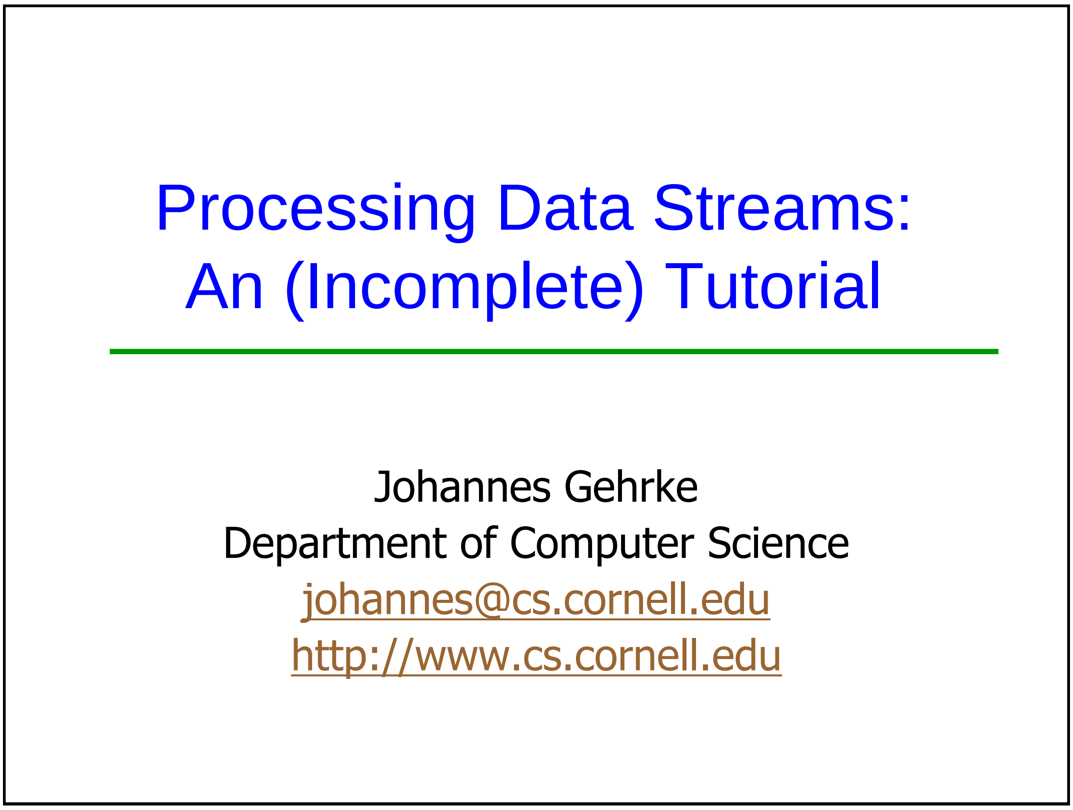
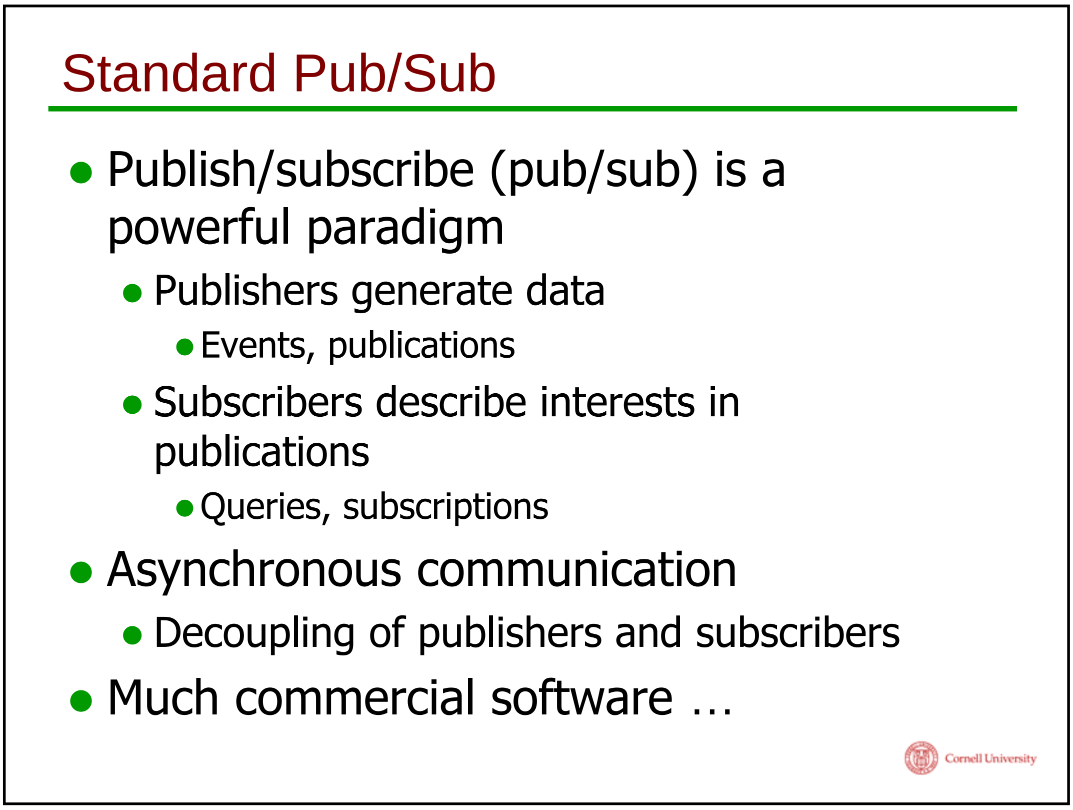


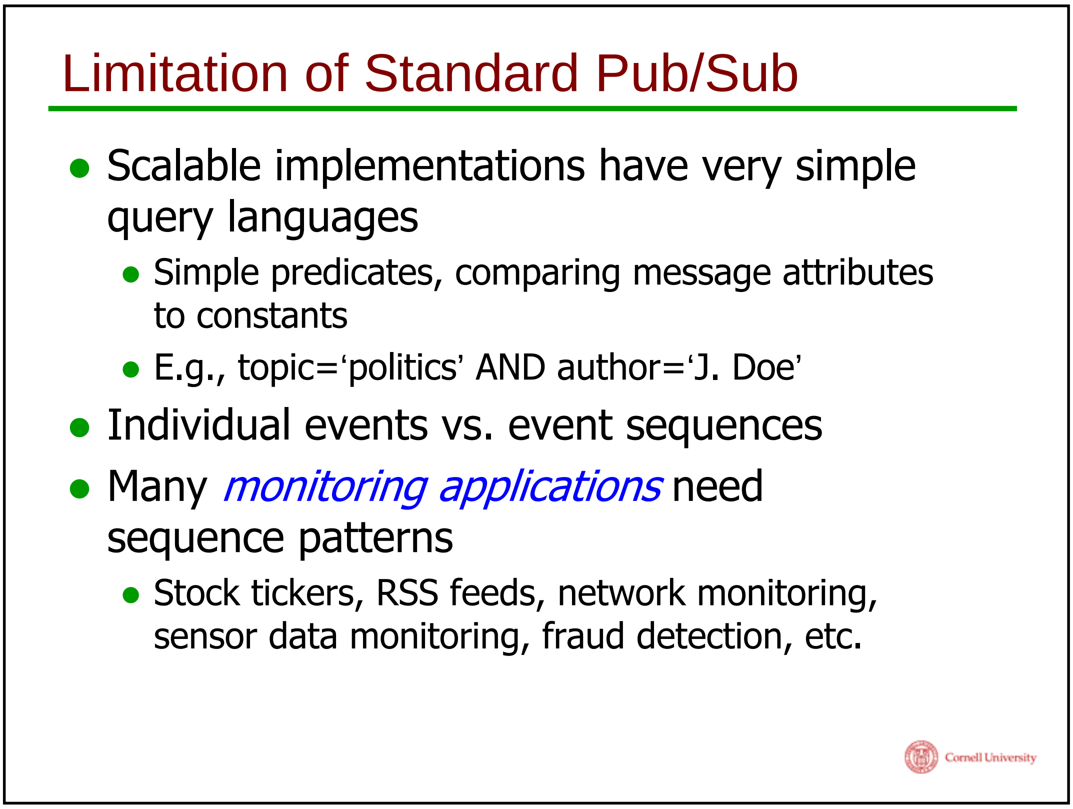
# Processing Data Streams: An (Incomplete) Tutorial

Johannes Gehrke
Department of Computer Science
johannes@cs.cornell.edu
http://www.cs.cornell.edu



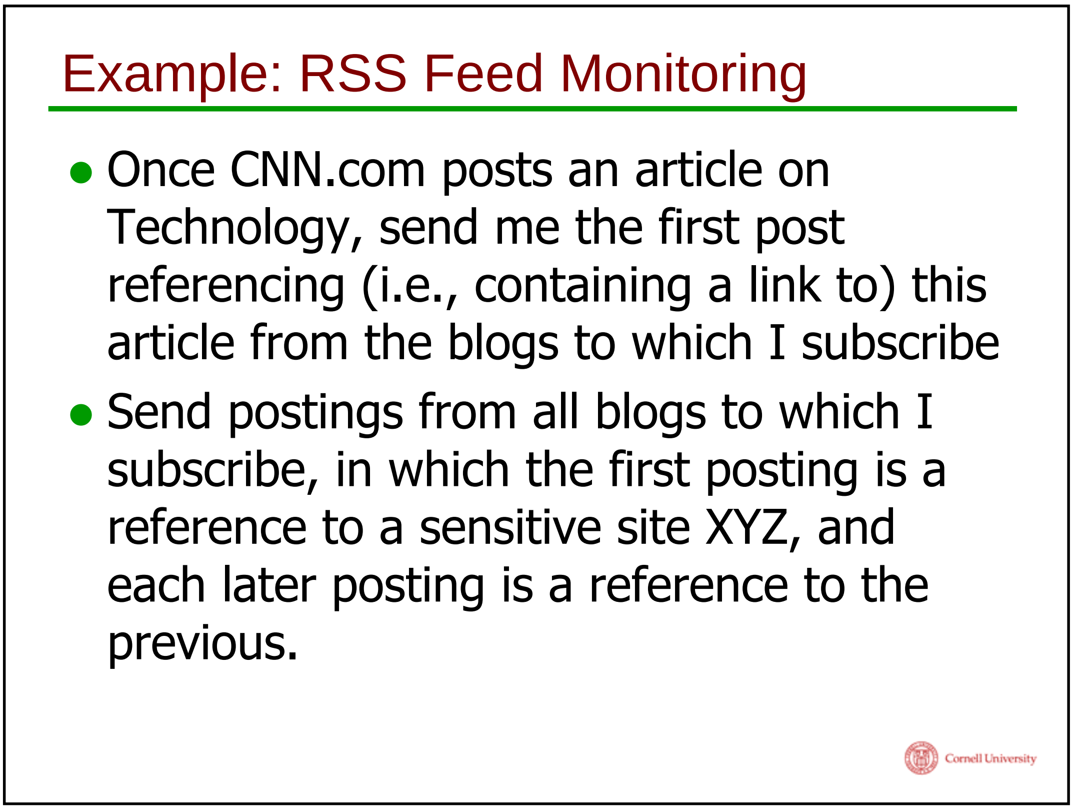
## Standard Pub/Sub

- Publish/subscribe (pub/sub) is a powerful paradigm
  - Publishers generate data
    - Events, publications
  - Subscribers describe interests in publications
    - Queries, subscriptions
- Asynchronous communication
  - Decoupling of publishers and subscribers
- Much commercial software …



## Limitation of Standard Pub/Sub

- Scalable implementations have very simple query languages
  - Simple predicates, comparing message attributes to constants
  - E.g., topic=‘politics’ AND author=‘J. Doe’
- Individual events vs. event sequences
- Many *monitoring applications* need sequence patterns
  - Stock tickers, RSS feeds, network monitoring, sensor data monitoring, fraud detection, etc.



## Example: RSS Feed Monitoring

- Once CNN.com posts an article on Technology, send me the first post referencing (i.e., containing a link to) this article from the blogs to which I subscribe
- Send postings from all blogs to which I subscribe, in which the first posting is a reference to a sensitive site XYZ, and each later posting is a reference to the previous.

## Example: System Event Log Monitoring

- In the past 60 seconds, has the number of failed logins (security logs) increased by more than 5? (break-in attempt)
- Have there been any failed connections in the past 15 minutes? If yes, is the rate increasing?
- Have there been any disk errors in the past 30 minutes? If yes, is the rate increasing? (failed disk indicator)
- Have there been any critical errors (those added to the dbase table to monitor by administrators) in the past 10 minutes?

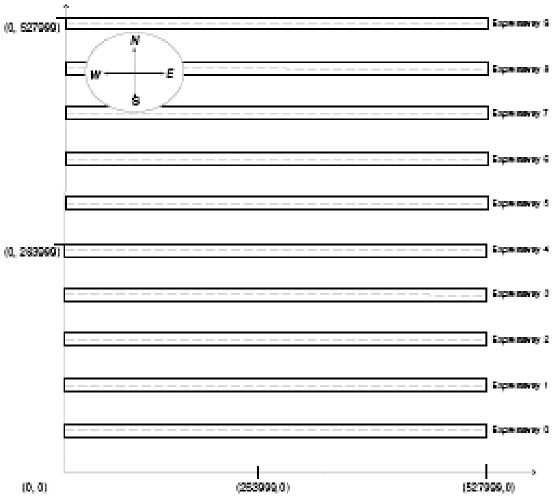
## Example: Stock Monitoring

- Notify me when the price of IBM is above $83, and the *first* MSFT price afterwards is below $27.
- Notify me when some stock goes up by at least 5% from one transaction to the next.
- Notify me when the price of any stock increases monotonically for ≥30 min.
- Notify me when the next IBM stock is above its 52-week average.

