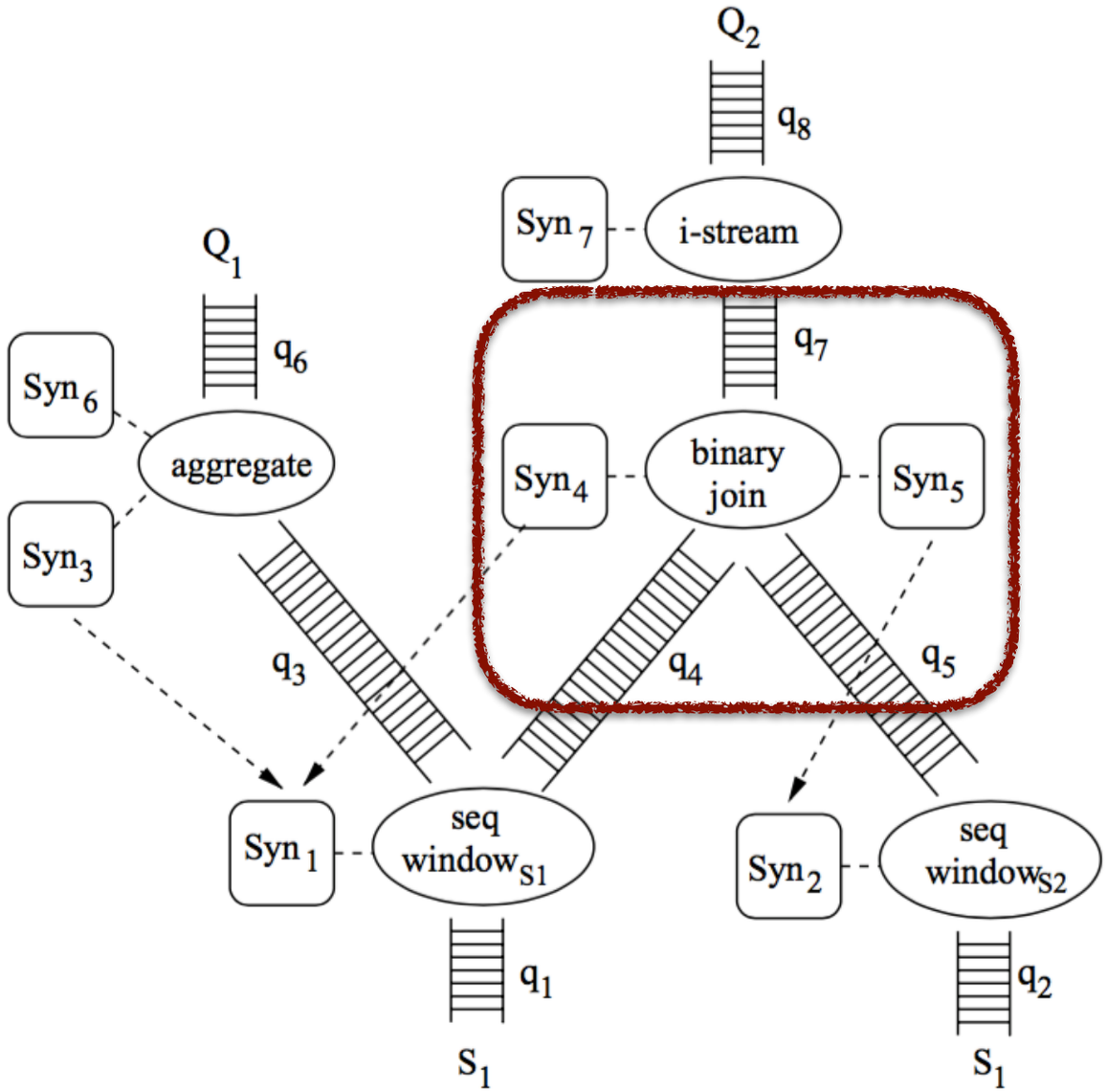
- **operator binary-join**
  - Incrementally computes the join of two relations
    - Uses insertions and deletions in q_4 and q_5, to produce insertions and deletions in q_7
- **Synopsis Syn_4 (resp. Syn_5) is shared with Syn_1 (resp Syn_2)**
  - An inserted tuple in q_4 is joined with the tuples in Syn_5 to compute an insertion in q_7
  - A deleted tuple in q_4 is joined with the tuples in Syn_5 to compute a deletion in q_7
  - The same for q_5 and $Sync_4$ to produce tuples in q_7




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

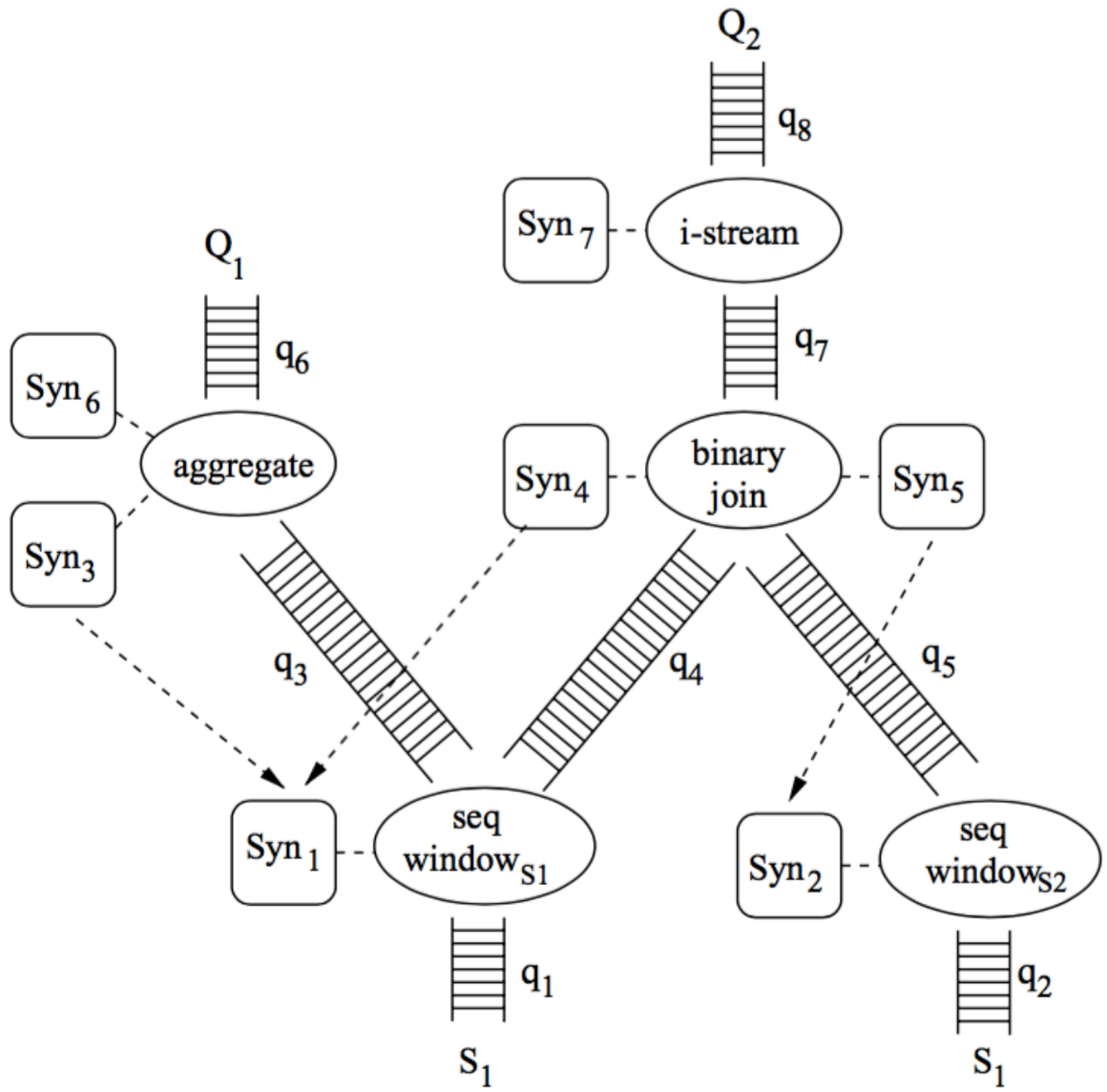
- **operator binary-join**
  - Incrementally computes the join of two relations
    - Uses insertions and deletions in q_4 and q_5, to produce insertions and deletions in q_7
- **Synopsis Syn_4 (resp. Syn_5) is shared with Syn_1 (resp Syn_2)**
  - An inserted tuple in q_4 is joined with the tuples in Syn_5 to compute an insertion in q_7
  - A deleted tuple in q_4 is joined with the tuples in Syn_5 to compute a deletion in q_7
  - The same for q_5 and $Sync_4$ to produce tuples in q_7
- **When joining with a table, the queue *reports* changes, and the synopsis *is* the table content**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

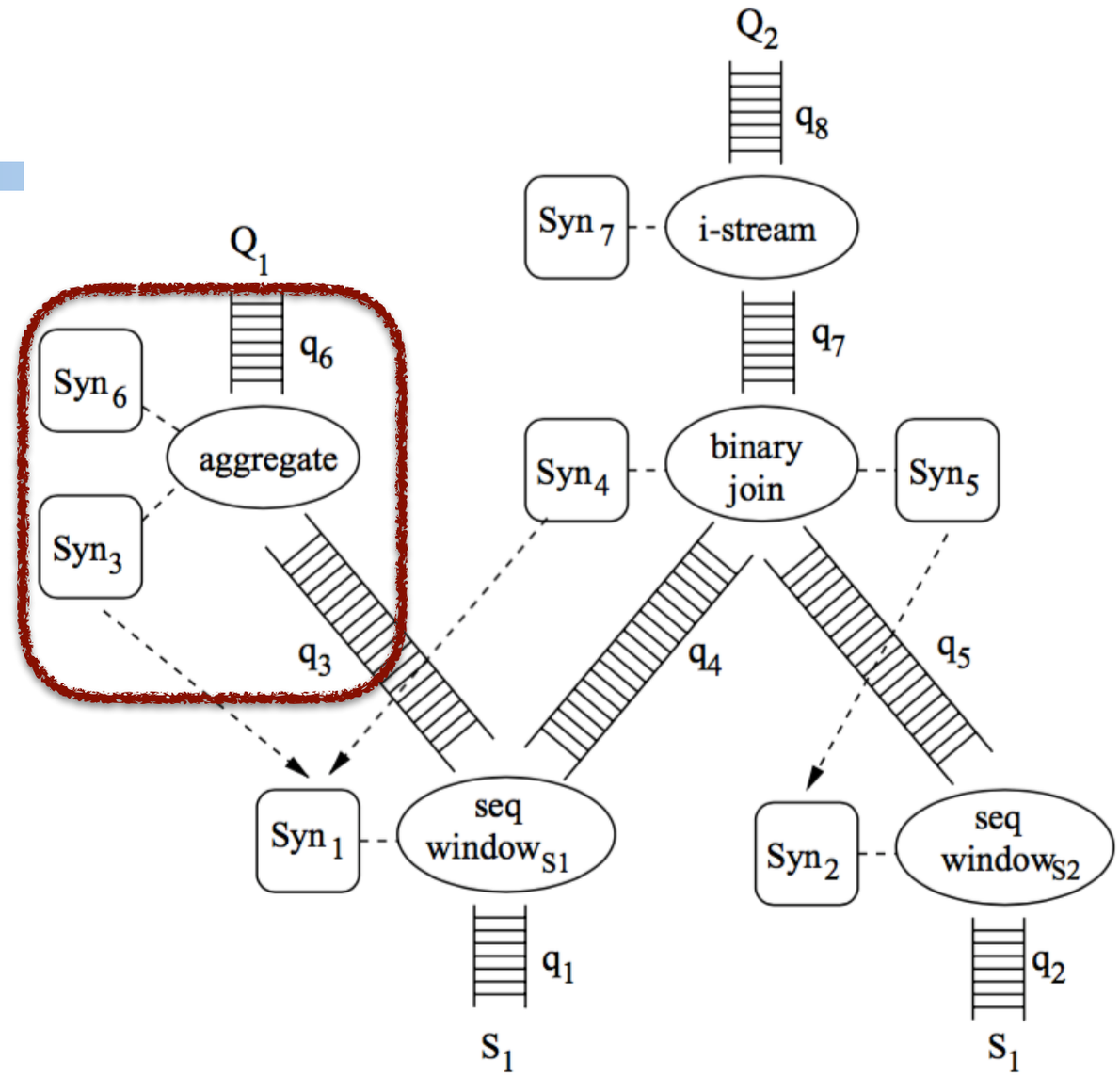
- **operator aggregate**
  - (Incrementally) maintains the max value of A for each distinct B
  - Synopsis Syn_6 stores those max values




