

# CQL: A Language for Continuous Queries over Streams and Relations

Jennifer Widom  
Stanford University

Joint work with Arvind Arasu & Shivnath Babu

stanfordstreamdatamanager

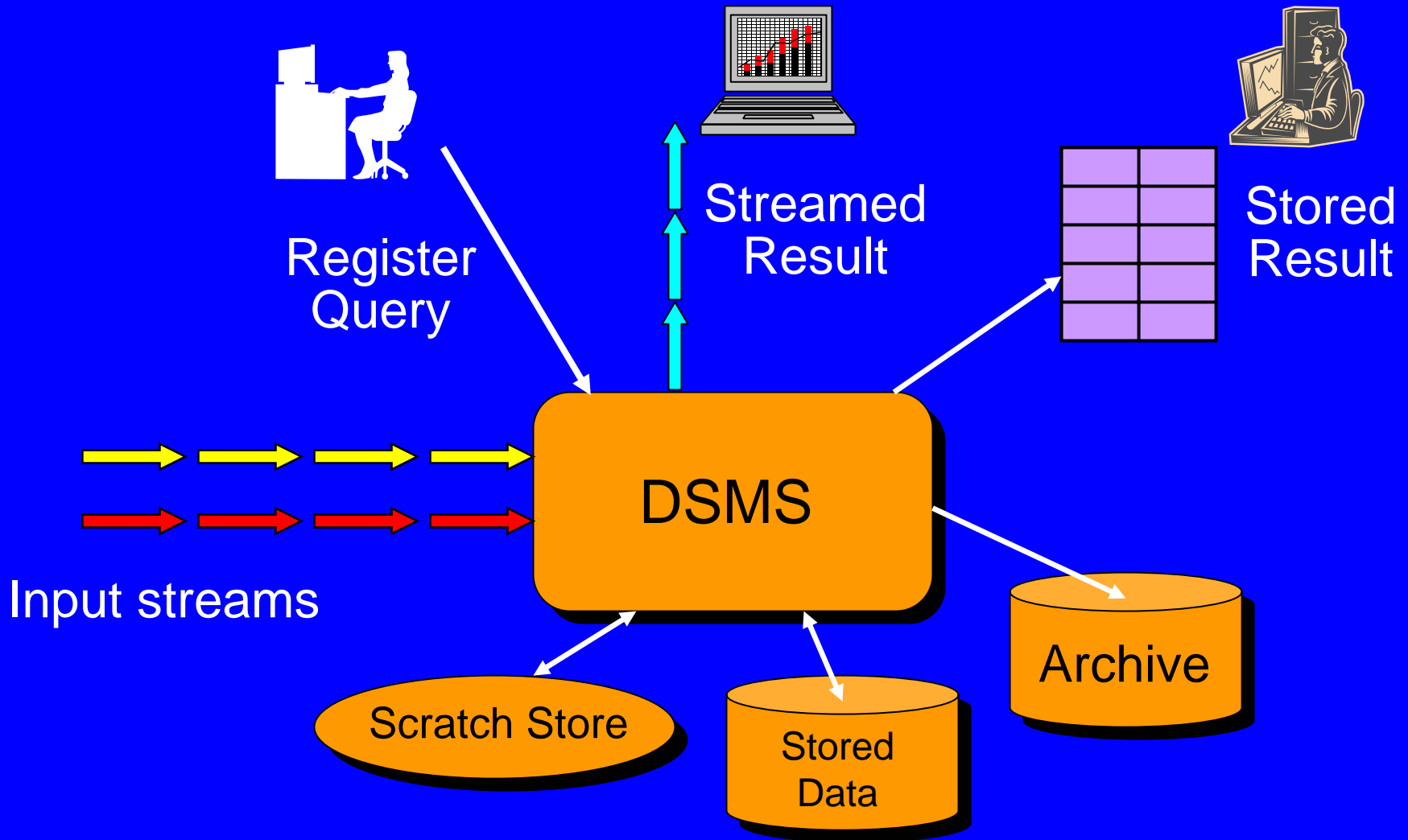
# Data Streams

- Continuous, unbounded, rapid, time-varying streams of data elements
- Occur in a variety of modern applications
  - Network monitoring and traffic engineering
  - Sensor networks, RFID tags
  - Telecom call records
  - Financial applications
  - Web logs and click-streams
  - Manufacturing processes
- **DSMS** = Data Stream Management System