# Linear Road Benchmark

Linear City

- 100x100 miles
- 10 parallel expressways, 100 segments each
- Each expressway has 4 lanes in each direction
  - 3 travel lanes
  - 1 entry/exit lane
- Vehicles with sensors that report their position



Cornell University

Figure from Linear Road: A Stream Data Management Benchmark, VLDB 2004

# Linear Road Benchmark (2)

- Vehicle:
  - Begins at some segment and exists at some segments
  - Reports its position every 30 seconds
  - Vehicle speed is set such that:
    - One report from entrance and exit ramps
    - At least one report from each segment
- One accident every 20 minutes
  - Reduced speed in that segment
  - Takes 10-20 minutes to clear out the accident

Cornell University

# Linear Road Benchmark (3)

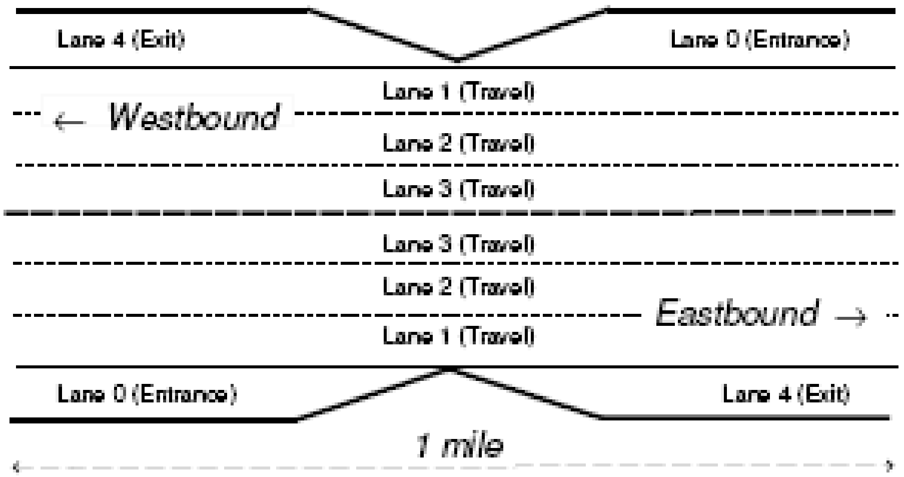


Figure from Linear Road: A Stream Data Management Benchmark, VLDB 2004

# Linear Road Benchmark (4)

- Streams:
  - Position reports
  - Historical query requests:
    - Account balances
    - Daily expenditures
    - Travel time estimation

# Linear Road Benchmark (5)

- Benchmark requirements:
  - Compute tolls every time a position is reported
    - Toll notification at every position update
    - Toll assessment at every segment crossing
  - Accident detection
    - Four consecutive identical position reports
    - Accident notification: If there is an accident in a segment, notify all incoming vehicles of the accident
  - Historical queries
    - Account balance
    - Daily expenditure
    - Travel time estimation

# Linear Road Benchmark (6)

- System achieves L-Rating
  - Maximum scale factor at which the system meets response time and accuracy requirements
- Example of DSMS versus dinosaur system: Response time

| Expressways | X | Aurora |
|---|---|---|
| 0.5 | 3 | 1 |
| 1 | 2031 | 1 |
| 1.5 | ~16000 | 1 |
| 2 | ~52000 | 2 |

## Solutions?

- Traditional pub/sub
  - Scalable, but not expressive enough
- Database Management System
  - Static datasets
  - One-shot queries
  - Triggers
- Data Stream Management Systems
- Event Processing Systems

## Real-Time DSP Requirements

(1) Support a high-level "StreamSQL" language

(2) Deal with out-of-order data

(3) Generate predictable and repeatable outcomes

(4) Integrate well with static data

(5) Fault-tolerance

(6) Scale with hardware resources

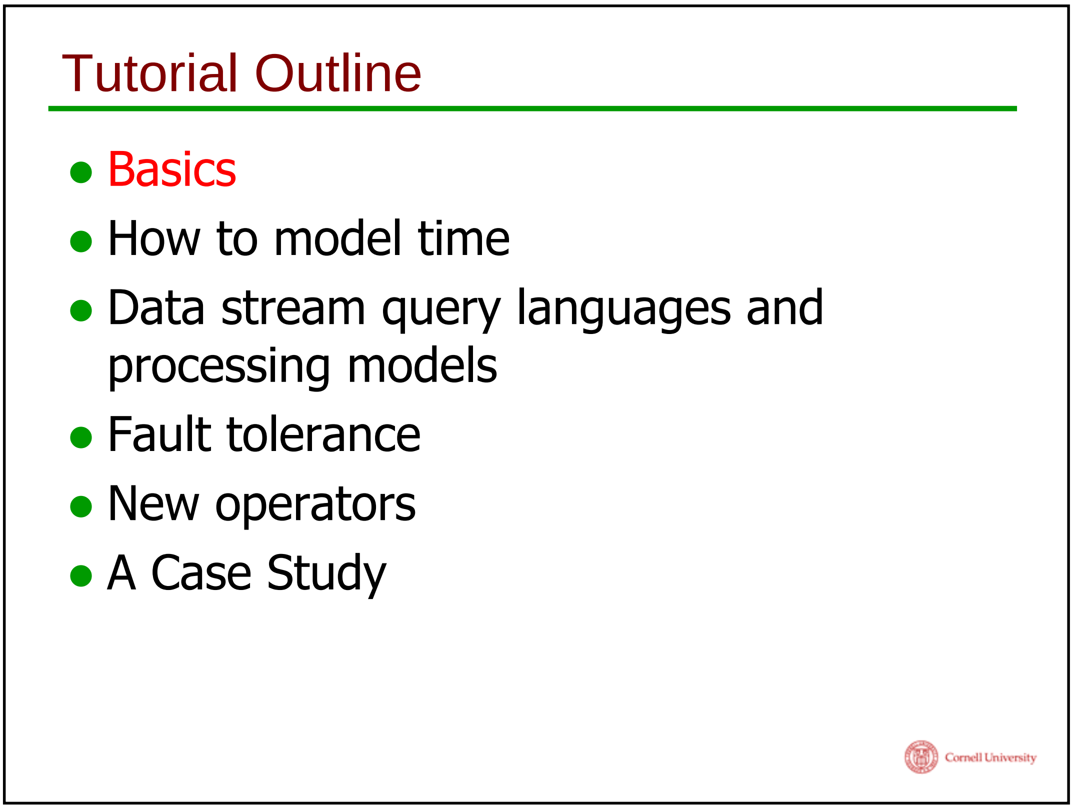
(7) Low latency → process data as it streams by ("in-stream processing"); no requirement to store data first

## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
  - STREAM and CQL
  - Cayuga
- Fault tolerance
- New operators
  - Change detection
  - Burst detection
- A Case Study

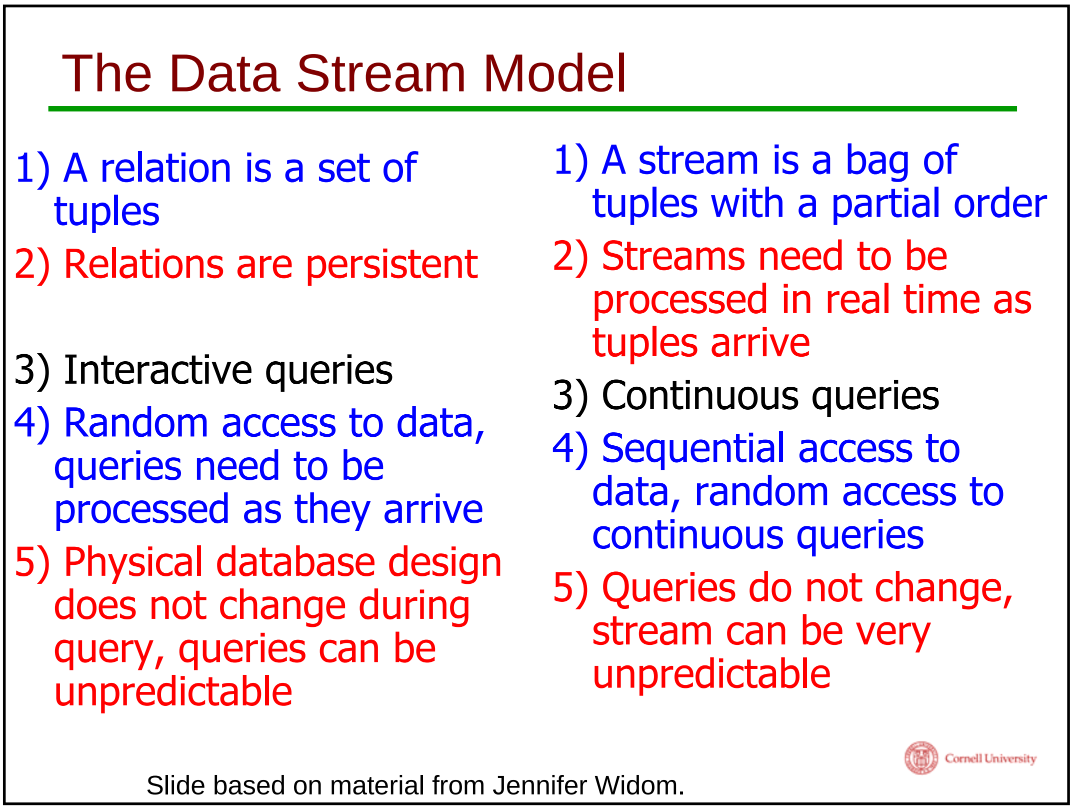
## Caveat

- To trade breadth for some depth, this tutorial ignores many important topics among them:
  - In-depth discussion of applications
  - Query processing
  - Heartbeats
  - Query optimization
  - Query rewrite
  - Access methods
  - XML
  - Theoretical results on the language side



## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
- Fault tolerance
- New operators
- A Case Study



## The Data Stream Model

1) A relation is a set of tuples

2) Relations are persistent

3) Interactive queries

4) Random access to data, queries need to be processed as they arrive

5) Physical database design does not change during query, queries can be unpredictable

1) A stream is a bag of tuples with a partial order

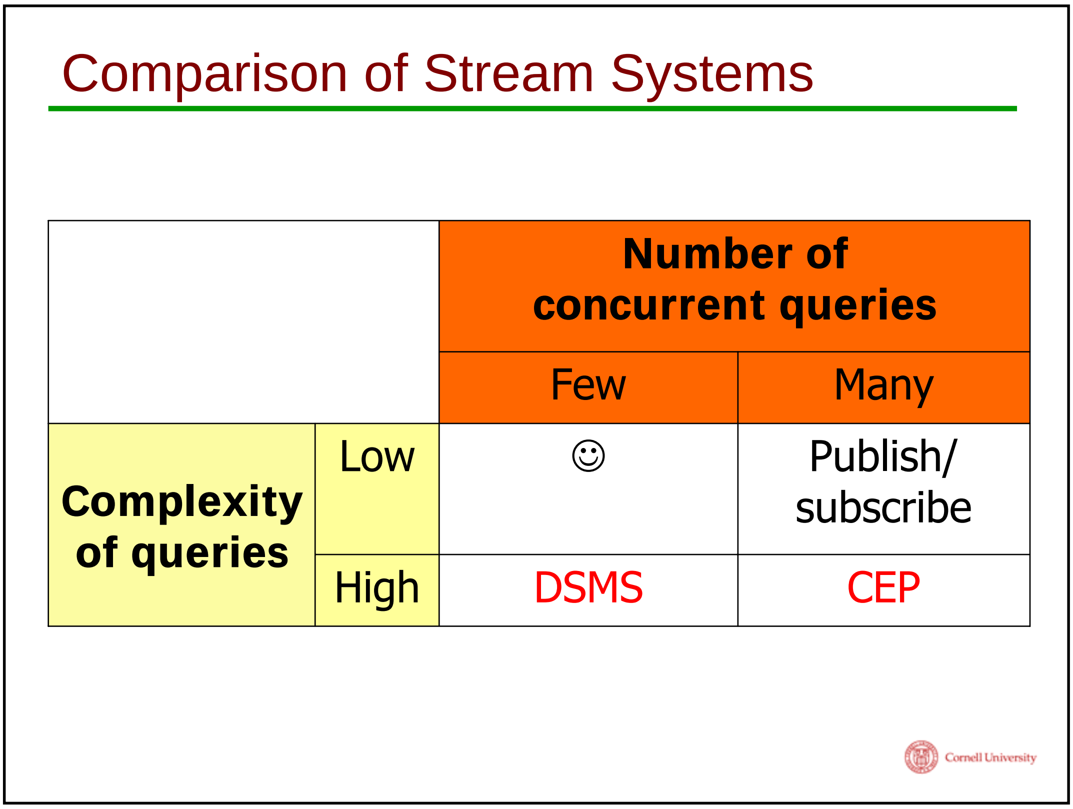
2) Streams need to be processed in real time as tuples arrive