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

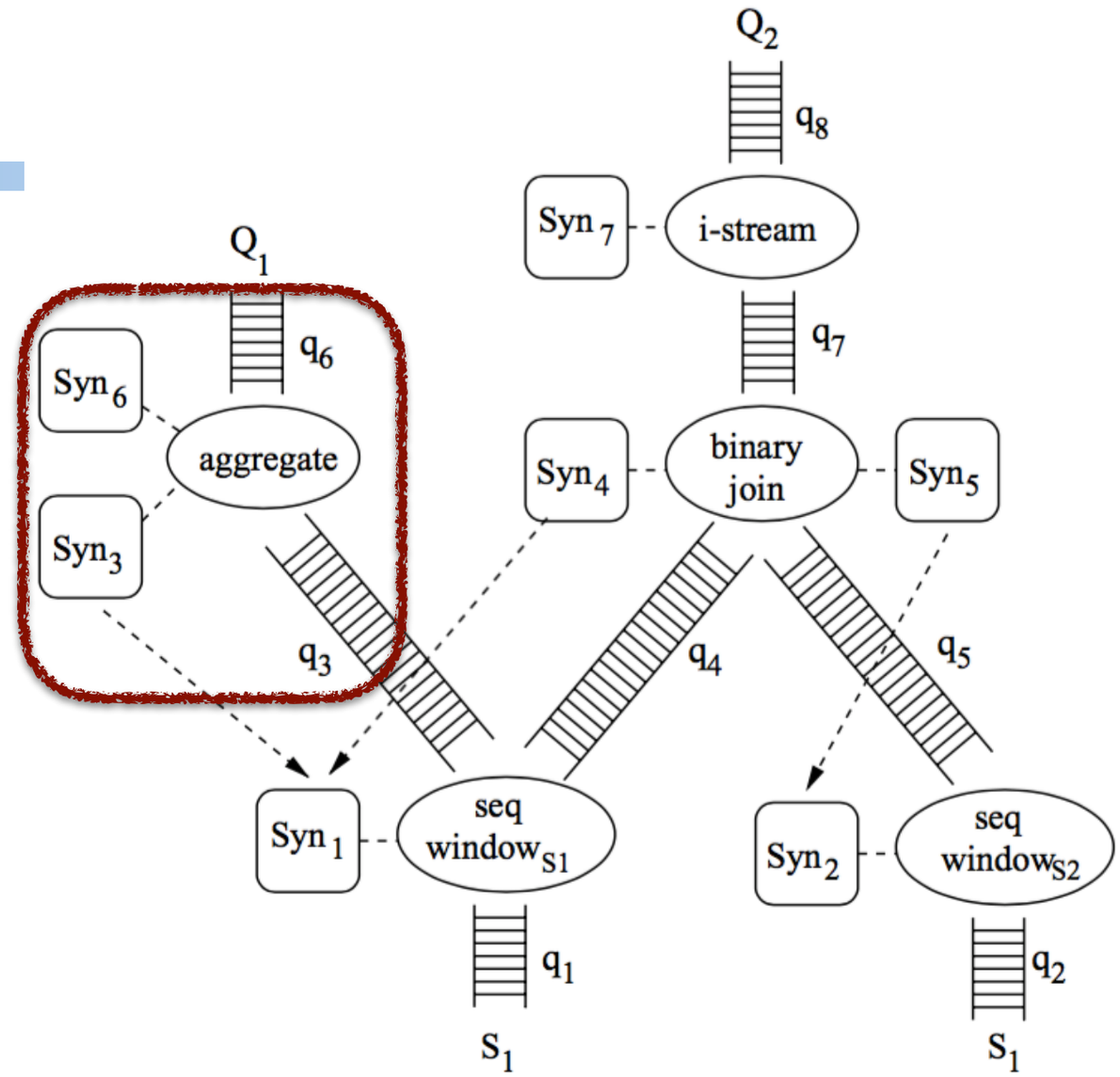
- **operator aggregate**
  - (Incrementally) maintains the max value of A for each distinct B
  - Synopsis Syn_6 stores those max values




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

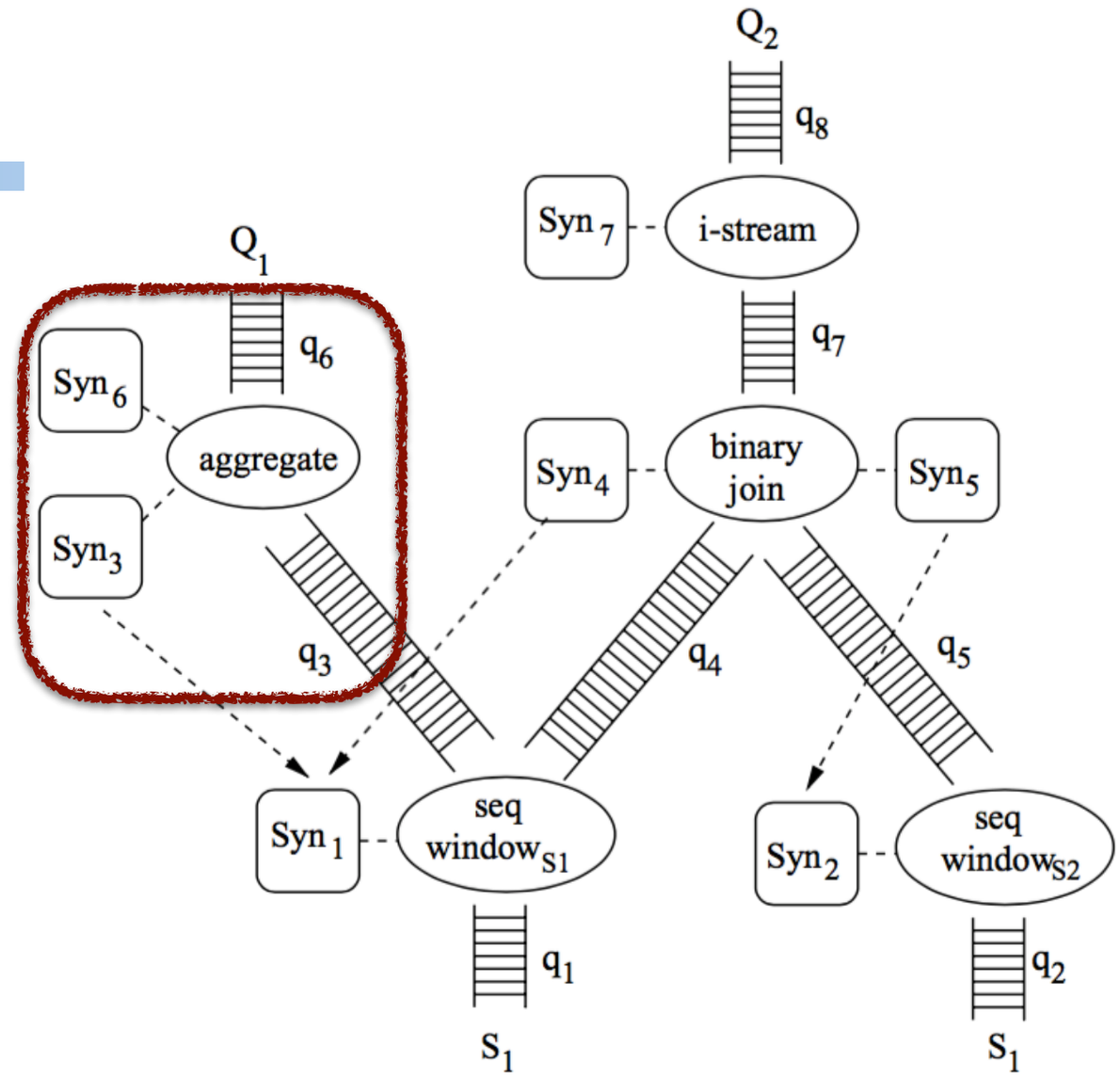
- **operator aggregate**
  - (Incrementally) maintains the max value of A for each distinct B
  - Synopsis Syn_6 stores those max values
- **Any time an insertion (resp. deletion) appears in q_3, the operator must look at Syn_6, and see whether that changes a max value**
  - When a max is deleted, it must consult the values in the window; that is in Syn_3
    - Some of this info may be shared with Syn_1
  - If a max value changes, it outputs in q_6 both a deletion (of the old max) and an insertion (of the new max)