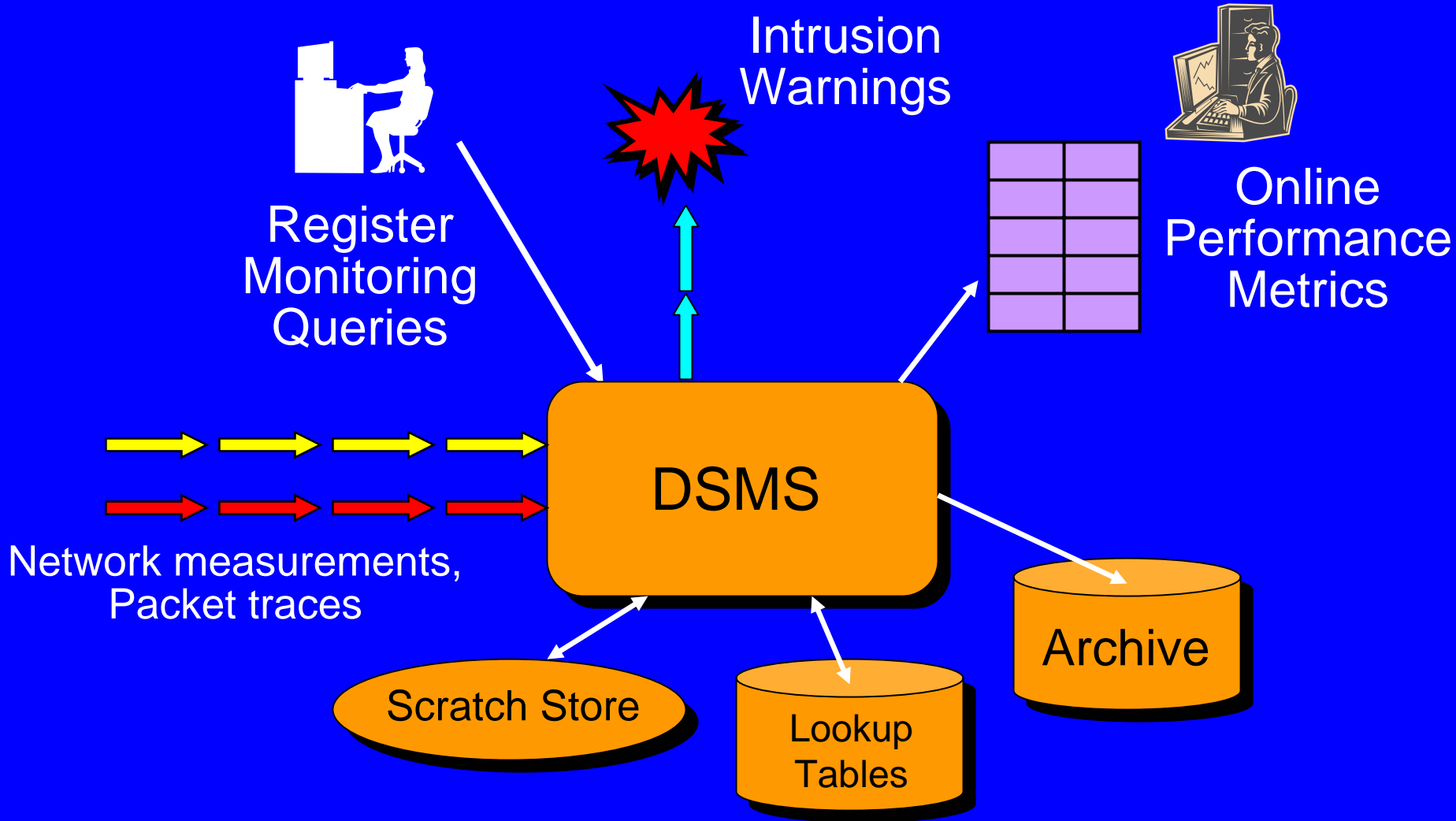
# DBMS versus DSMS

- Persistent data
- One-time queries
- Random access
- Access plan determined by query processor and physical DB design
- Transient data (and persistent data)
- Continuous queries
- Sequential access
- Unpredictable data characteristics and arrival patterns

# The (Simplified) Big Picture



# (Simplified) Network Monitoring



# STREAM

## The Stanford Stream Data Manager

- General-purpose DSMS for streams and stored data
- Relational (unlikely to change)
- Centralized server model (likely to change)
  - Single-threaded and parallel versions
- Declarative language for registering continuous queries

# The STREAM System: Some Implementation Issues

- Designed to cope with:
  - **Stream rates** that may be high, variable, bursty
  - **Continuous query loads** that may be high, volatile
- Primary coping techniques
  - Continuous **self-monitoring** and **reoptimization**
  - Graceful **approximation** as necessary
  - Careful **resource allocation and use**
- These issues do not affect the query language directly

# The STREAM System: Current Status

- Version 1.0 up and running
- Implements most of query language
- Now incorporating the fun stuff
- Currently approx. 25,000 lines of C++ code
- Release plans
  - 1) Web-accessible running server
  - 2) Downloadable software