3) Continuous queries

4) Sequential access to data, random access to continuous queries

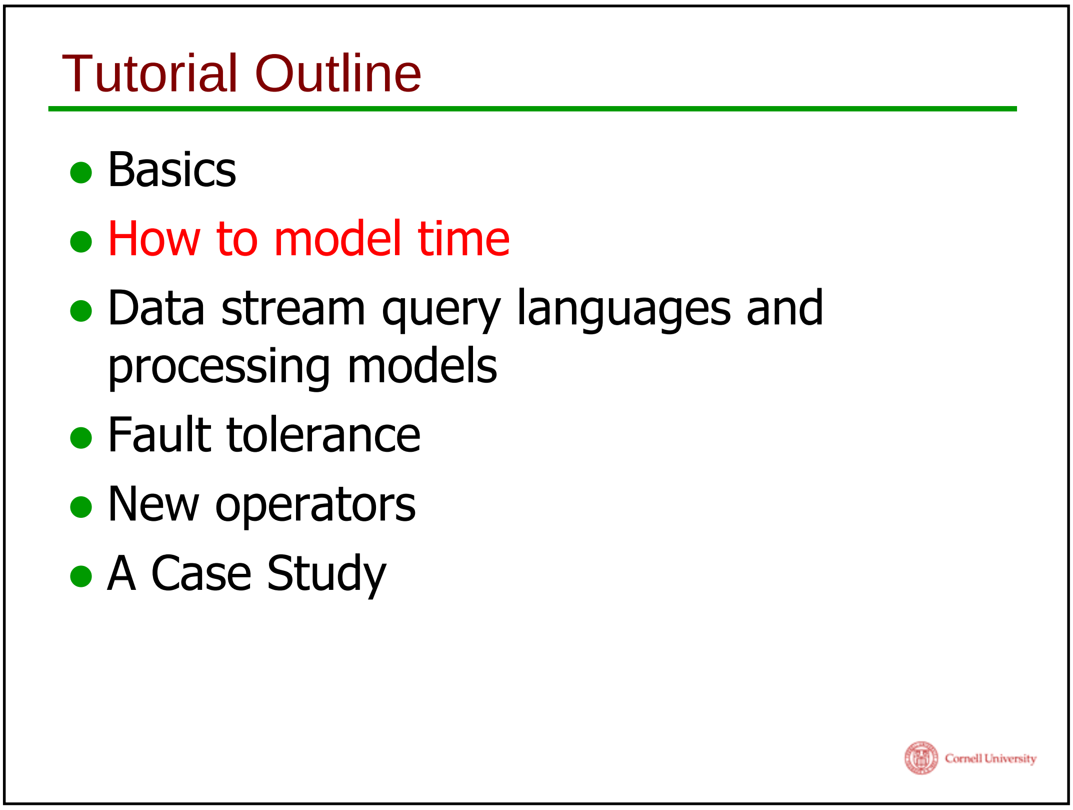
5) Queries do not change, stream can be very unpredictable

Slide based on material from Jennifer Widom.



## Comparison of Stream Systems

| | | **Number of concurrent queries** | |
|---|---|---|---|
| | | Few | Many |
| **Complexity of queries** | Low | ☺ | Publish/ subscribe |
| | High | DSMS | CEP |



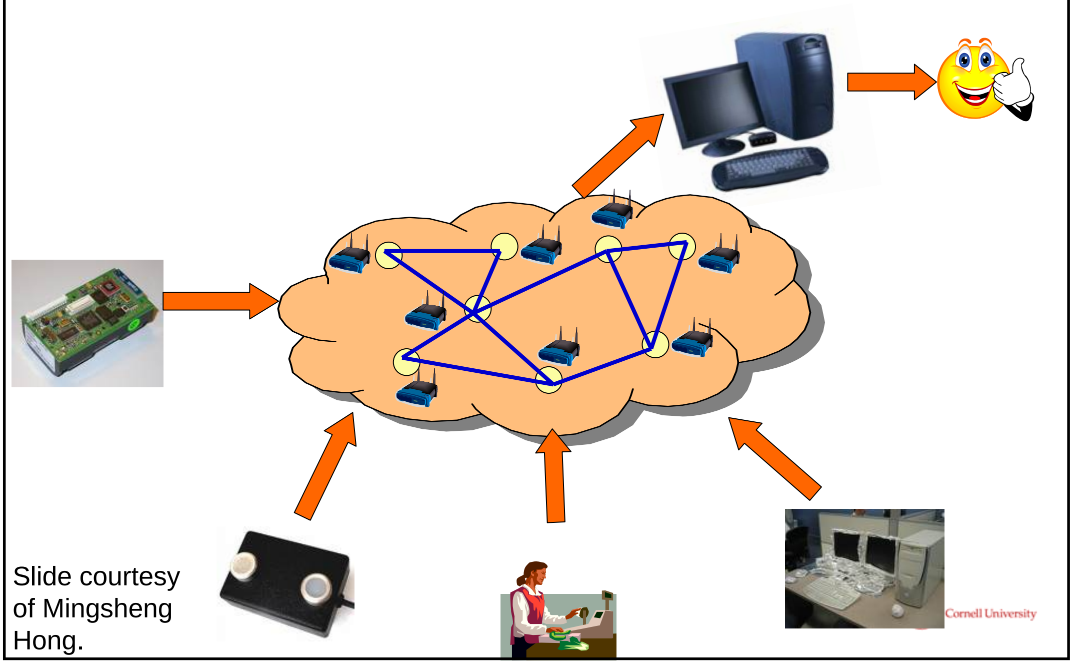
## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
- Fault tolerance
- New operators
- A Case Study

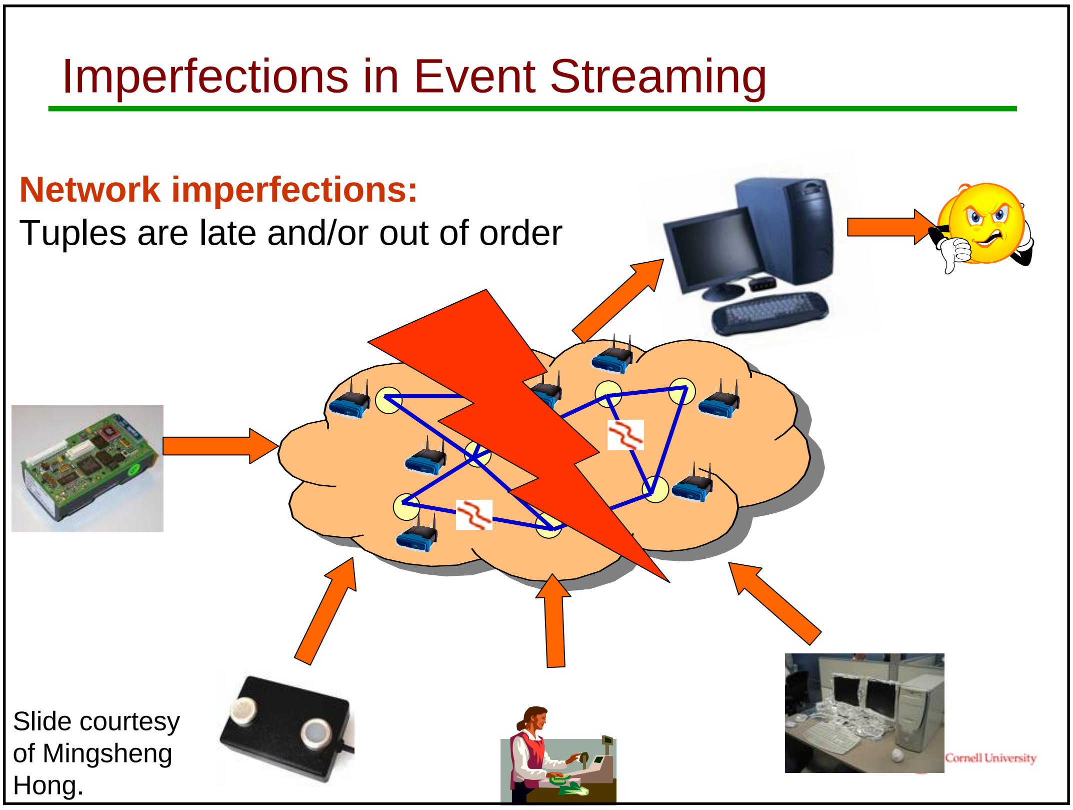
## Temporal Model

- Questions:
  - How are timestamps defined?
  - What is the timestamp of an output record?
- Approaches:
  - Point timestamps
  - Interval timestamps
- Surprises like E1;(E2;E3)=E2;(E1;E3)?

## Imperfections in Event Streaming



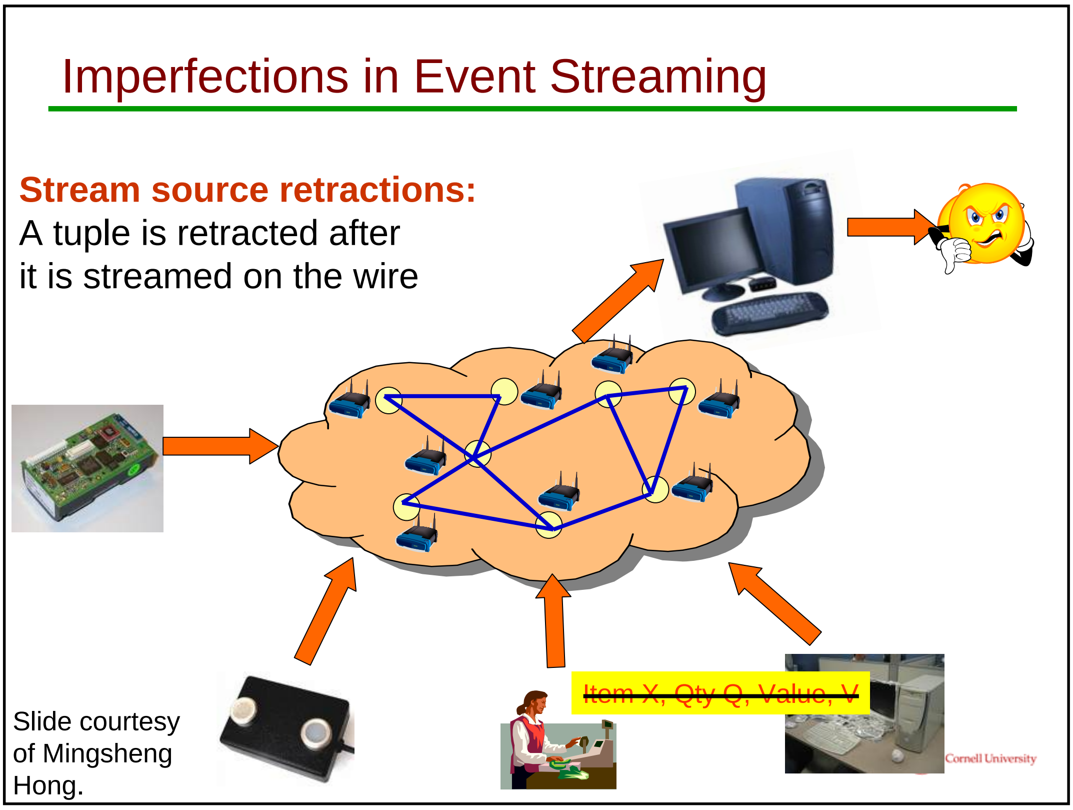
Slide courtesy of Mingsheng Hong.



## Imperfections in Event Streaming

**Network imperfections:**
Tuples are late and/or out of order

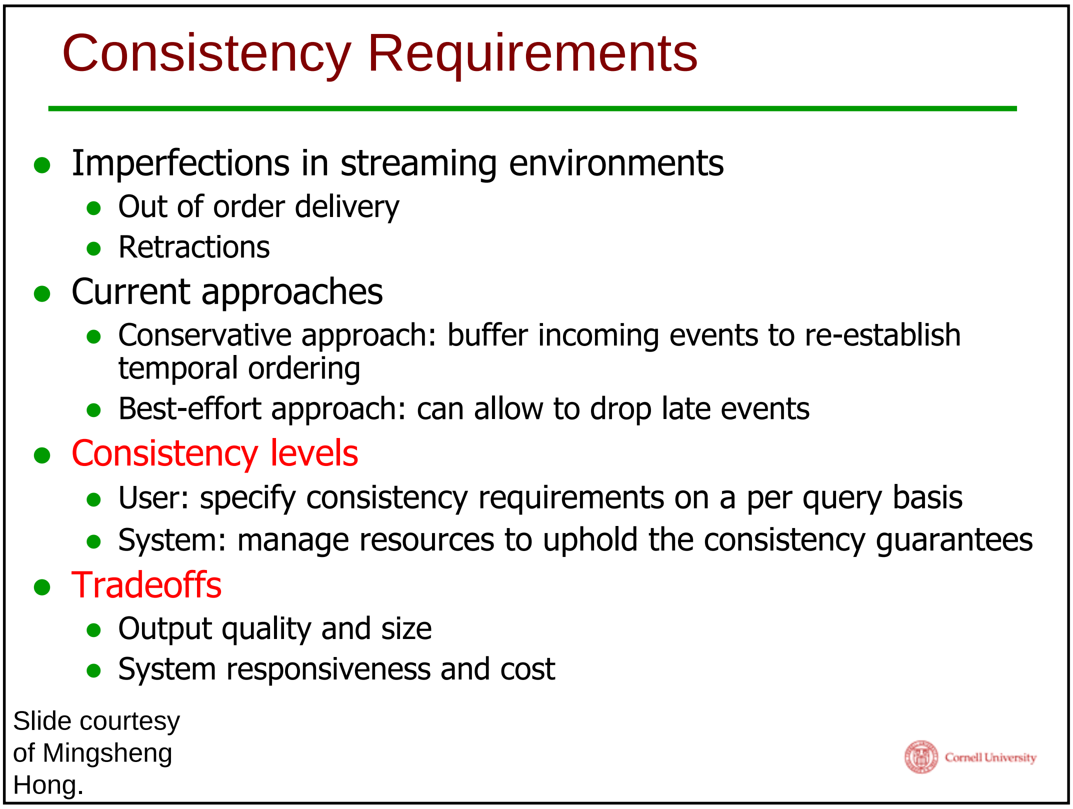
Slide courtesy of Mingsheng Hong.