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

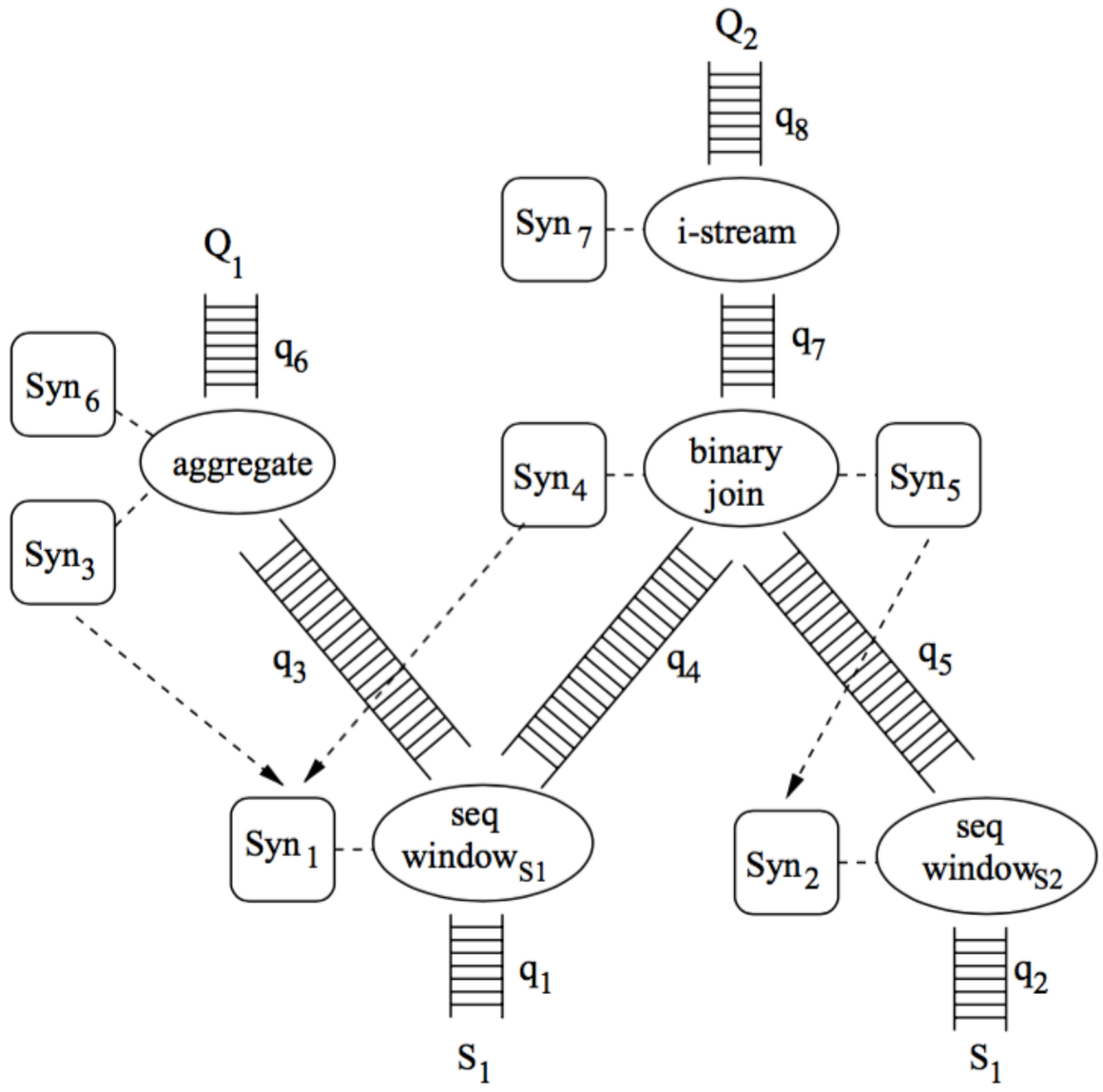
- **operator aggregate**
  - (Incrementally) maintains the max value of A for each distinct B
  - Synopsis Syn_6 stores those max values
- **Any time an insertion (resp. deletion) appears in q_3, the operator must look at Syn_6, and see whether that changes a max value**
  - When a max is deleted, it must consult the values in the window; that is in Syn_3
    - Some of this info may be shared with Syn_1
  - If a max value changes, it outputs in q_6 both a deletion (of the old max) and an insertion (of the new max)
- **Note that this operator is non-monotonic**




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

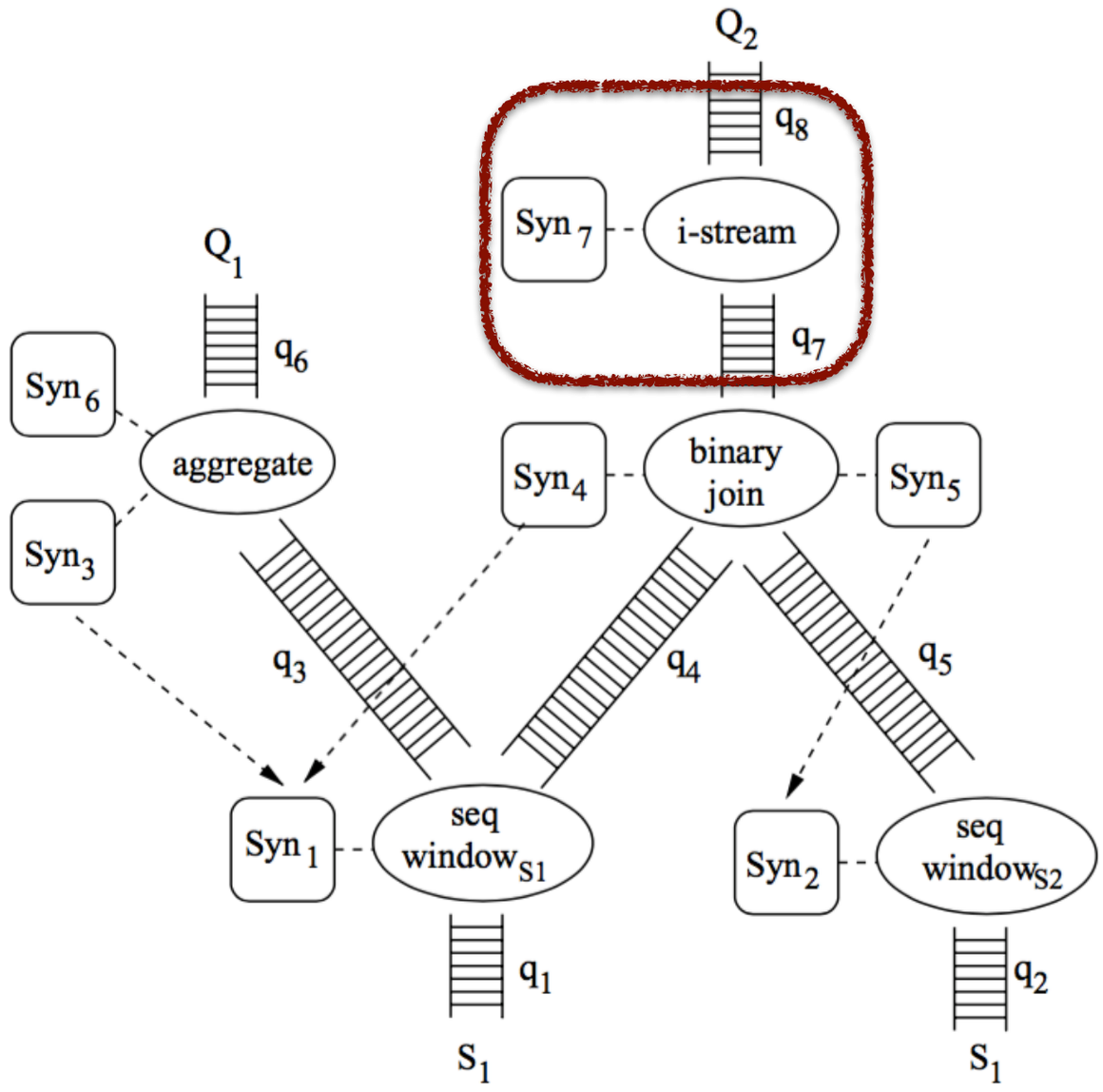
- **operator i-stream**
  - In a naive implementation, whenever an insertion appears in q_7, it outputs the inserted value to q_8
  - This seem to be exactly the meaning of `Istream`




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

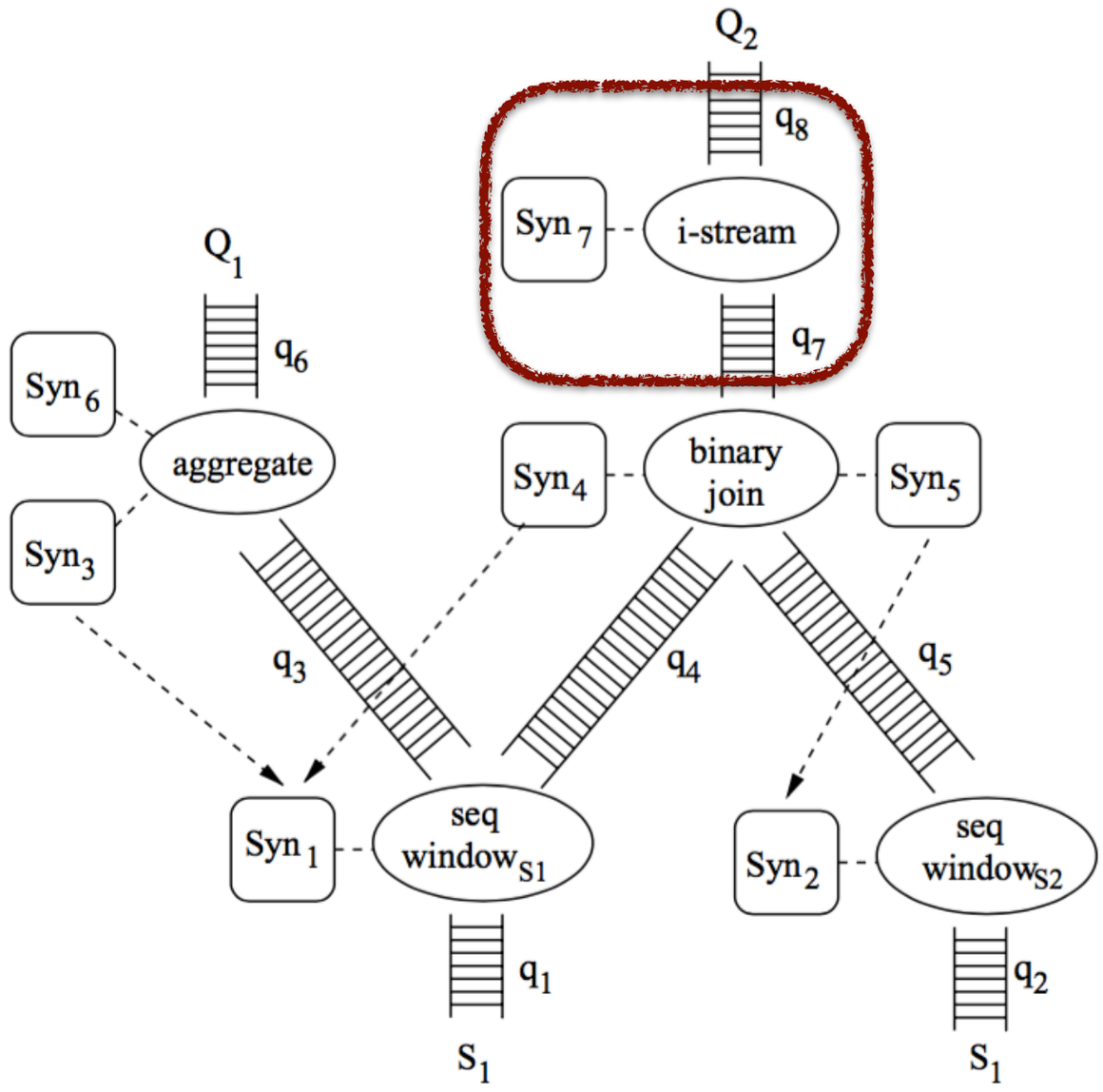
- **operator i-stream**
  - In a naive implementation, whenever an insertion appears in q_7, it outputs the inserted value to q_8
  - This seem to be exactly the meaning of `Istream`