# The Stream Systems Landscape

- (At least) three general-purpose DSMS prototypes underway
  - **STREAM** (Stanford)
  - **Aurora** (Brown, Brandeis, MIT)
  - **TelegraphCQ** (Berkeley)
- Cooperating to develop **stream system benchmark**
  - One goal: demonstrate that conventional systems are far inferior for data stream applications

# Declarative Language for Continuous Queries

- A distinction between Aurora and the other two systems (STREAM and TelegraphCQ)
  - Aurora users directly manipulate a single large execution plan through a “boxes-and-arrows” interface
  - STREAM compiles declarative queries into individual plans, system may merge plans
  - STREAM also supports direct entry of plans

# Declarative Language (cont'd)

- STREAM's declarative language is based on **SQL** syntax + some aspects of semantics
  - TelegraphCQ's also based on SQL but more loosely
- Modifications to standard SQL semantics for:
  - **Streams plus relations**
  - **Continuous query results**
- Additional constructs for **sliding windows** and **sampling**

# Aside: Query Language Semantics

- Around 1980

People were inventing, understanding, and implementing relational query languages

- Around 1990

People were inventing and implementing trigger (active rule) systems without completely defining or understanding rule behavior

- Around 2000

Ditto for continuous queries

# Aside on Semantics (cont'd)

- The semantics of SQL queries is (relatively) easy to understand
  - Even lots of SQL queries running together
- The semantics of a single trigger is (relatively) easy to understand
  - But lots of triggers together can be complex
- The semantics of even a single continuous query may not be obvious
  - But lots running together is no harder

# A Nonobvious Continuous Query

- Stream of stock quotes: `Stocks(ticker,price)`
- Monitor last 10 minutes of quotes:  
`Select * From Stocks [Range 10 minutes]`
- Is result a relation, a stream, or something else?
- If a relation, what exactly does it contain?
- If a stream, how does query differ from:  
`Select * From Stocks [Range 1 minute]`  
or `Select * From Stocks [ $\infty$ ]`

# Another Nonobvious CQ

- Stream of ordered items, table of item prices
- Prices for five most recent ordered items:  
Select P.price  
From Items [Rows 5] I, PriceTable P  
Where I.itemID = P.itemID
- Is result a stream or a relation?
- What if item price changes?

# Our Original Working Semantics

*“The result of a continuous query at time  $T$  is the result of treating the streams up to  $T$  as relations and evaluating the query using standard relational semantics.”*

- + Relies on existing well-understood semantics
- Asymmetric: streams-to-relations but not relations-to-streams
- Can break down for complex queries (subqueries, aggregation)

# Our Final Semantics and Language

- **Abstract:** interpretation for continuous queries based on certain “black boxes”
- **Concrete:** SQL-based instantiation for our system; includes syntactic shortcuts, defaults, equivalences

# Goals in Language Design

- 1) Support continuous queries over multiple streams and relations
- 2) Exploit relational semantics to the extent possible
- 3) Easy queries should be easy to write
- 4) Simple queries should do what you expect

# Example Query 1

Two streams, contrived for ease of examples:

Orders (orderID, customer, cost)

Fulfillments (orderID, clerk)

Total cost of orders fulfilled over the last day by  
clerk “Sue” for customer “Joe”

```
Select Sum(O.cost)
```

```
From Orders O, Fulfillments F [Range 1 Day]
```

```
Where O.orderID = F.orderID And F.clerk = “Sue”
```

```
And O.customer = “Joe”
```

# Example Query 2

Using a 10% sample of the Fulfillments stream, take the 5 most recent fulfillments for each clerk and return the maximum cost

```
Select F.clerk, Max(O.cost)
From Orders O,
      Fulfillments F [Partition By clerk Rows 5] 10% Sample
Where O.orderID = F.orderID
Group By F.clerk
```