## Imperfections in Event Streaming

**Stream source retractions:**
A tuple is retracted after
it is streamed on the wire

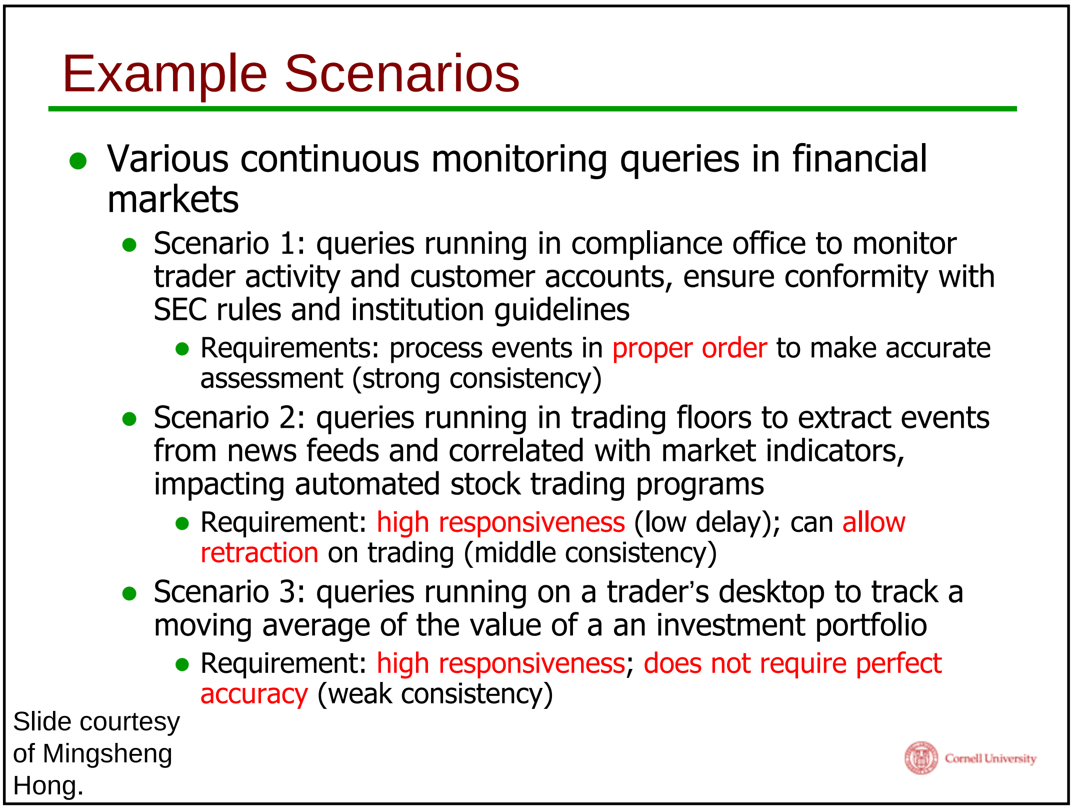
Slide courtesy of Mingsheng Hong.



## Consistency Requirements

- Imperfections in streaming environments
  - Out of order delivery
  - Retractions
- Current approaches
  - Conservative approach: buffer incoming events to re-establish temporal ordering
  - Best-effort approach: can allow to drop late events
- Consistency levels
  - User: specify consistency requirements on a per query basis
  - System: manage resources to uphold the consistency guarantees
- Tradeoffs
  - Output quality and size
  - System responsiveness and cost

Slide courtesy of Mingsheng Hong.



## Example Scenarios

- Various continuous monitoring queries in financial markets
  - Scenario 1: queries running in compliance office to monitor trader activity and customer accounts, ensure conformity with SEC rules and institution guidelines
    - Requirements: process events in proper order to make accurate assessment (strong consistency)
  - Scenario 2: queries running in trading floors to extract events from news feeds and correlated with market indicators, impacting automated stock trading programs
    - Requirement: high responsiveness (low delay); can allow retraction on trading (middle consistency)
  - Scenario 3: queries running on a trader's desktop to track a moving average of the value of a an investment portfolio
    - Requirement: high responsiveness; does not require perfect accuracy (weak consistency)

Slide courtesy of Mingsheng Hong.

## Key Insight

- Optimistic query processing provides a spectrum of consistency levels

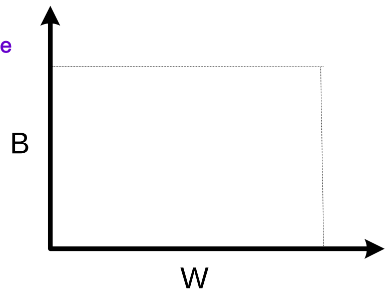
Slide courtesy of Mingsheng Hong.

Cornell University

## Consistency Domain and Levels

Blocking

Late & conservative



Fast & optimistic

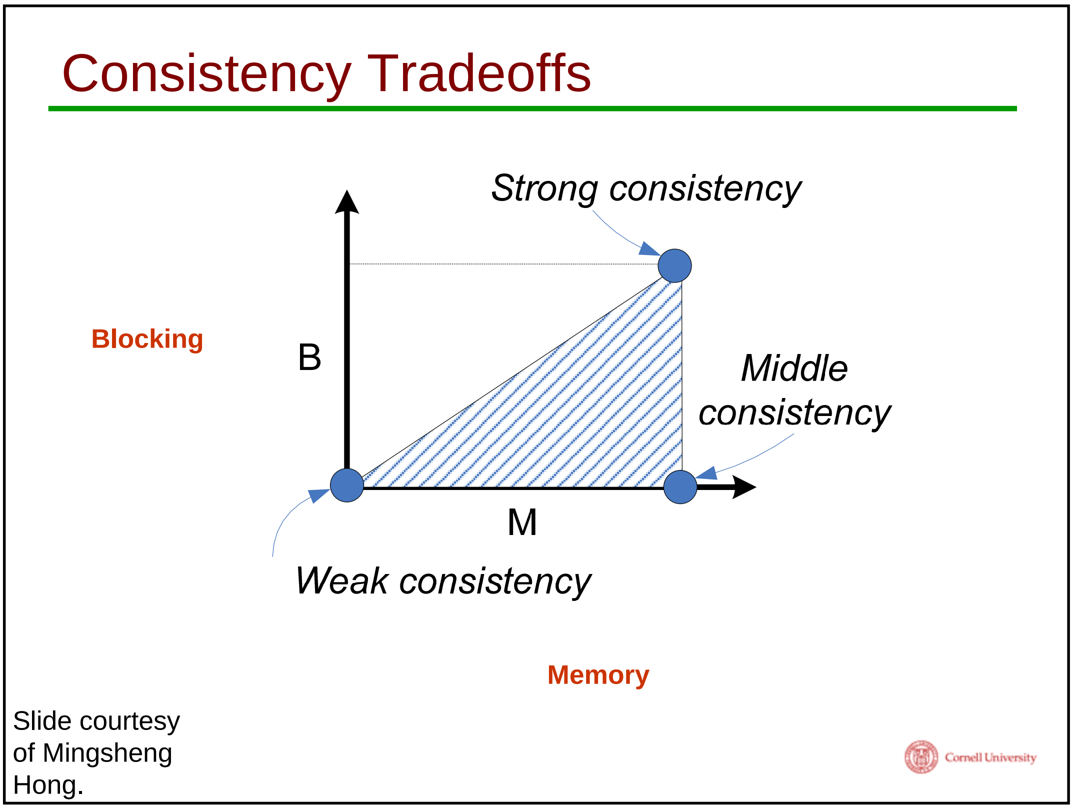
Memory

Cheap &
less correct

Expensive &
more correct

Slide courtesy of Mingsheng Hong.

Cornell University



## Consistency Tradeoffs

Strong consistency

Blocking

B

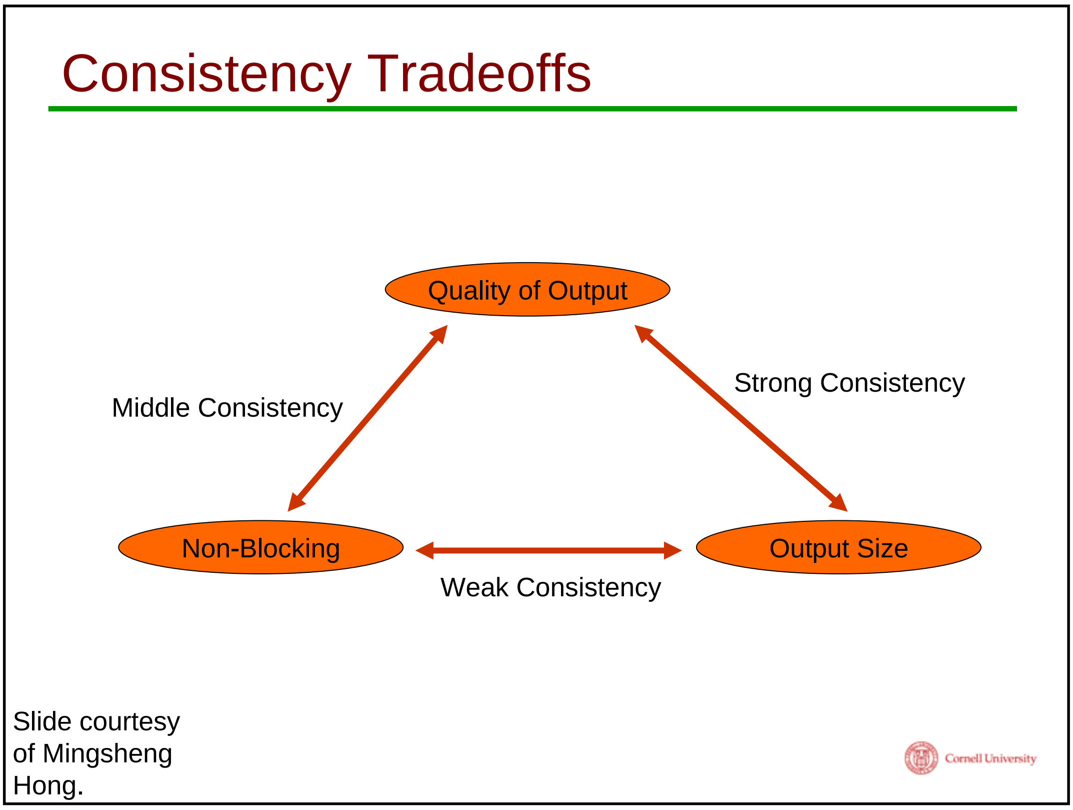
Middle consistency

M

Weak consistency

Memory

Slide courtesy of Mingsheng Hong.