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

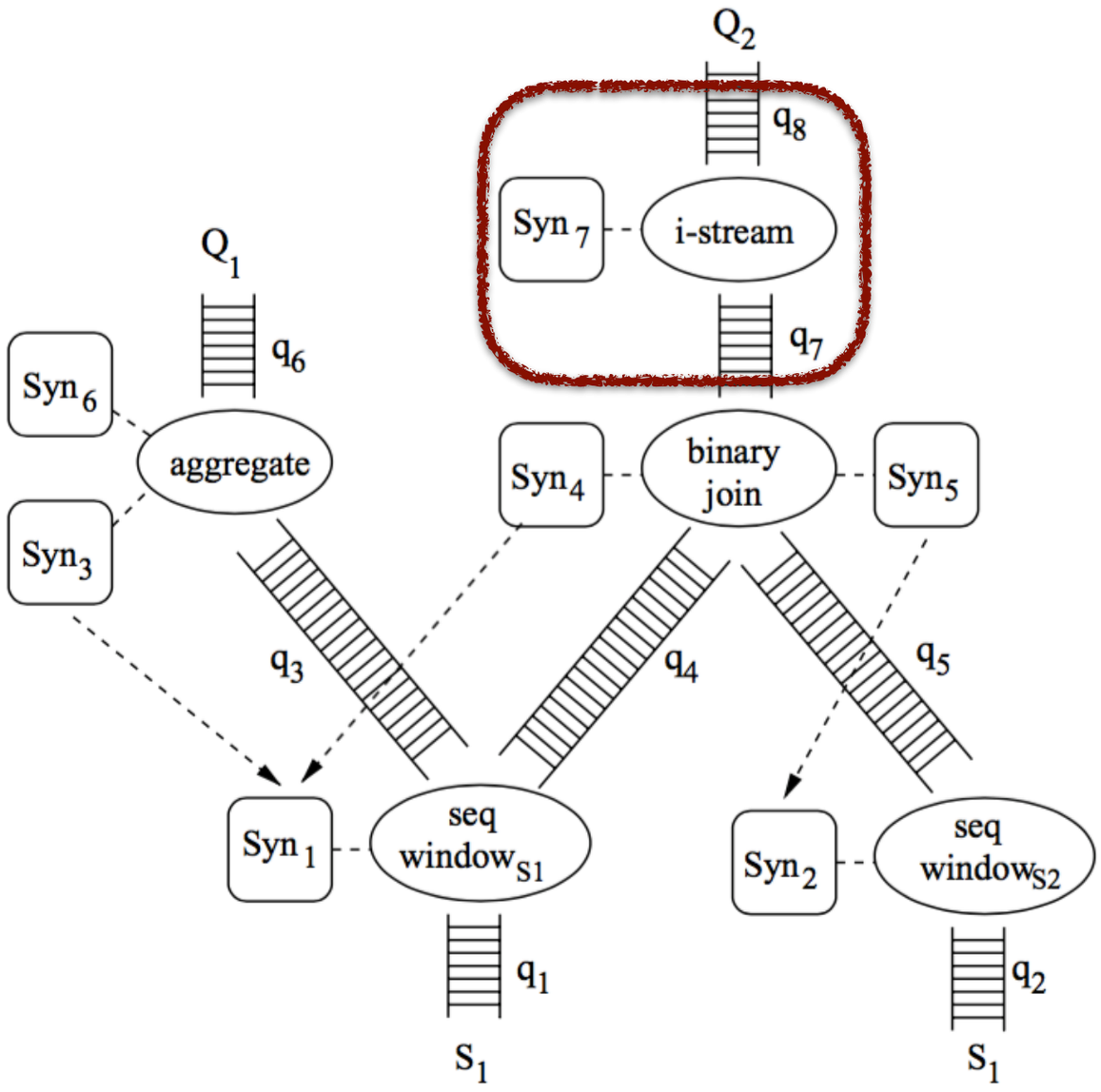
- **operator i-stream**
  - In a naive implementation, whenever an insertion appears in q_7, it outputs the inserted value to q_8
  - This seem to be exactly the meaning of `Istream`
- **But care must be taken!**
  - What if there is both an insertion and a deletion of a same tuple at the same time?




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

Example plan

- **operator i-stream**
  - In a naive implementation, whenever an insertion appears in q_7, it outputs the inserted value to q_8
  - This seem to be exactly the meaning of `Istream`
- **But care must be taken!**
  - What if there is both an insertion and a deletion of a same tuple at the same time?
- **Syn_7 buffers the insertion and deletion at a given time *t***
  - The operator outputs insertions without deletions at *t* to q_8




```
Q1:  select B, max(A)
     from S1 [Rows 50000]
     group by B;

Q2:  select Istream(*)
     from  S1 [Rows 40000]
           S2 [Range 600s]
     where S1.A = S2.A;
```

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**
  - When an input tuple at time *t* causes a tuple based window to slide, the generated insertion and deletion have timestamp *t*

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**
  - When an input tuple at time *t* causes a tuple based window to slide, the generated insertion and deletion have timestamp *t*
  - When **two tuples are joined, the timestamp is the higher of the two**

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**
  - When an input tuple at time *t* causes a tuple based window to slide, the generated insertion and deletion have timestamp *t*
  - When **two tuples are joined, the timestamp is the higher of the two**
  - When a new aggregation result is generated due to an insertion or deletion at *t*, the generated insertion and deletion has timestamp *t*

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**
  - When an input tuple at time *t* causes a tuple based window to slide, the generated insertion and deletion have timestamp *t*
  - When **two tuples are joined, the timestamp is the higher of the two**
  - When a new aggregation result is generated due to an insertion or deletion at *t*, the generated insertion and deletion has timestamp *t*
  - A tuple arriving at *t* in a stream with a time based window with size *T*, generates an insertion at *t*, and a deletion at *t+T+1*

On generated timestamps

- **Special care must be taken with the timestamps that are attached to generated tuples in queues**
  - When an input tuple at time *t* causes a tuple based window to slide, the generated insertion and deletion have timestamp *t*
  - When **two tuples are joined, the timestamp is the higher of the two**
  - When a new aggregation result is generated due to an insertion or deletion at *t*, the generated insertion and deletion has timestamp *t*
  - A tuple arriving at *t* in a stream with a time based window with size *T*, generates an insertion at *t*, and a deletion at *t+T+1*
    - The *+1* depends on the granularity of the system

System operators

- **Real systems have some more operators, to deal with several low-level details. E.g.**
  - To asynchronously receive the tuples from an input stream, and transforming them to the internal representation
  - To manage load balancing
  - To merge plans from views
  - To send stream results to remote clients
  - …

STREAM operators

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 [43]
`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 specified 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

Query plan generation

- **Apply heuristics to generate a first plan**
  - Execute selections before joins (as in DBMSs)
  - Use indexed nested loop joins for joining with synopses
  - Apple filters before ranges
  - Share synopses and operators whenever possible

Query plan generation

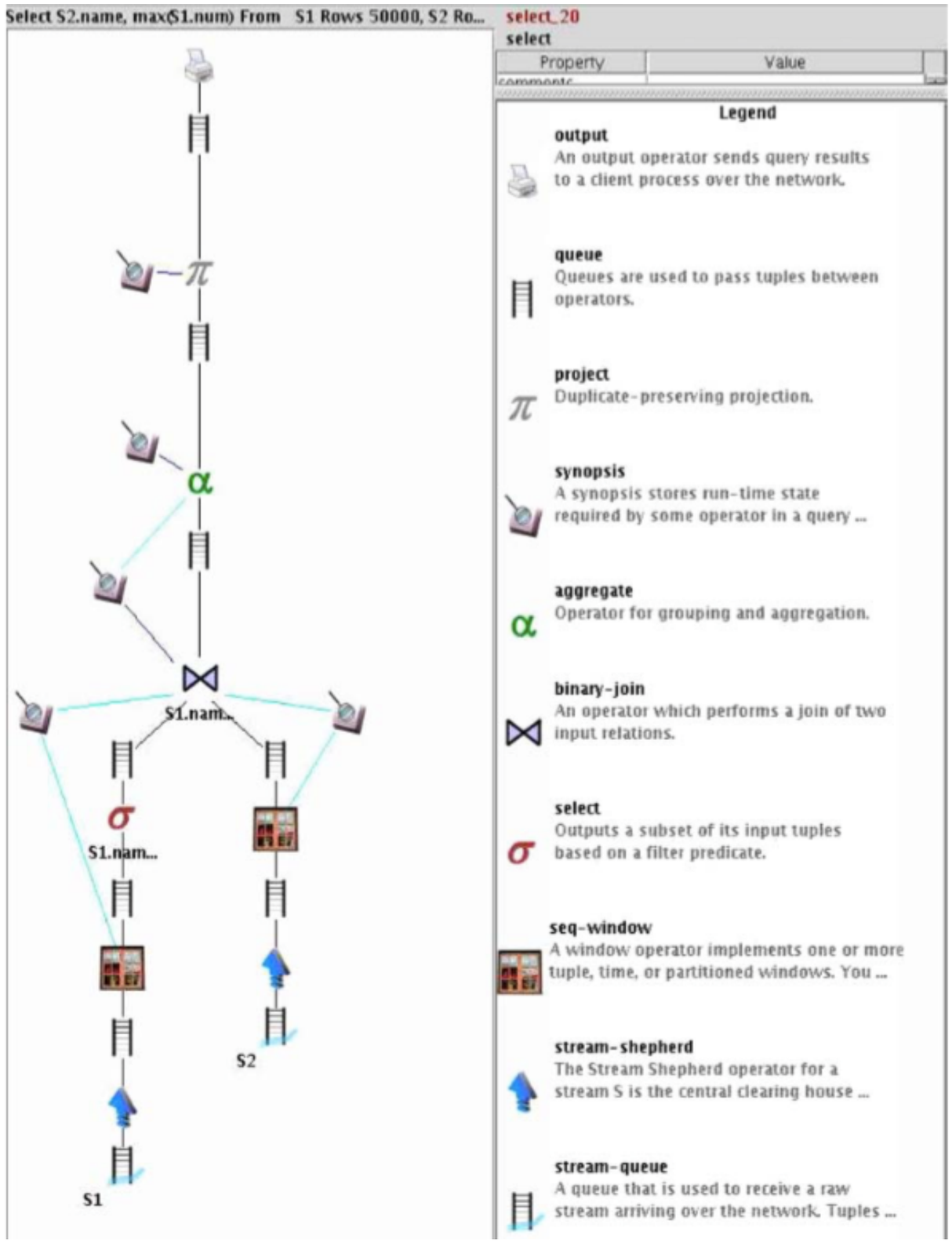
- **Apply heuristics to generate a first plan**
  - Execute selections before joins (as in DBMSs)
  - Use indexed nested loop joins for joining with synopses
  - Apple filters before ranges
  - Share synopses and operators whenever possible
- **Upon execution, statistical data on what has been executed can be used to rearrange initial query plans**
  - Requires a *monitor* for monitoring the plans that are being executed
  - It can use optimisation techniques similar to those of DBMSs, based on what has been monitored