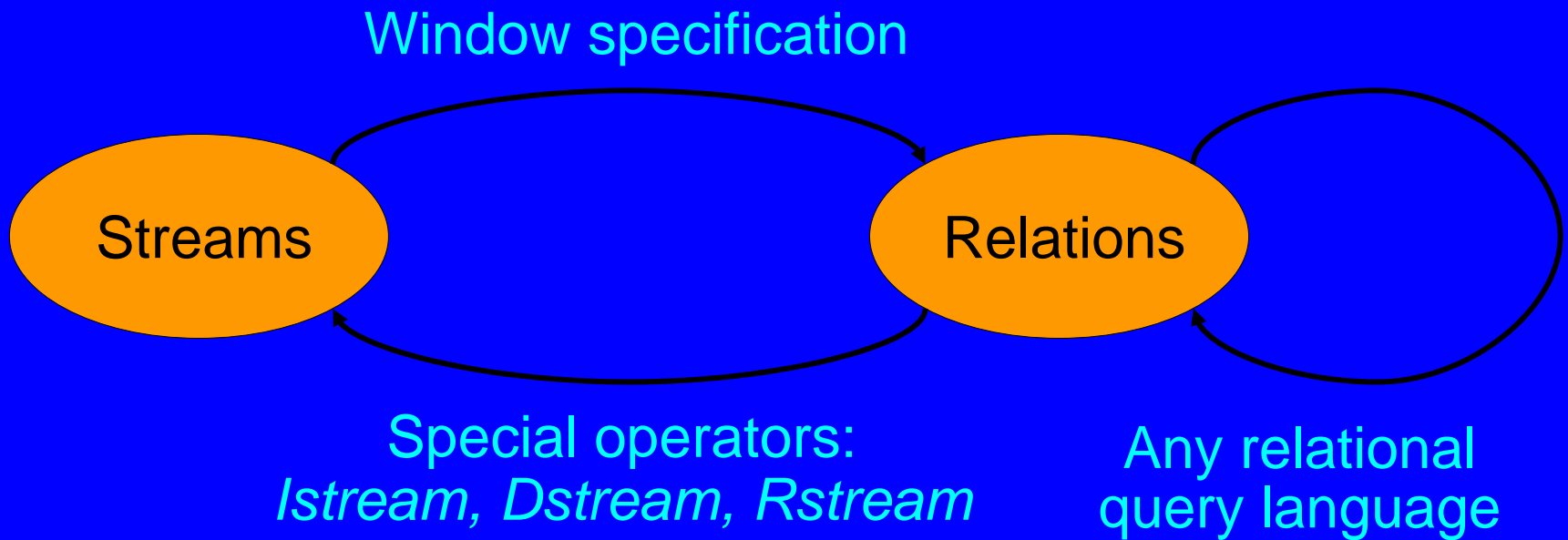
# Formalization (and Rest of Talk)

- Formal definitions for relations and streams
- Formal conversions between them
- Abstract semantics
- Concrete language: CQL
- Syntactic defaults and shortcuts
- Equivalence-based transformations
- Benchmark, system screenshots

# Relations and Streams

- Assume global, discrete, ordered time domain (more on this later)
- Relation
  - Maps *time*  $T$  to *set-of-tuples*  $R$
- Stream
  - Set of *(tuple,timestamp)* elements

# Conversions



# Conversion Definitions

- Stream-to-relation
  - $S[W]$  is a relation — at time  $T$  it contains all tuples in window  $W$  applied to stream  $S$  up to  $T$
  - When  $W = \infty$ , contains all tuples in stream  $S$  up to  $T$
- Relation-to-stream
  - $Istream(R)$  contains all  $(r, T)$  where  $r \in R$  at time  $T$  but  $r \notin R$  at time  $T-1$
  - $Dstream(R)$  contains all  $(r, T)$  where  $r \in R$  at time  $T-1$  but  $r \notin R$  at time  $T$
  - $Rstream(R)$  contains all  $(r, T)$  where  $r \in R$  at time  $T$

# Abstract Semantics

- Take any relational query language
- Can reference streams in place of relations
  - But must convert to relations using any window specification language  
( default window =  $[\infty]$  )
- Can convert relations to streams
  - For streamed results
  - For windows over relations  
(note: converts back to relation)

# Query Result at Time $T$

- Use all relations at time  $T$
- Use all streams up to  $T$ , converted to relations
- Compute relational result
- Convert result to streams if desired

# Abstract Semantics – Example 1

```
Select F.clerk, Max(O.cost)
From O [∞], F [Rows 1000]
Where O.orderID = F.orderID
Group By F.clerk
```

- Maximum-cost order fulfilled by each clerk in last 1000 fulfillments

# Abstract Semantics – Example 1

```
Select F.clerk, Max(O.cost)
From O [ $\infty$ ], F [Rows 1000]
Where O.orderID = F.orderID
Group By F.clerk
```

- At time  $T$ : entire stream  $O$  and last 1000 tuples of  $F$  as relations
- Evaluate query, update result relation at  $T$

# Abstract Semantics – Example 1

Select **Istream**(F.clerk, Max(O.cost))

From O [ $\infty$ ], F [Rows 1000]

Where O.orderID = F.orderID

Group By F.clerk

- At time  $T$ : entire stream  $O$  and last 1000 tuples of  $F$  as relations
- Evaluate query, update result relation at  $T$
- **Streamed result:** New element  $(\langle \text{clerk}, \text{max} \rangle, T)$  whenever  $\langle \text{clerk}, \text{max} \rangle$  changes from  $T-1$

# Abstract Semantics – Example 2

Relation CurPrice(stock, price)

Select stock, Avg(price)

From Istream(CurPrice) [Range 1 Day]