## Consistency Tradeoffs

Quality of Output

Middle Consistency

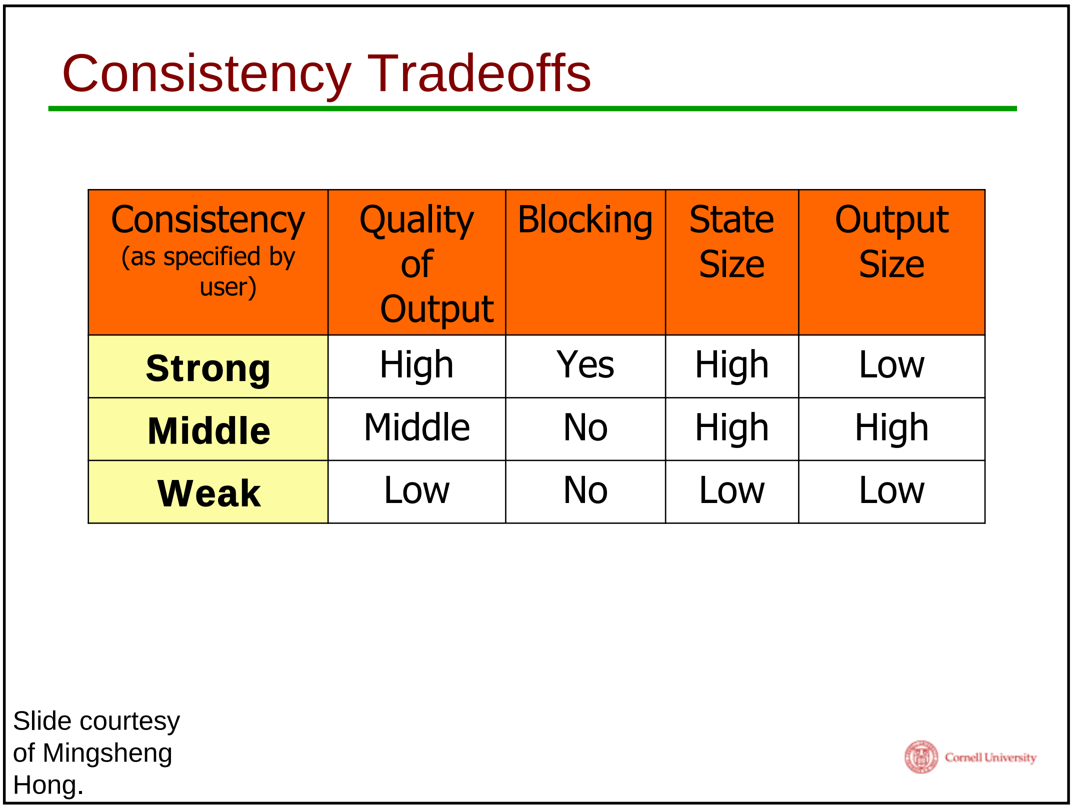
Strong Consistency

Non-Blocking

Output Size

Weak Consistency

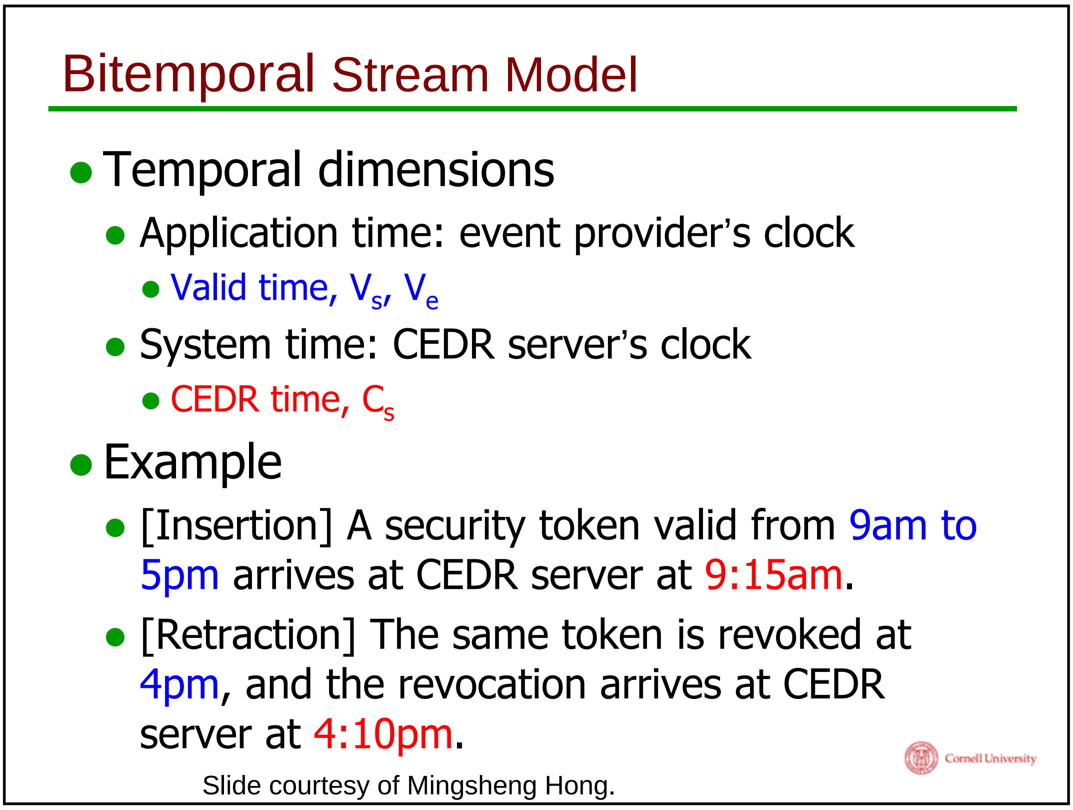
Slide courtesy of Mingsheng Hong.



## Consistency Tradeoffs

| Consistency (as specified by user) | Quality of Output | Blocking | State Size | Output Size |
|---|---|---|---|---|
| **Strong** | High | Yes | High | Low |
| **Middle** | Middle | No | High | High |
| **Weak** | Low | No | Low | Low |

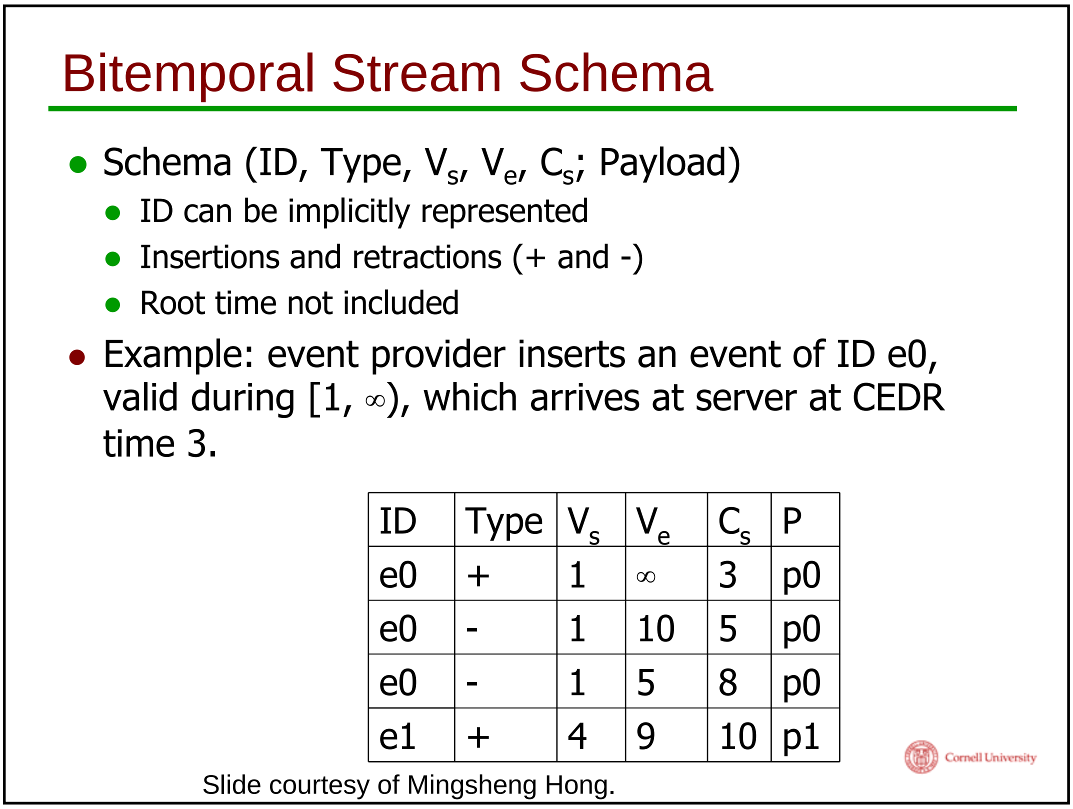
Slide courtesy of Mingsheng Hong.



## Bitemporal Stream Model

- Temporal dimensions
  - Application time: event provider's clock
    - Valid time, $V_s$, $V_e$
  - System time: CEDR server's clock
    - CEDR time, $C_s$
- Example
  - [Insertion] A security token valid from 9am to 5pm arrives at CEDR server at 9:15am.
  - [Retraction] The same token is revoked at 4pm, and the revocation arrives at CEDR server at 4:10pm.

Slide courtesy of Mingsheng Hong.

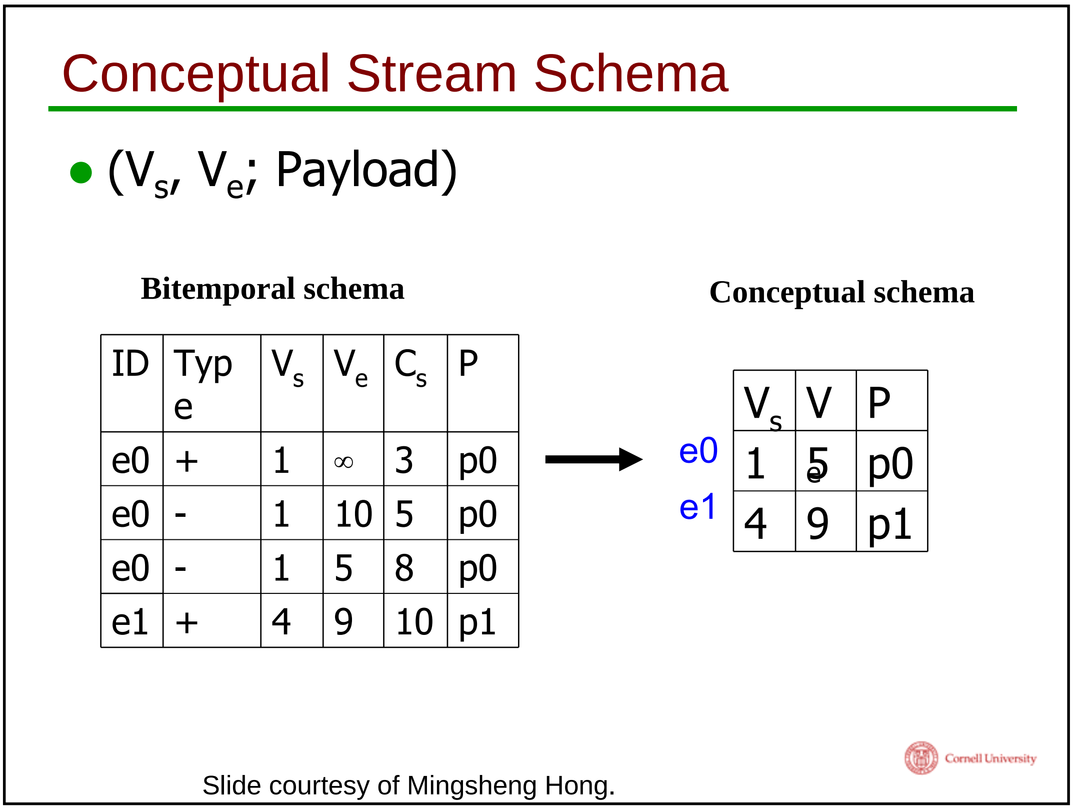


## Bitemporal Stream Schema

- Schema (ID, Type, $V_s$, $V_e$, $C_s$; Payload)
  - ID can be implicitly represented
  - Insertions and retractions (+ and -)
  - Root time not included
- Example: event provider inserts an event of ID e0, valid during [1, ∞), which arrives at server at CEDR time 3.

| ID | Type | $V_s$ | $V_e$ | $C_s$ | P |
|---|---|---|---|---|---|
| e0 | + | 1 | ∞ | 3 | p0 |
| e0 | - | 1 | 10 | 5 | p0 |
| e0 | - | 1 | 5 | 8 | p0 |
| e1 | + | 4 | 9 | 10 | p1 |

Slide courtesy of Mingsheng Hong.



## Conceptual Stream Schema

- ($V_s$, $V_e$; Payload)

**Bitemporal schema**

| ID | Typ<br>e | $V_s$ | $V_e$ | $C_s$ | P |
|---|---|---|---|---|---|
| e0 | + | 1 | ∞ | 3 | p0 |
| e0 | - | 1 | 10 | 5 | p0 |
| e0 | - | 1 | 5 | 8 | p0 |
| e1 | + | 4 | 9 | 10 | p1 |

→

**Conceptual schema**

| | $V_s$ | V | P |
|---|---|---|---|
| e0 | 1 | e5 | p0 |
| e1 | 4 | 9 | p1 |

Slide courtesy of Mingsheng Hong.

## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
  - STREAM and CQL
  - Cayuga
- Fault tolerance
- New operators
- A Case Study

## Continuous Query Language – CQL

SQL with:

- Streams
- Windows
- New semantics (stream)
  - Three relation-to-stream operators: Istream, Dstream Rstream
- Sampling

Slide based on material from Jennifer Widom.

## CQL: Stream

- A stream S is a (possibly infinite) bag (multiset) of elements (s,t), where s is a tuple belonging to the schema of S and t in T is the timestamp of the element.
- Base stream versus derived stream

Slide based on material from Jennifer Widom.

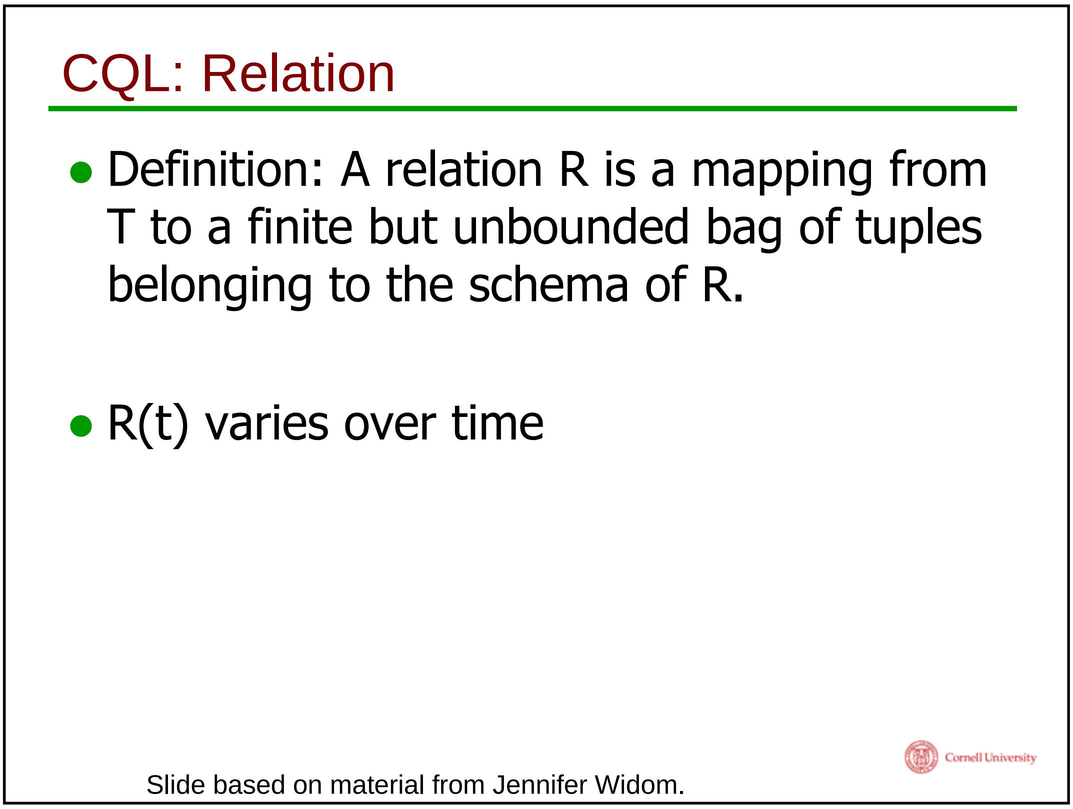
## CQL and Linear Road Examples

Simplified Linear Road Setup:

- A single input stream: The stream of positions and speeds of vehicles
- A single continuous query computing the tolls
- A single output toll stream: **PosSpeedStr(vehicleId,speed,xPos,dir,hwy)**

- vehicleId: vehicle
- speed: speed in MPH
- xPos: Position of the vehicle within the highway in feet
- dir: direction (east or west)
- hwy: highway number

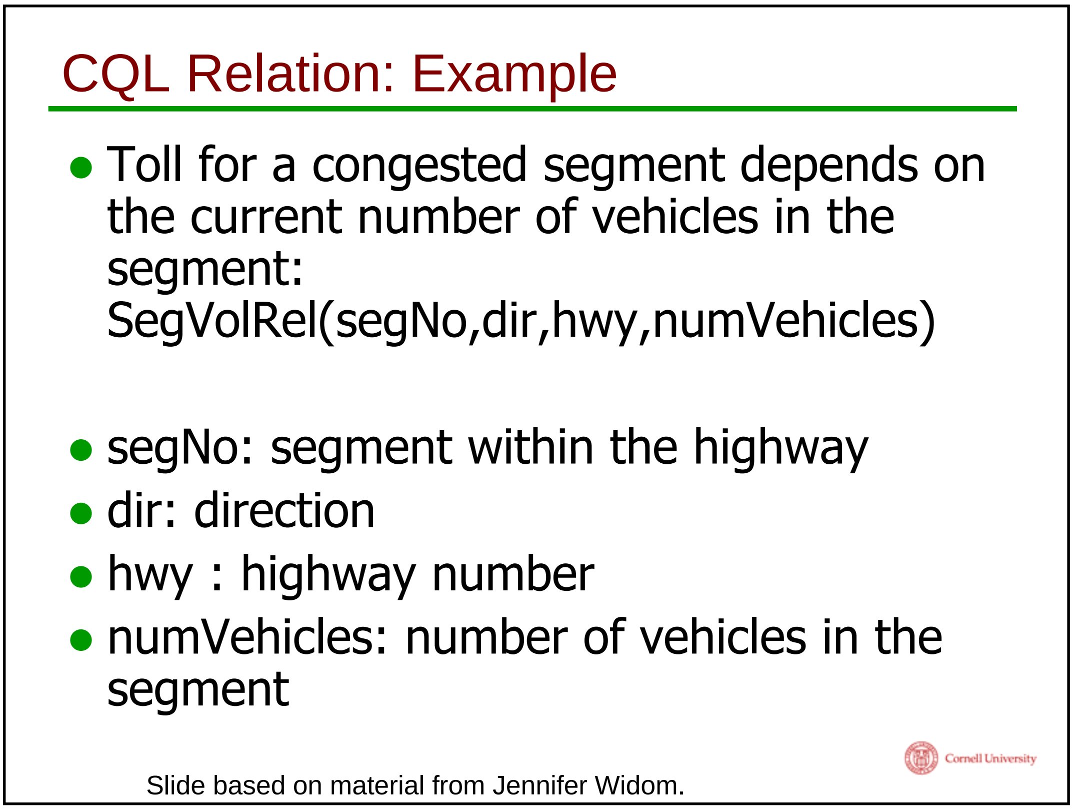
Slide based on material from Jennifer Widom.



## CQL: Relation

- Definition: A relation R is a mapping from T to a finite but unbounded bag of tuples belonging to the schema of R.
- R(t) varies over time

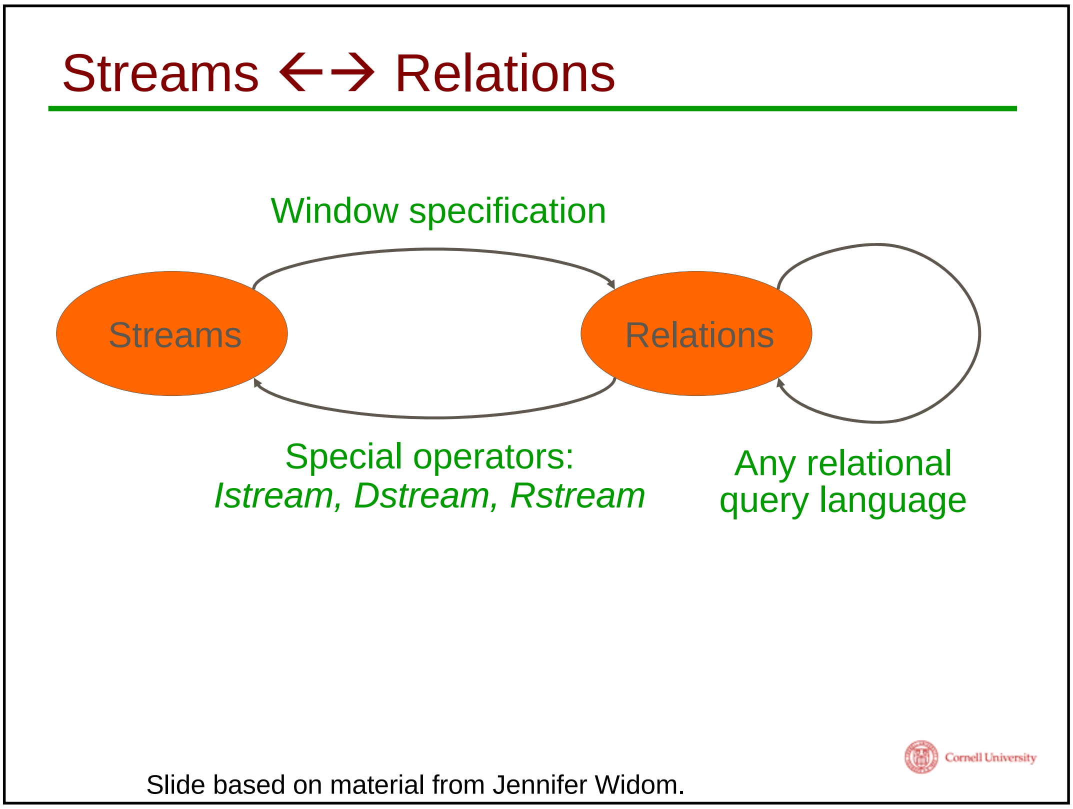
Slide based on material from Jennifer Widom.



## CQL Relation: Example

- Toll for a congested segment depends on the current number of vehicles in the segment: SegVolRel(segNo,dir,hwy,numVehicles)
- segNo: segment within the highway
- dir: direction
- hwy : highway number
- numVehicles: number of vehicles in the segment

Slide based on material from Jennifer Widom.



## Streams ←→ Relations

Window specification

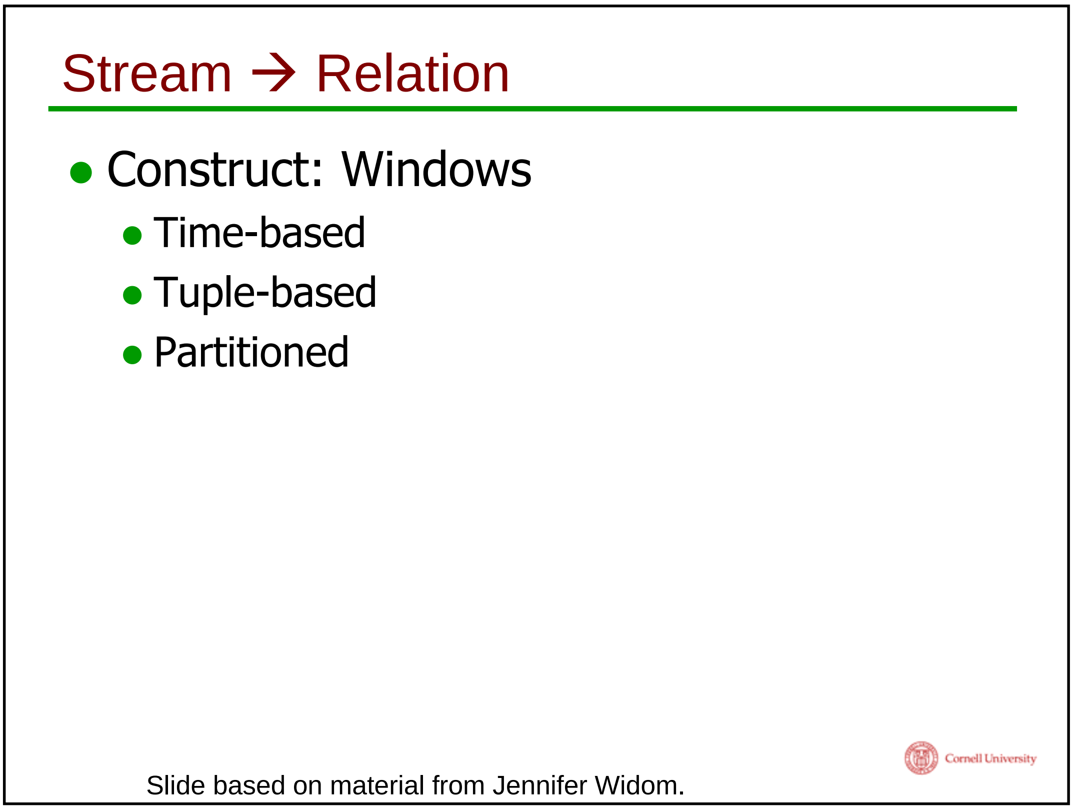
Streams

Relations

Special operators:
*Istream, Dstream, Rstream*

Any relational
query language

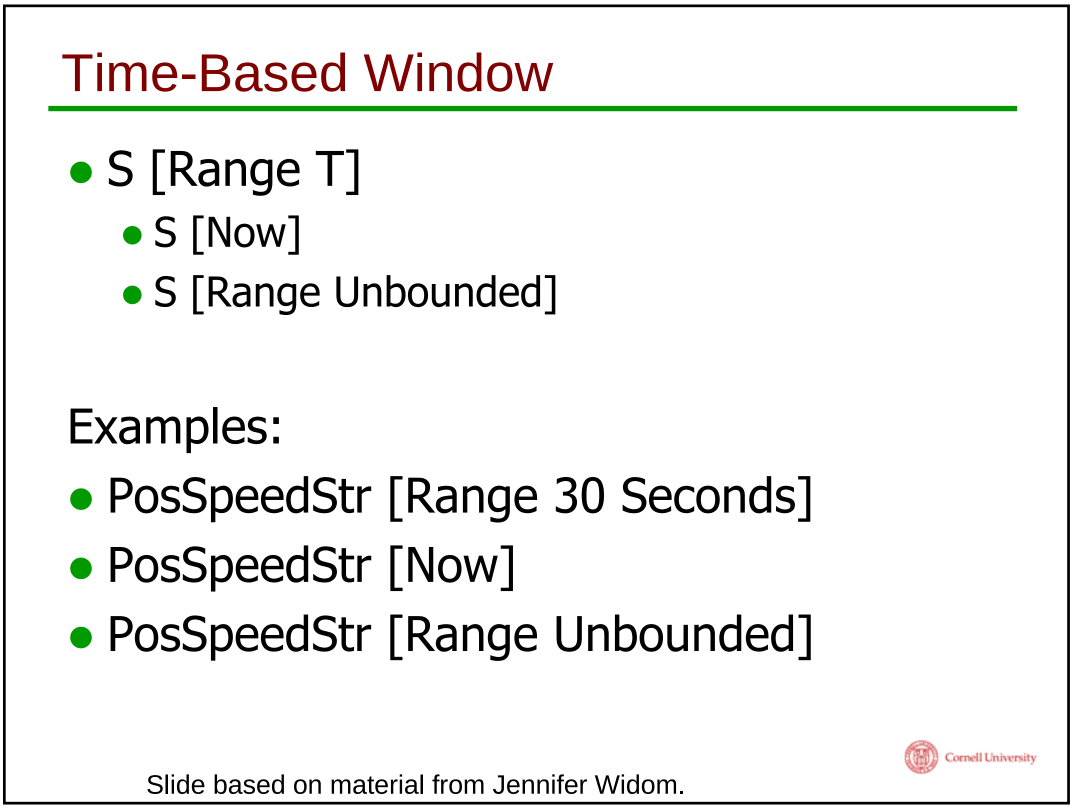
Slide based on material from Jennifer Widom.



## Stream → Relation

- Construct: Windows
  - Time-based
  - Tuple-based
  - Partitioned

Slide based on material from Jennifer Widom.