Query execution

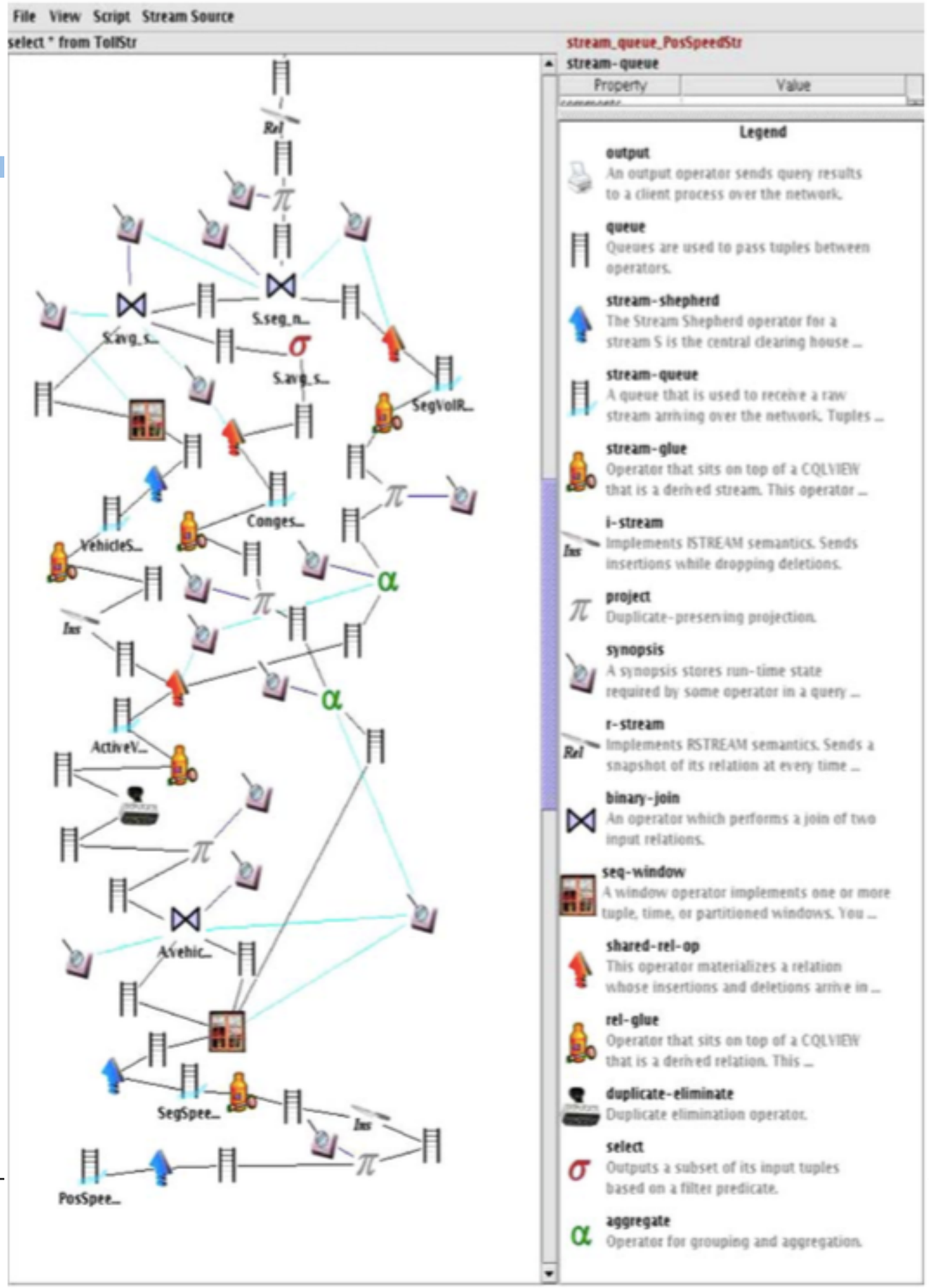
- **The execution of a plan is controlled by a global scheduler**
- **The scheduler is invoked periodically**
  - It selects an operator to execute
  - Calls a specific procedure for the operator, passing it the required parameters (with data from queues and synopses)
- **The scheduler can be a simple round-robin over operators, but may include parallel execution, and more sophisticated scheduling algorithms**
  - This is outside the scope of this course

An example plan in STREAM

```
select S2.name, max(S1.n)
from  S1 [Rows 50000]
      S2 [Rows 50000]
where S1.name <= 'i' and
      S1.n = S2.n
group by S2.name;
```



Linear Road plan



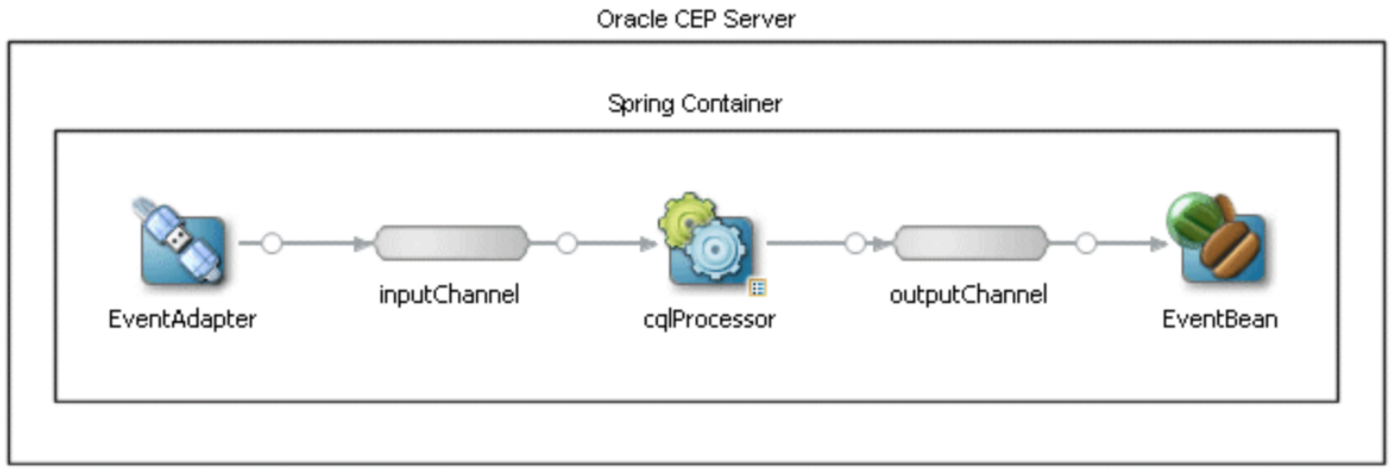
CQL in Oracle CEP

Oracle Event Processor

- **Java Server for event driven applications**
  - A lightweight Java application container
  - Includes a service engine that allows one to write CQL queries
- **Provides tools for developing applications**
  - Oracle CEP IDE for Eclipse
  - Oracle CEP Visualizer

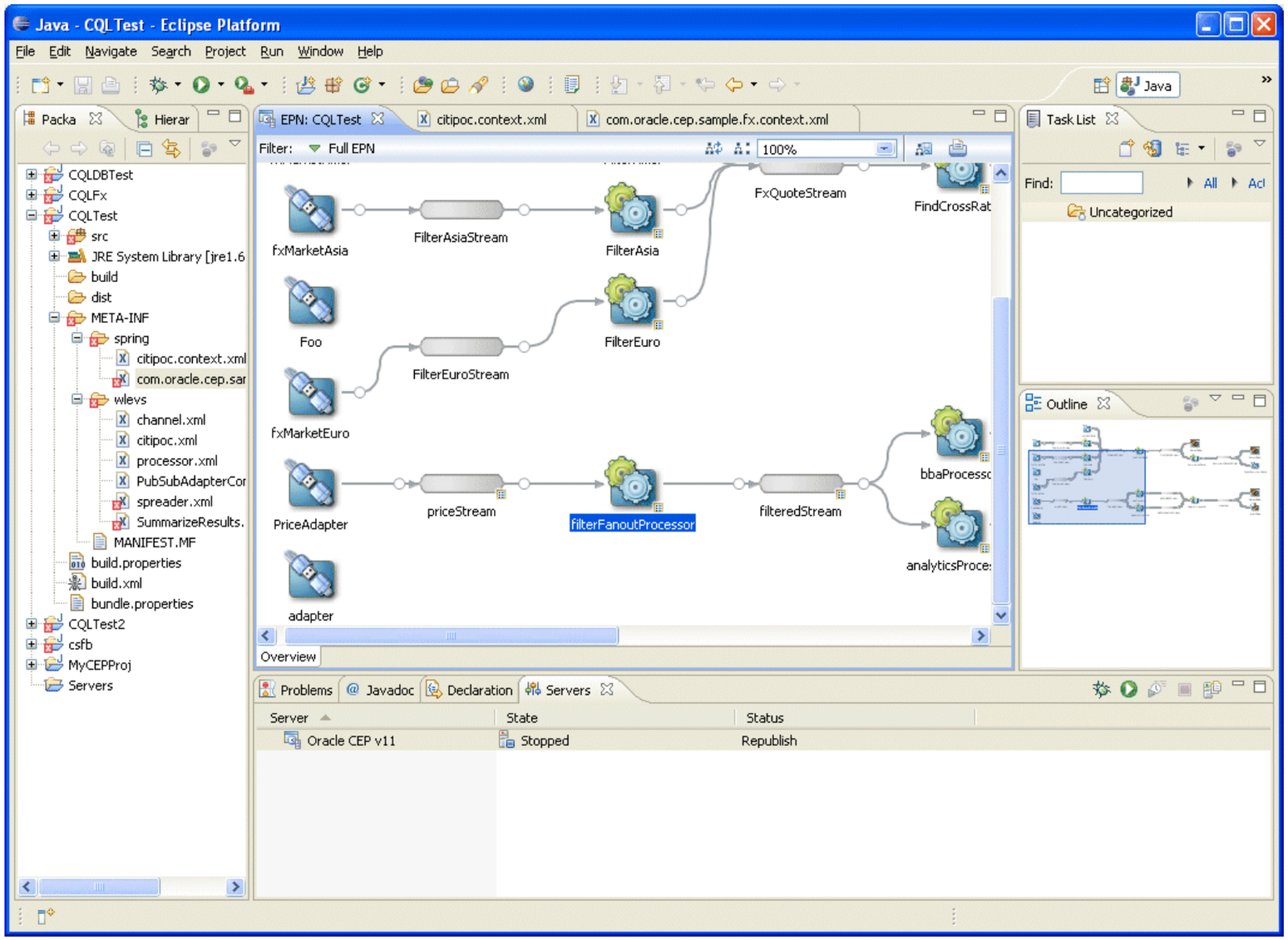
Using Oracle CEP

- **With the IDE for Eclipse and the Visualizer, one can:**
  - Create an Event Processing Network
  - Associate CQL queries to Oracle CQL processors
  - Package applications and deploy them to Oracle server
- **Event adapters support several datasources: JMS, http publish-subscribe, spring beans, relational tables, files, etc**