Group By stock

- Average price over last day for each stock

# Abstract Semantics – Example 2

Relation `CurPrice(stock, price)`

Select stock, Avg(price)

From `Istream(CurPrice)` [Range 1 Day]

Group By stock

- *Istream* provides history of *CurPrice*
- Window on history, back to relation, group and aggregate

# Digression: Time

- Relation updates carry timestamps too
- “Well-behaved” streams and relation updates:
  - Arrive in order
  - No “delayed activity” – no long pauses then resume with very old timestamps
- Semantics of query language tacitly assumes well-behaved streams and updates
  - Poor behavior handled separately, not within language
  - Distinction from Aurora, which does not separate

# Time Digression (cont'd)

- Simplest scenario: global system clock
  - Stream elements and relation updates timestamped on entry to system
  - Guaranteed ordering, no delayed activity
  - Query results also based on system time

# Flexible Application-Defined Time

- Streams and relation updates contain application timestamps
  - May arrive out of order
  - May pause and resume
- Query results in application time
- Application generates “heartbeats”
  - Or deduce heartbeat from parameters: scrambling, skew, latency, clock progress
  - *Reminder: separate from query language semantics*

# Concrete Language – CQL

- Relational query language: SQL
- Window specification language derived from SQL-99
  - Tuple-based windows
  - Time-based windows
  - Partitioned windows
- Simple “X% Sample” construct
  - May incorporate emerging SQL standard

# CQL (cont'd)

- Syntactic shortcuts and defaults
  - *So easy queries are easy to write and simple queries do what you expect*
- Equivalences
  - Basis for query-rewrite optimizations
  - Includes all relational equivalences, plus new stream-based ones
- Examples: already seen some, more later

# Shortcuts and Defaults

- Prevalent stream-relation conversions can make some queries cumbersome
  - *Easy queries should be easy to write*
- Two defaults:
  - **Omitted window:** Default  $[\infty]$
  - **Omitted relation-to-stream operator:**  
Default *Istream* operator on:
    - Monotonic outermost queries
    - Monotonic subqueries with windows

# The Simplest CQL Query

Select \* From Strm

- Had better return *Strm* (It does)
  - Default  $[\infty]$  window for *Strm*
  - Default *Istream* for result

# Simple Join Query

Select \* From Strm, Rel Where Strm.A = Rel.B

- Default  $[\infty]$  window on *Strm*, but often want **Now** window for stream-relation joins

Select Istream(O.orderID, A.City)

From Orders O, AddressRel A

Where O.custID = A.custID

# Simple Join Query

Select \* From Strm, Rel Where Strm.A = Rel.B

- Default  $[\infty]$  window on *Strm*, but often want **Now** window for stream-relation joins

Select Istream(O.orderID, A.City)

From Orders O [**Now**], AddressRel A

Where O.custID = A.custID

- We decided against a separate default

# Equivalences and Transformations

- All relational equivalences apply to all relational constructs directly
  - Queries are highly relational
- Many low-level transformations in implementation
- Two new language-based transformations, more to come
  - *Window reduction*
  - *Filter-window commutativity*

# Window Reduction

Select Istream(L) From S [ $\infty$ ] Where C

is equivalent to

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

- Question for audience – **time to wake up!**
  - Why *Rstream* and not *Istream* in second query?
- Answer: Consider stream <5>, <5>, <5>, <5>, ...

# Window Reduction (cont'd)

Select Istream(L) From S [ $\infty$ ] Where C

is equivalent to

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

- First query form is very common due to defaults
- In a naïve implementation second form is much more efficient

# Some Subtleties (or Dirty Laundry): *Istream*, *Rstream*, and Windows

- Example: Emit 5-second moving average on every timestep  
`Select Istream(Avg(A)) From S [Range 5 seconds]`  
Only emits a result when average changes
- To emit a result on every timestep  
`Select Rstream(Avg(A)) From S [Range 5 seconds]`
- To emit a result every second  
`Select Rstream(Avg(A)) From S  
[Range 5 seconds Slide 1 second]`

# More on *Istream*, *Rstream*, and Windows

- Example: windowed join, emit new result tuples whenever there's a new input tuple

```
Select Rstream(*) From S1 [w1], S2 [w2]
```

- *Istream* may not emit results in presence of duplicates
- *Rstream* emits entire join result at each timestep or slide interval
- Correct solution requires union of multiple joins, somewhat messy

# Some Windowing Rules of Thumb

- *Dstream* used infrequently but is needed
- *Istream* and *Dstream* can be expressed using *Rstream* and other constructs — but don't try it at home
- *Istream* always intuitive in presence of keys
- *Rstream* sensible with **Now** or **Slide** windows
- Row-based windows nondeterministic in presence of duplicate timestamps
- No need for partitioned time-based windows

(end of long aside)

# Filter-Window Commutativity

- Another question for the audience

When is

Select L From S [window] Where C

equivalent to

Select L From (Select L From S Where C) [window]

- Is this transformation always advantageous?