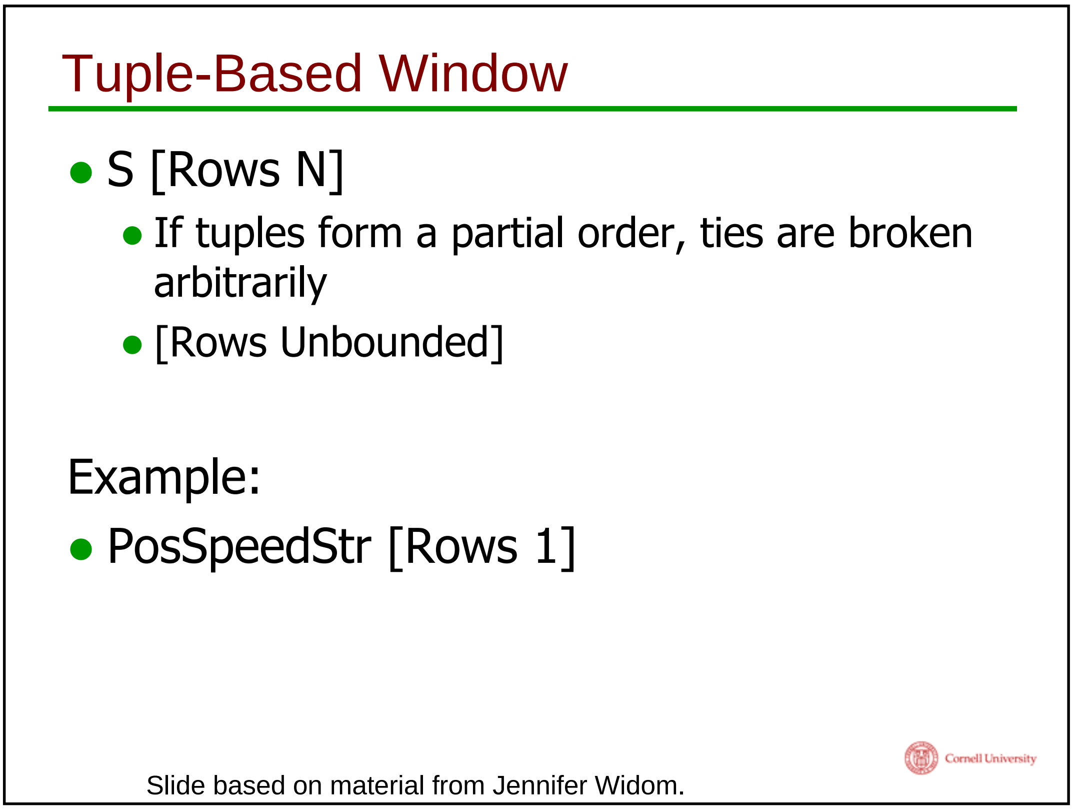
## Time-Based Window

- S [Range T]
  - S [Now]
  - S [Range Unbounded]

Examples:

- PosSpeedStr [Range 30 Seconds]
- PosSpeedStr [Now]
- PosSpeedStr [Range Unbounded]

Slide based on material from Jennifer Widom.



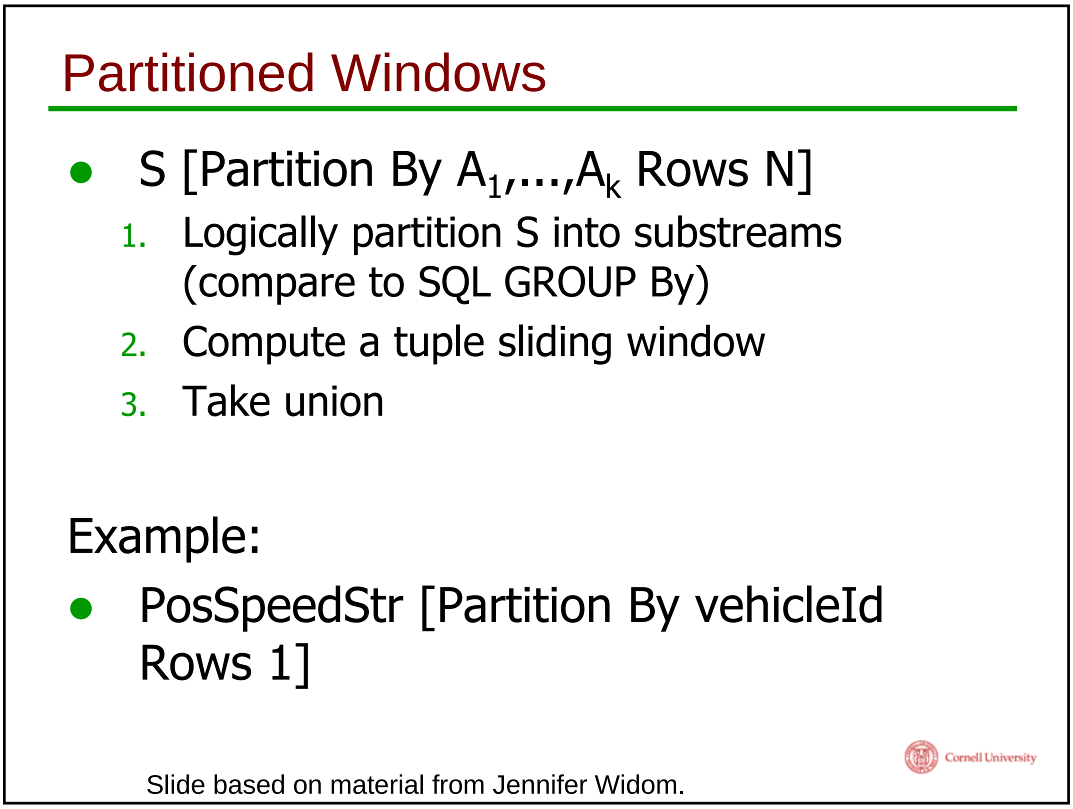
## Tuple-Based Window

- S [Rows N]
  - If tuples form a partial order, ties are broken arbitrarily
  - [Rows Unbounded]

Example:

- PosSpeedStr [Rows 1]

Slide based on material from Jennifer Widom.



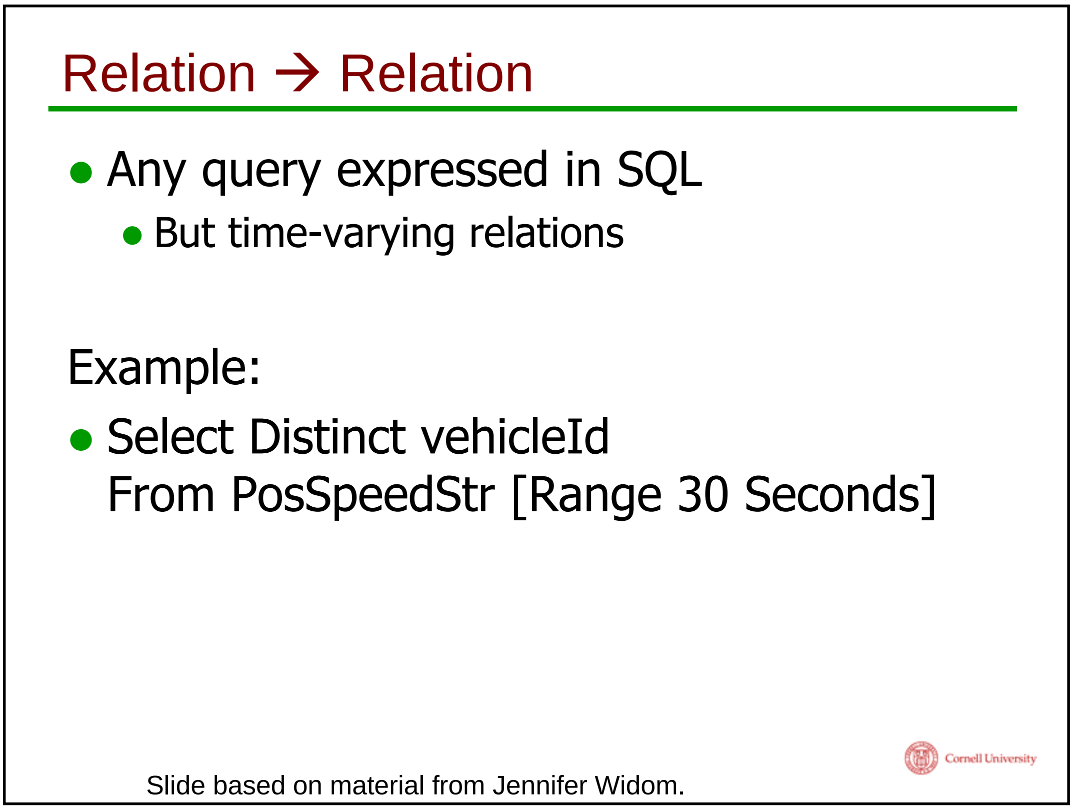
## Partitioned Windows

- S [Partition By $A_1,\ldots,A_k$ Rows N]
  1. Logically partition S into substreams (compare to SQL GROUP By)
  2. Compute a tuple sliding window
  3. Take union

Example:

- PosSpeedStr [Partition By vehicleId Rows 1]

Slide based on material from Jennifer Widom.



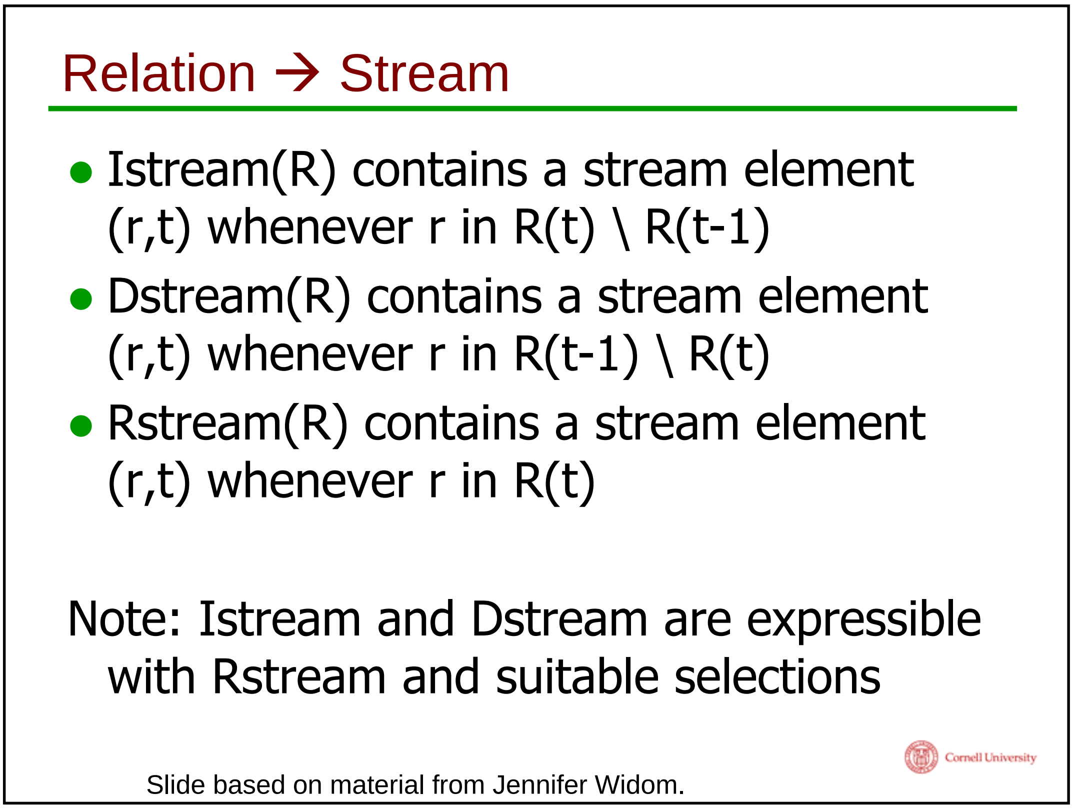
## Relation → Relation

- Any query expressed in SQL
  - But time-varying relations

Example:

- Select Distinct vehicleId
  From PosSpeedStr [Range 30 Seconds]

Slide based on material from Jennifer Widom.

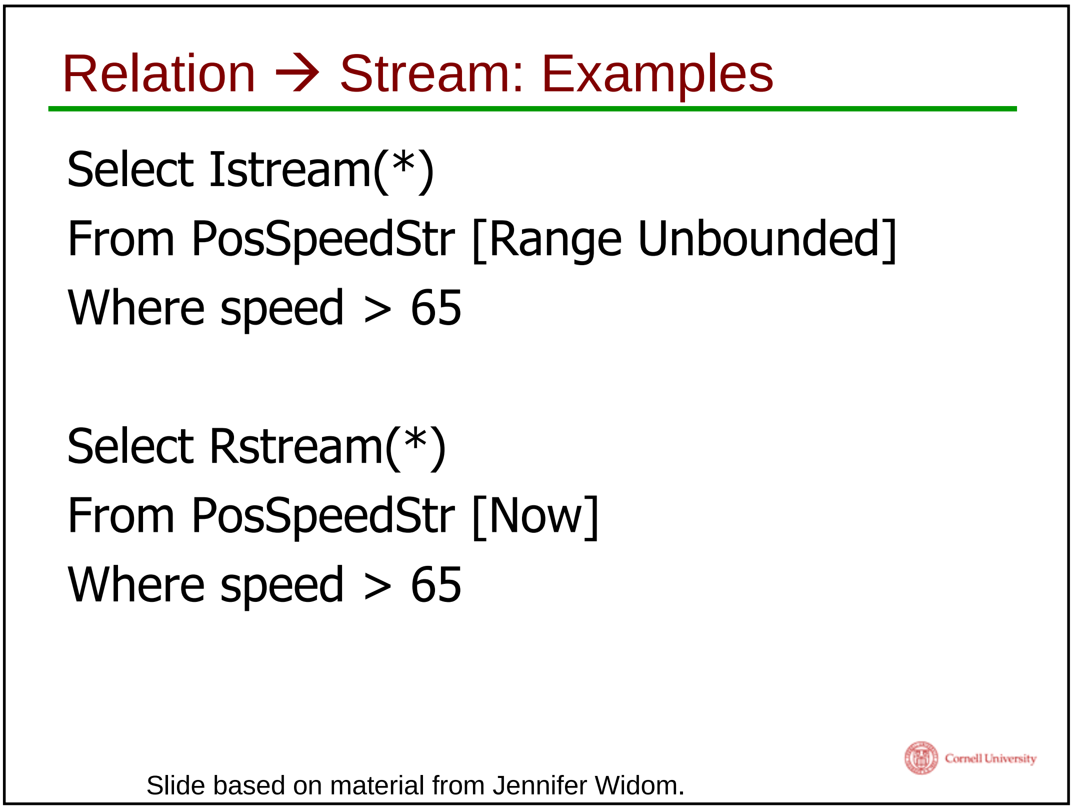


## Relation → Stream

- Istream(R) contains a stream element (r,t) whenever r in R(t) \ R(t-1)
- Dstream(R) contains a stream element (r,t) whenever r in R(t-1) \ R(t)
- Rstream(R) contains a stream element (r,t) whenever r in R(t)

Note: Istream and Dstream are expressible with Rstream and suitable selections

Slide based on material from Jennifer Widom.