Event Processing Network (EPN)

- **One can connect the output of an event processor to the input of another event processor**
  - This way we form an EPN
  - Each processor contains a CQL query (possibly with views)



Oracle’s support of CQL

Oracle's support of CQL

- **Pretty much everything that we covered in the course!**
  - With some extra features (details, functions etc)
  - And some (small) differences in the syntax

Putting it to work

- **A lot of “bureaucracy” involved :-(**

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels
  - Configuring CQL server, and Event Processing Language processors

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels
  - Configuring CQL server, and Event Processing Language processors
  - Configuring caching

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels
  - Configuring CQL server, and Event Processing Language processors
  - Configuring caching
  - …

Putting it to work

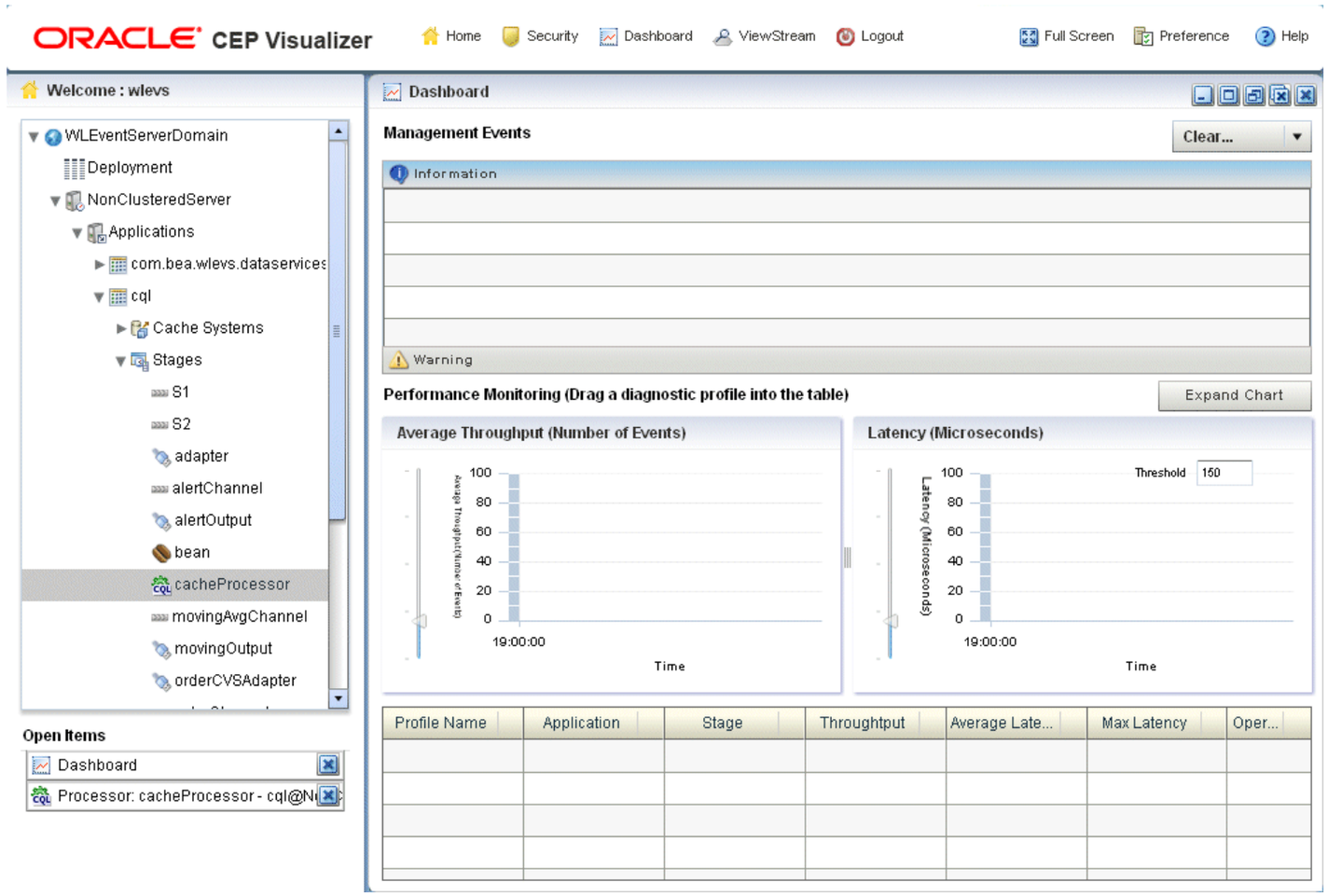
- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels
  - Configuring CQL server, and Event Processing Language processors
  - Configuring caching
  - …
  - Assembling and deploying applications

Putting it to work

- **A lot of “bureaucracy” involved :-(**
  - Configuring input adapters
  - Configuring beans
  - Configuring channels
  - Configuring CQL server, and Event Processing Language processors
  - Configuring caching
  - …
  - Assembling and deploying applications
- **All of this can be done on Eclipse, which helps…**

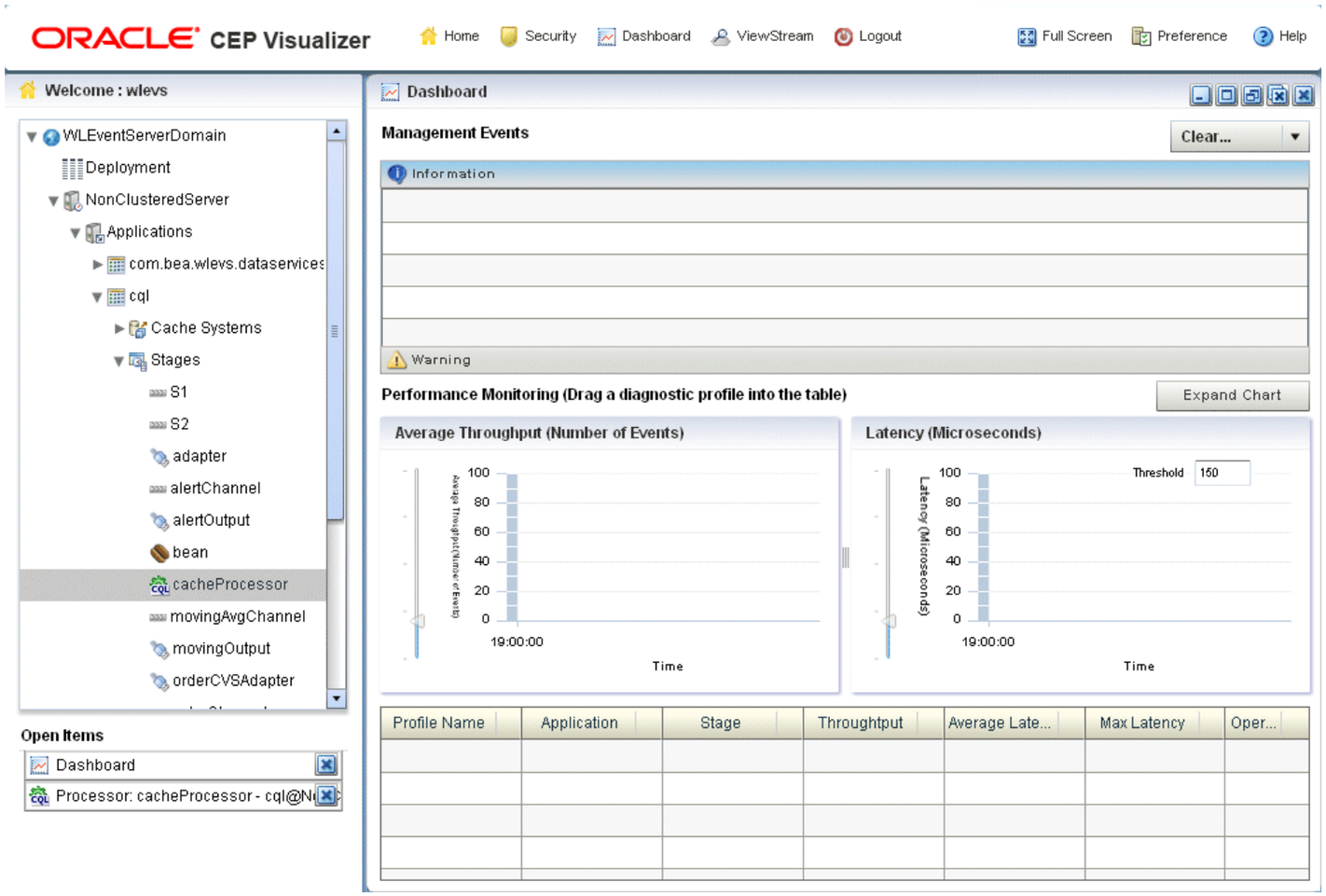
Event Processor Visualizer

- **A browser based tool where one can develop, configure and view aspects of event driven applications**