# Constraint-Based Transformations

- Recall first example query (simplified)

```
Select Sum(O.cost)
From Orders O, Fulfillments F [Range 1 Day]
Where O.orderID = F.orderID
```

- If orders always fulfilled within one week

```
Select Sum(O.cost)
From Orders O [Range 8 Days],
      Fulfillments F [Range 1 Day]
Where O.orderID = F.orderID
```

- Useful constraints: **keys, stream referential integrity, clustering, ordering**

# Evaluating a Query Language

- Is it expressive enough?
- Is it user-friendly?
- Can it be implemented?
- Can it be implemented efficiently?

# Evaluating a Query Language

- Is it expressive enough?

**Yes**, see our **Stream Query Repository**

Online auctions, network traffic management, habitat monitoring, military logistics, immersive environments, road traffic monitoring

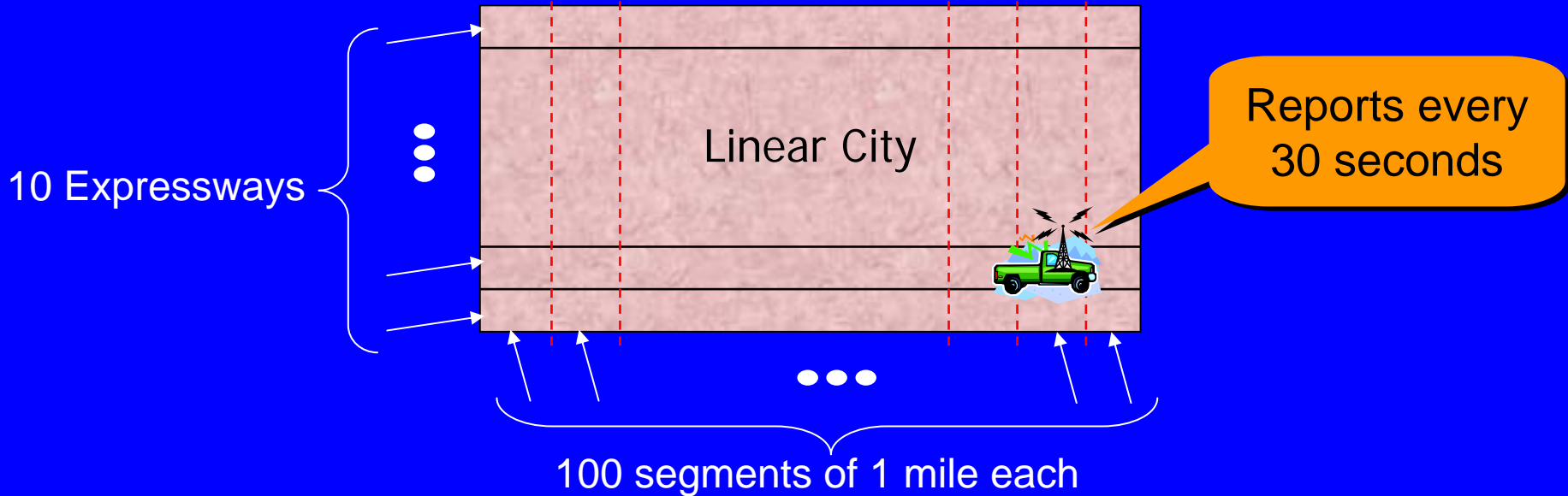
- Is it user-friendly? **To be determined**

- Can it be implemented? **Yes, see our demo**

- Can it be implemented efficiently?

**We think so, with lots of fun research**

# Stream System Benchmark: "Linear Road"



## Main Input Stream: Car Locations (CarLocStr)

car_id	speed	exp_way	lane	x_pos
1000	55	5	3 (Right)	12762
1035	30	1	0 (Ramp)	4539
...	...	...	...	...