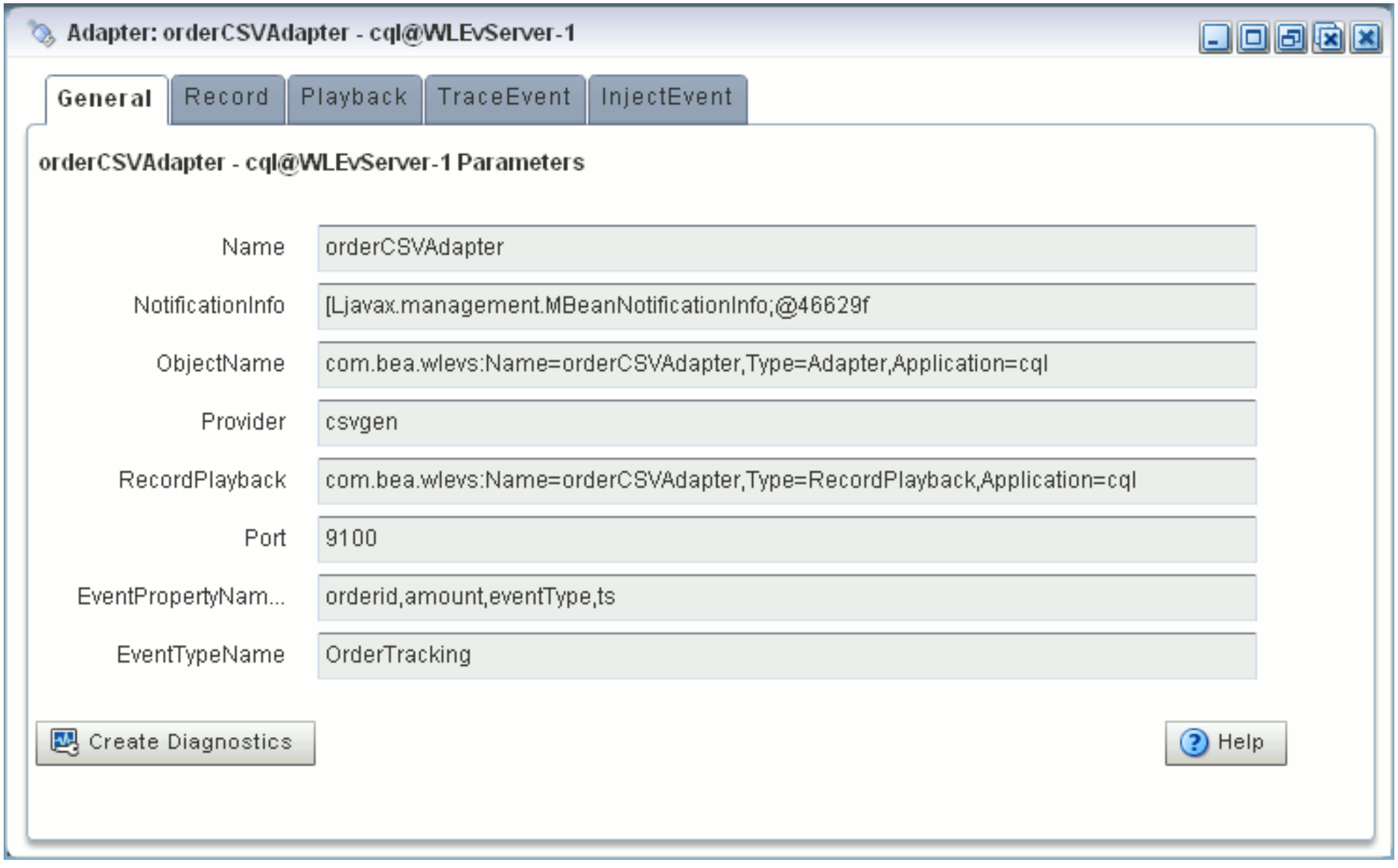
Event Processor Visualizer

- **A browser based tool where one can develop, configure and view aspects of event driven applications**
- **More specifically, one can:**
  - Define EPNs
  - Register CQL queries
    - Including a CQL query wizard
      - (for non-CQL-experts)
    - A query plan viewer
  - Configure event adapters
  - View events in dashboards
  - View event statistics



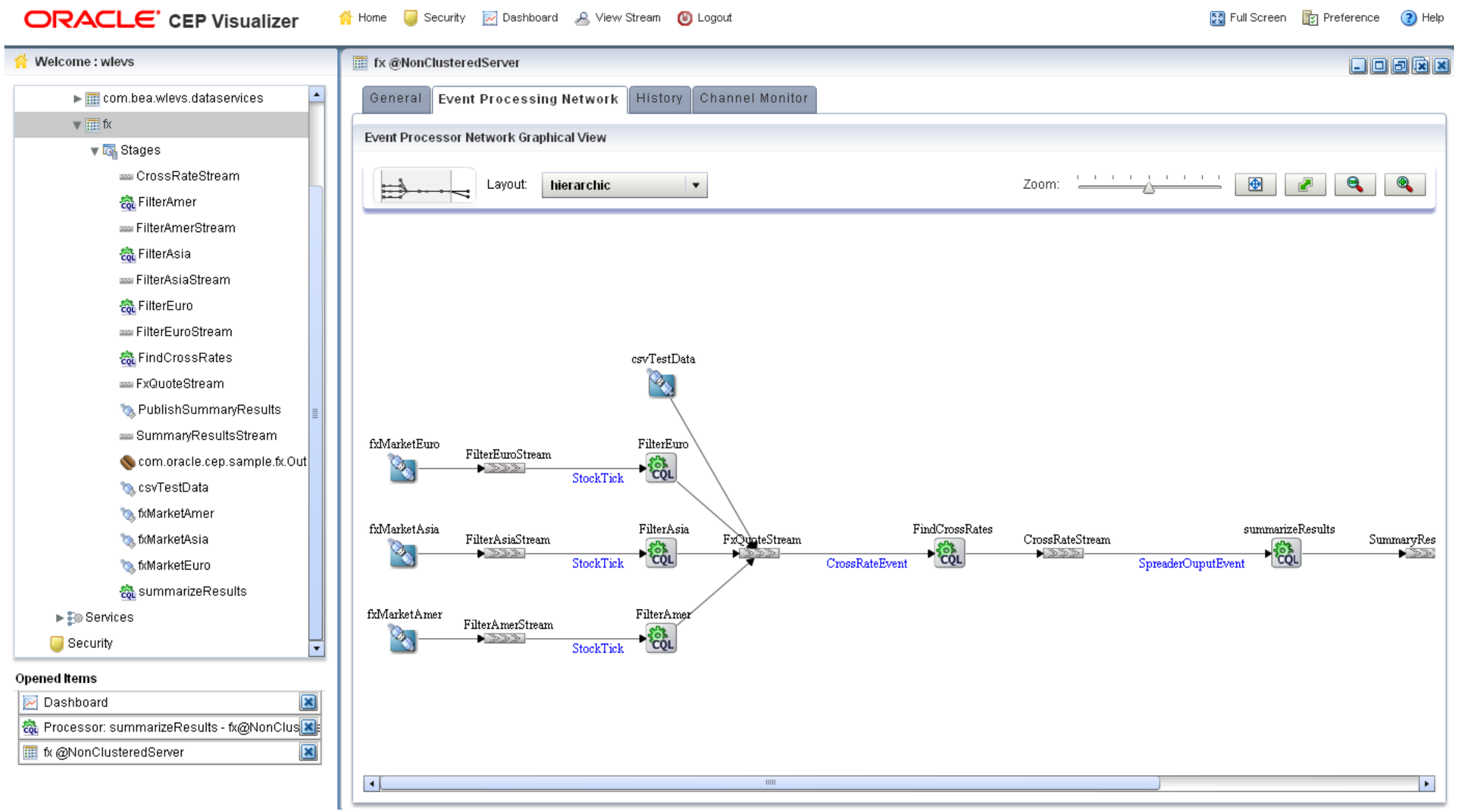
Some Visualizer snapshots

Defining an adapter



Some Visualizer snapshots

Defining an EPN



Some Visualizer snapshots

Defining a CQL query

Processor: summarizeResults - fx@NonClusteredServer

General | Record | Playback | Query Wizard | **CQL Rules** | Query Plan | Trace Event | Inject Event

View ◯ Query ◉ All Rules

Rule ID	Rule	Type	Running
Rule	select crossRate1 || crossRate2 as crossRatePair, count(*) as totalCount, avg(internalPrice) as averageInternalPrice from CrossRateStream group by crossRate1,crossRate2 having count(*) > 0	QUERY	true
SummarizeResultsRule	select crossRate1 || crossRate2 as crossRatePair, count(*) as totalCount, :1 as averageInternalPrice from CrossRateStream group by crossRate1,crossRate2 having :2	QUERY	true

Working Area - for Modify and Delete Operation, select a rule from the table

Query ID: SummarizeResultsRule

Query: select crossRate1 || crossRate2 as crossRatePair, count(*) as totalCount, :1 as averageInternalPrice from CrossRateStream group by crossRate1,crossRate2 having :2

Enable: ◉ true ◯ false

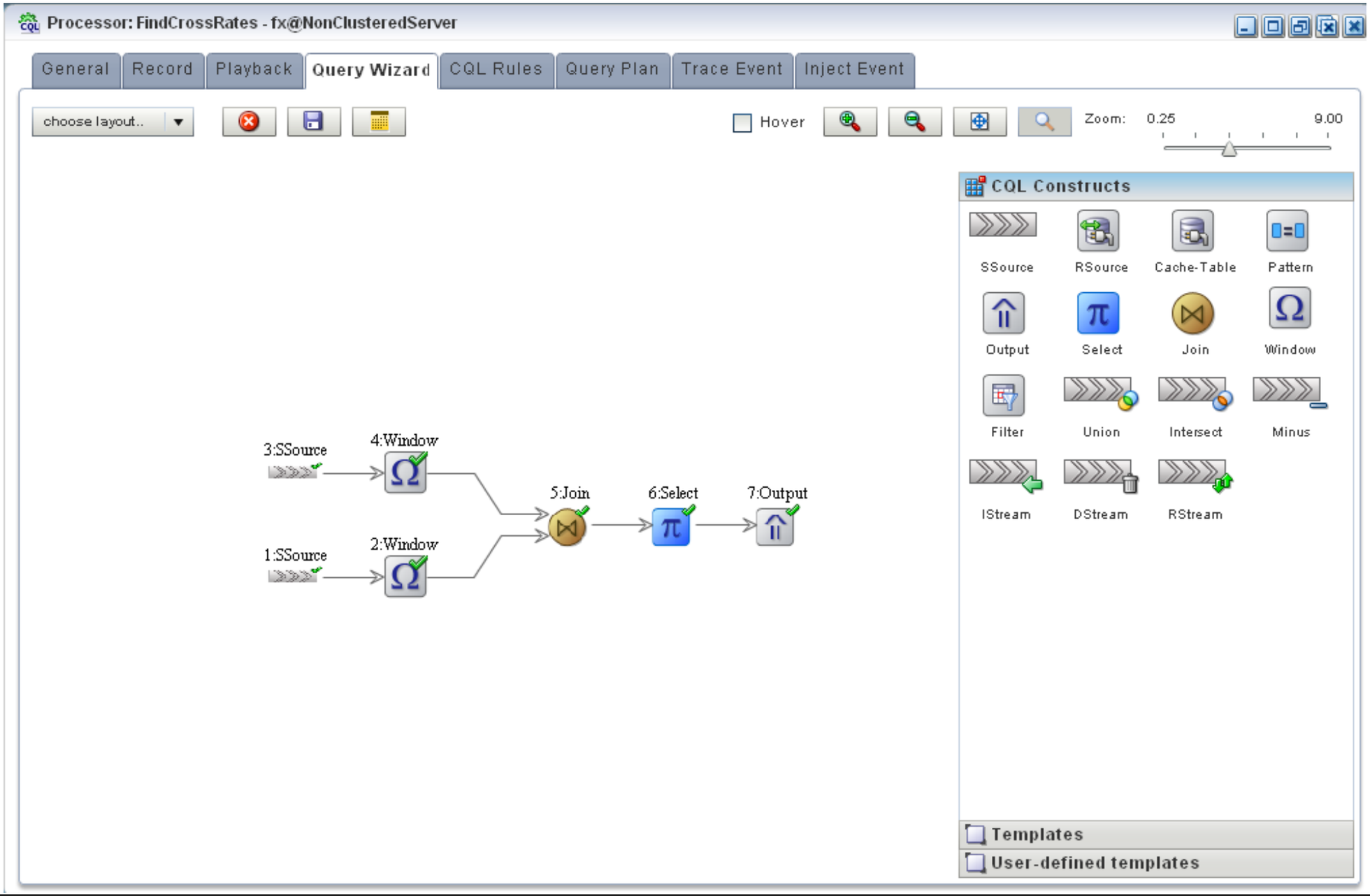
Parameters

Binding Name	1	2
avgcount	avg(internalPrice)	count(*) > 0

Delete Query | Edit Query | Add Parameter | Replace Parameter | Delete Parameter | Stop | Save | Cancel | Help

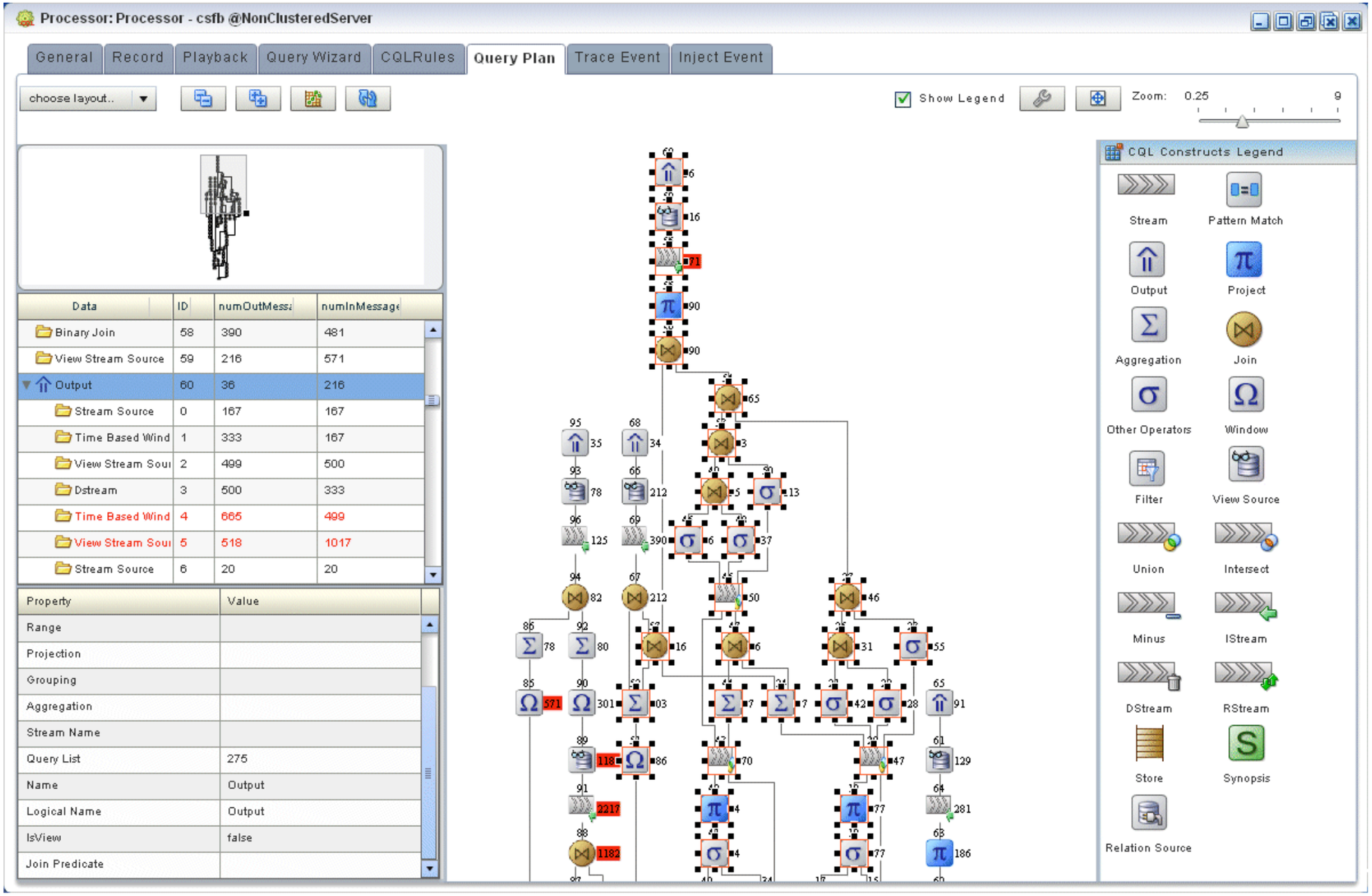
Some Visualizer snapshots

CQL “for dummies” (not for you!)



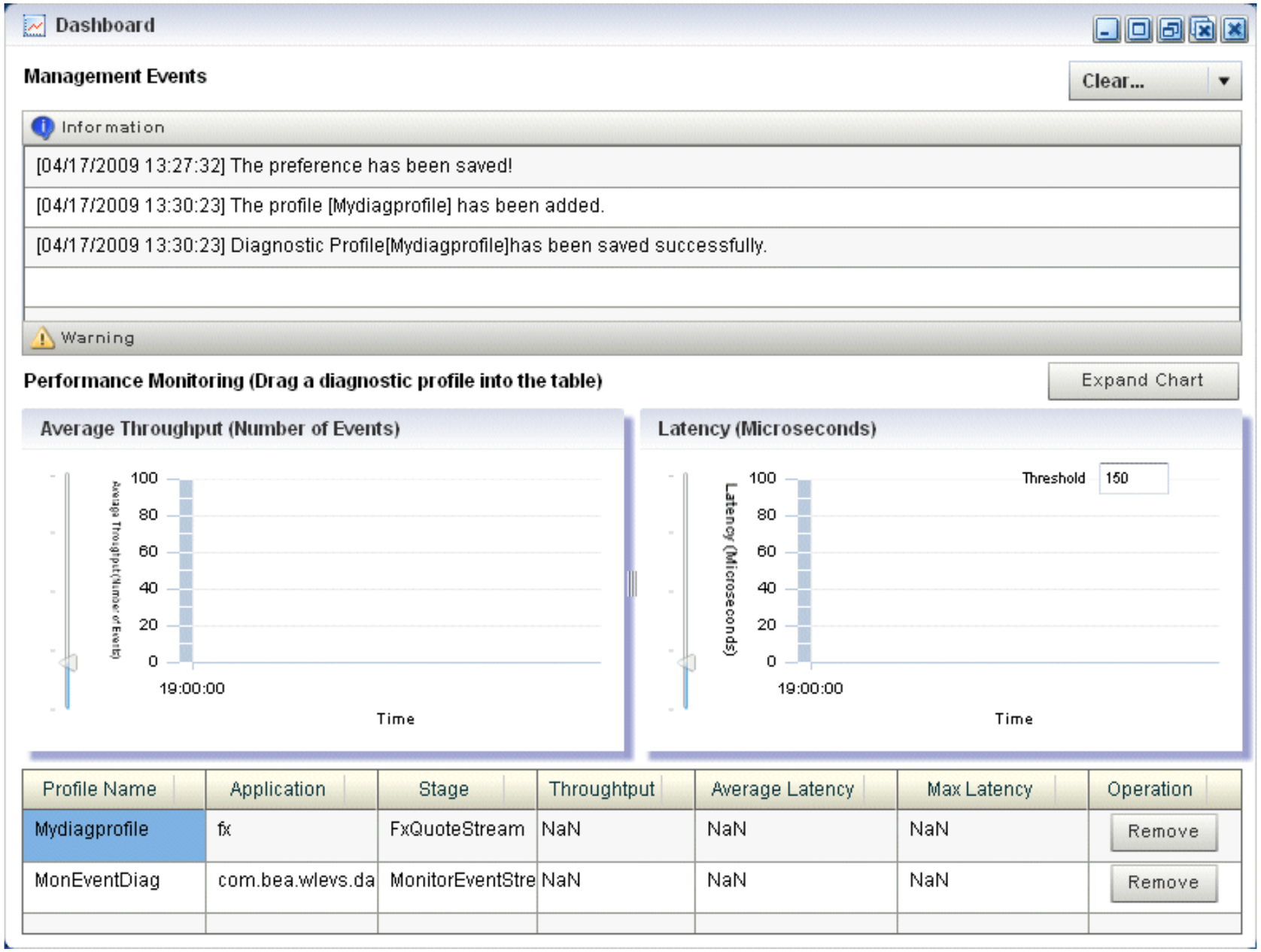
Some Visualizer snapshots

Query plans



Some Visualizer snapshots

Event dashboard and statistics



Readings

- **Oracle CEP CQL Language Reference**
  - https://docs.oracle.com/cd/E16764_01/doc.1111/e12048/toc.htm
- **Oracle CEP IDE Developer's Guide for Eclipse**
  - https://docs.oracle.com/cd/E16764_01/doc.1111/e14301/toc.htm
- **Oracle Event Processing Visualizer User Guide**
  - http://docs.oracle.com/cd/E28280_01/doc.1111/e14302/toc.htm
- **CURRENT DOC**

https://docs.oracle.com/middleware/1213/eventprocessing/index.html

Further Reading and Summary

Q&A

Further Reading

- **Recommend Readings**
  - Arvind Arasu, Shivnath Babu, Jennifer Widom: The CQL continuous query language: semantic foundations and query execution. VLDB J. 15(2): 121-142 (2006)

- **Supplemental readings:**
  - Arvind Arasu et al.: STREAM: The Stanford Data Stream Management System. Stanford InfoLab Tech. Rep. (2004)
  - Arvind Arasu et al.: Linear Road: A Stream Data Management Benchmark. VLDB 2004: 480-491
  - Arvind Arasu, Jennifer Widom: A Denotational Semantics for Continuous Queries over Streams and Relations. SIGMOD Record 33(3): 6-12 (2004)

Further Reading and Summary

Q&A