# Linear Road Benchmark

- Primarily developed by Aurora group
  - With some help from us
- Suite of continuous queries based on real traffic management proposals. Example CQs:
  - Stream car segments based on x-positions (**easy**)
  - Identify probable accidents (**medium**)
  - Compute toll whenever car enters segment (**hard**)
- **Metric:** Scale to as many expressways as possible without falling behind

# Easy Example

Monitor speed and segments of cars 1-100

Select car\_id, speed, x\_pos/5280 as segment  
From CarLocStr  
Where car\_id >= 1 and car\_id <= 100

+/-	Timestamp	car_id	speed	segment
+	6/6/03 12:34:05	22	23	0
+	6/6/03 12:34:05	10	23	1
+	6/6/03 12:34:05	16	23	11
+	6/6/03 12:34:05	2	30	12
+	6/6/03 12:34:05	25	25	15
+	6/6/03 12:34:05	23	26	18
+	6/6/03 12:34:05	18	20	24
+	6/6/03 12:34:05	5	30	29
+	6/6/03 12:34:05	12	26	40
+	6/6/03 12:34:05	1	27	41
+	6/6/03 12:34:05	4	23	47
+	6/6/03 12:34:05	29	30	53
+	6/6/03 12:34:05	28	30	55
+	6/6/03 12:34:05	7	32	65
+	6/6/03 12:34:05	6	28	99
+	6/6/03 12:34:05	8	30	96
+	6/6/03 12:34:05	9	30	93
+	6/6/03 12:34:05	14	27	90
+	6/6/03 12:34:05	27	27	82
+	6/6/03 12:34:05	26	28	78
+	6/6/03 12:34:05	20	23	77
+	6/6/03 12:34:05	3	23	76
+	6/6/03 12:34:05	21	23	75
+	6/6/03 12:34:05	13	27	71
+	6/6/03 12:34:05	19	32	68

# Hard Example

Whenever a car enters a segment, issue it the current toll for that segment

+/-	Timestamp	E.car_id	E.seg	T.toll
+	6/6/03 12:34:35	6	98	8
+	6/6/03 12:34:35	8	95	4
+	6/6/03 12:34:35	9	92	10
+	6/6/03 12:34:35	14	89	7
+	6/6/03 12:34:35	27	81	9
+	6/6/03 12:34:35	26	77	4
+	6/6/03 12:34:35	21	74	9
+	6/6/03 12:34:35	13	70	2
+	6/6/03 12:34:35	19	67	9
+	6/6/03 12:34:35	11	65	5
+	6/6/03 12:34:35	17	60	4
+	6/6/03 12:34:35	24	35	2
+	6/6/03 12:34:36	53	95	5
+	6/6/03 12:34:36	55	91	8
+	6/6/03 12:34:36	45	90	6
+	6/6/03 12:34:36	35	85	10
+	6/6/03 12:34:36	40	79	8
+	6/6/03 12:34:36	780	79	9
+	6/6/03 12:34:36	784	74	10
+	6/6/03 12:34:36	37	73	3
+	6/6/03 12:34:36	46	71	6
+	6/6/03 12:34:36	739	71	7
+	6/6/03 12:34:36	757	67	10
+	6/6/03 12:34:36	776	65	6
+	6/6/03 12:34:36	50	64	7

# Hard Example in CQL

```
Select Rstream(E.car_id, E.seg, T.toll)
From CarSegEntryStr [NOW] as E, SegToll as T
Where E.loc = T.loc
```

CarSegEntryStr: Select Istream(\*) From CurCarSeg

CurCarSeg:

```
Select car_id, x_pos/5280 as seg,
       Location(expr_way, dir, x_pos/5280) as loc
From CarLocStr [Partition By car_id Rows 1]
```

# Hard Example (cont'd)

## SegToll:

```
Select S.loc, BaseToll * (V.volume - 150)2
From SegAvgSpeed as S, SegVolume as V
Where S.loc = V.loc and S.avg_speed < 40.0
```

## SegAvgSpeed:

```
Select loc, Avg(speed) as avg_speed
From CarLocStr [Range 5 minutes]
Group By location(expr_way, dir, x_pos/5280) as loc
```

## SegVolume:

```
Select loc, Count(*) as volume
From CurCarSeg
Group By loc
```

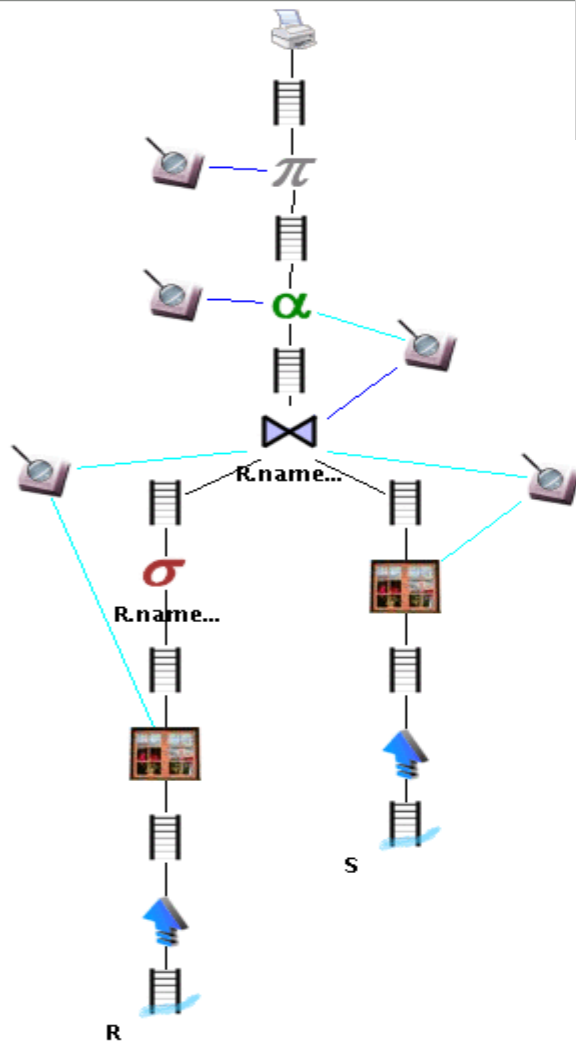
# STREAM Query Interface

File View Script Stream Source

select S.name, max(R.num) from R rows 10, S rows 20 where R.name <= "I" and R.name = S.name Group By S.name

select S.name, max(R.num) from R rows 10, S rows 20 where R.name <= "I" and R.name = S.name Group By S.name

OK



+/-	Timestamp	S.name	MAX R.num
+	9/3/03 17:50:23	hamster	2
+	9/3/03 17:50:24	dog	3
+	9/3/03 17:50:25	elephant	4
-	9/3/03 17:50:28	dog	3
+	9/3/03 17:50:28	dog	7
-	9/3/03 17:50:32	hamster	2
+	9/3/03 17:50:32	hamster	11
-	9/3/03 17:50:33	dog	7
+	9/3/03 17:50:33	dog	12
-	9/3/03 17:50:34	elephant	4
+	9/3/03 17:50:34	elephant	13
-	9/3/03 17:50:37	dog	12
+	9/3/03 17:50:37	dog	16
-	9/3/03 17:50:39	dog	16
+	9/3/03 17:50:39	dog	18
-	9/3/03 17:50:42	hamster	11
+	9/3/03 17:50:42	hamster	22
-	9/3/03 17:50:43	dog	18
+	9/3/03 17:50:43	dog	23
-	9/3/03 17:50:44	elephant	13
+	9/3/03 17:50:44	elephant	24

stream S reads tuples of S from the ...

stream S reads tuples of S from the ...

# STREAM Query Interface

File View Script Stream Source

select S.name, max(R.num) from R rows 10, S rows 20 where R.name <= "i" and R.name...

select\_20

Property	Value
comments	
filter-cond-0	R.name <= "i"
filter-cond-count	1
input-queue	simple-queue_15
monitor	0
output-queue	simple-queue_19
predicate	

**Legend**

- output**  
An output operator sends its tuples to a client process over the network.
- queue**  
Queues are used to pass tuples between operators.
- project**  
The project operator puts its output tuples into this relational synopsis.
- synopsis**  
Interface for all synopsis
- aggregate**  
Group-by Aggregation operator
- binary-join**  
An operator which performs a join over two streams or relations.
- select**  
Outputs a subset of its input tuples based on a filter predicate.
- seq-window**  
A sequential window operator implements one or more time or tuple-based windows. ...
- stream-shepherd**  
The Stream Shepherd operator for a stream S reads tuples of S from the ...

# “Introspection” Queries

- Real-time monitoring of query execution and general system properties, for example:
  - Tuple-flow rate in a queue
  - Selectivity of a join
  - Ordering of a stream
  - Overall memory usage
  - Etc.
- Monitoring performed using regular CQL queries over extensible system-generated **property stream**

# Query Plan Monitoring

File View Script Stream Source

select \* from VisS rows 20, VisR rows 20 where VisS.name = VisR.name

369

28

0.04

342

VisS

VisR

**simple-queue\_33**

**simple-queue**

Property	Value
comments	
memory	10000
monitor	0
publishes-state	yes
schema	
tuple-flow	0

**Legend**

- output**  
An output operator sends its tuples to a client process over the network.
- queue**  
Queues are used to pass tuples between operators.
- binary-join**  
An operator which performs a join over two streams or relations.
- synopsis**  
Interface for all synopses
- seq-window**  
A sequential window operator implements one or more time or tuple-based windows. ...
- stream-shepherd**  
The Stream Shepherd operator for a stream S reads tuples of S from the ...
- stream-queue**  
A queue that is used to receive the raw output of a stream. Tuples from the ...

## For more information:

- Talk to me this afternoon

- Visit our web site:

<http://www-db.stanford.edu/stream>

## The Stanford STREAM Team:

Arvind Arasu, Brian Babcock, Shivnath Babu, John Cieslewicz, Mayur Datar, Keith Ito, Rajeev Motwani, Itaru Nishizawa, Utkarsh Srivastava, Dilys Thomas