## Relation → Stream: Examples

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

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

Slide based on material from Jennifer Widom.

## Some Equivalences

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

Slide based on material from Jennifer Widom.

## Some Equivalences (Contd.)

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

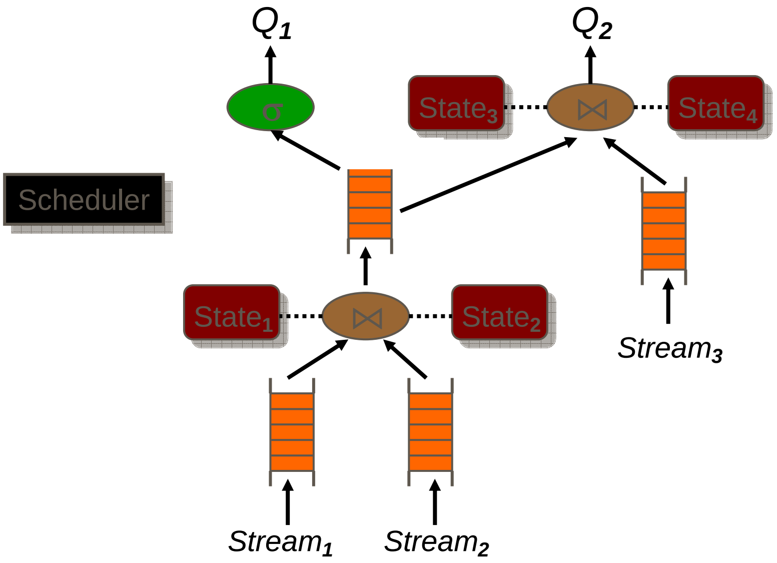
Slide based on material from Jennifer Widom.

## Query Execution

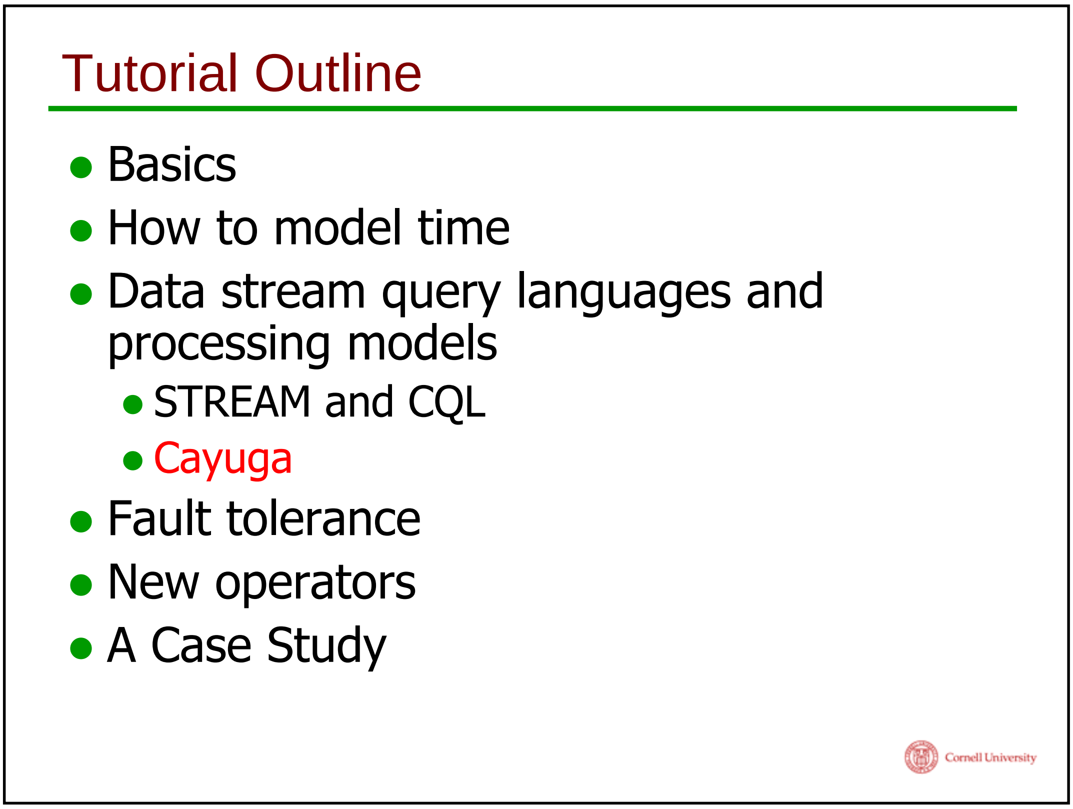
- When a continuous query is registered, generate a query execution plan
  - New plan merged with existing plans
  - Users can also create & manipulate plans directly
- Plans composed of three main components:
  - Operators
  - Queues (input and inter-operator)
  - State (windows, operators requiring history)
- Global scheduler for plan execution

Slide based on material from Jennifer Widom.

## Simple Query Plan



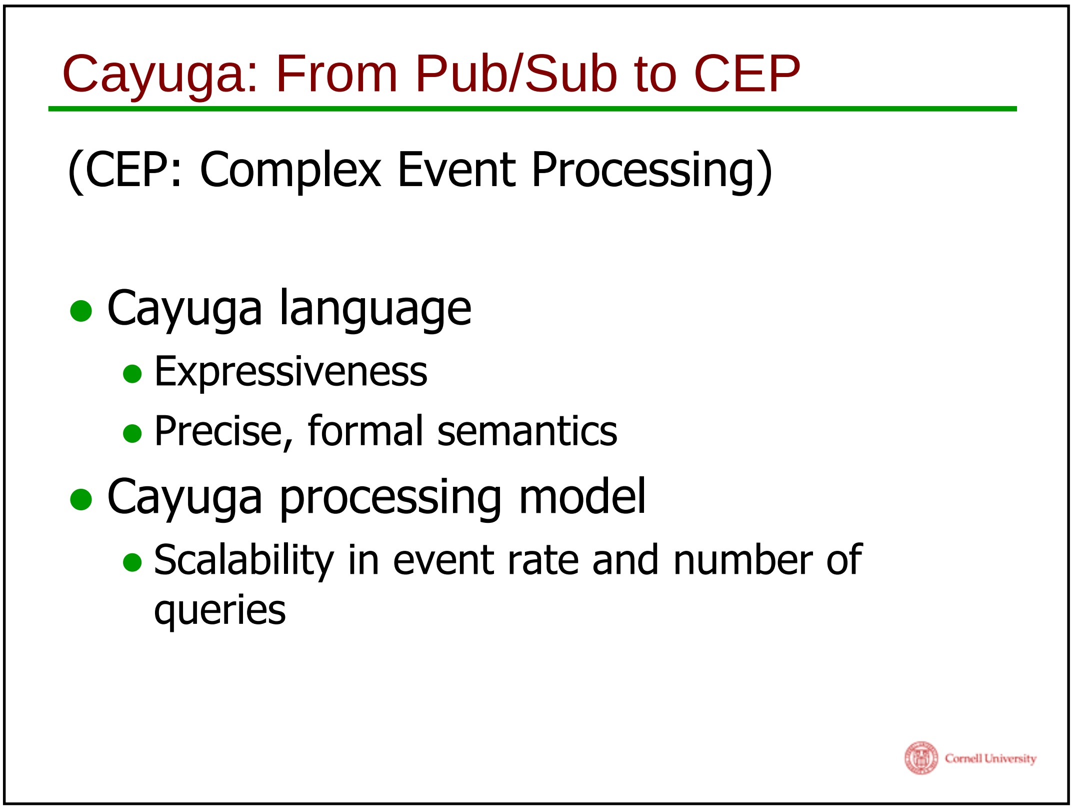
Slide courtesy of Jennifer Widom.



## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
  - STREAM and CQL
  - Cayuga
- Fault tolerance
- New operators
- A Case Study

Cornell University

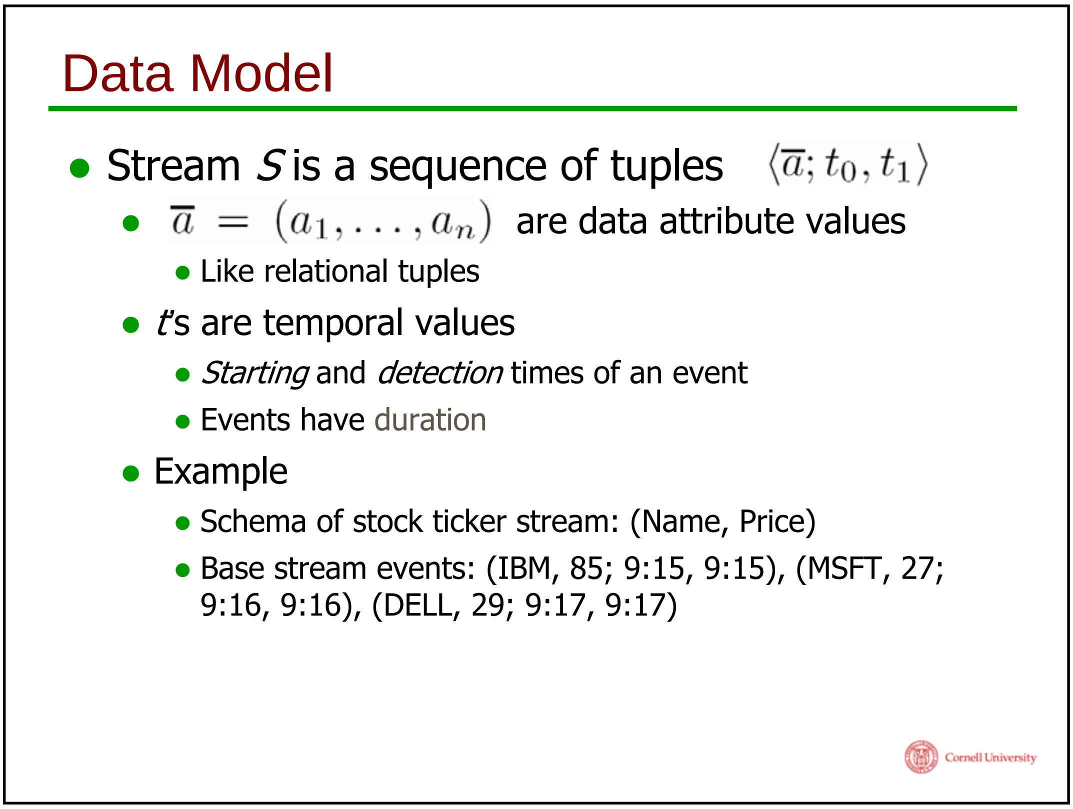


## Cayuga: From Pub/Sub to CEP

(CEP: Complex Event Processing)

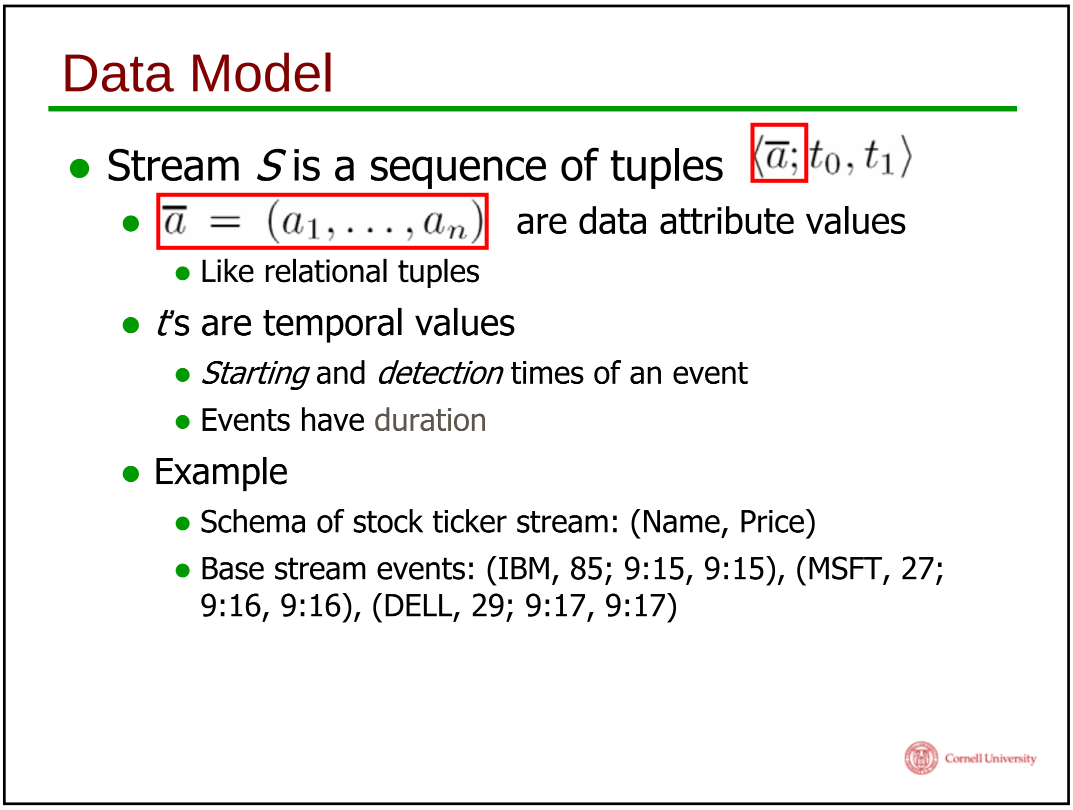
- Cayuga language
  - Expressiveness
  - Precise, formal semantics
- Cayuga processing model
  - Scalability in event rate and number of queries

Cornell University



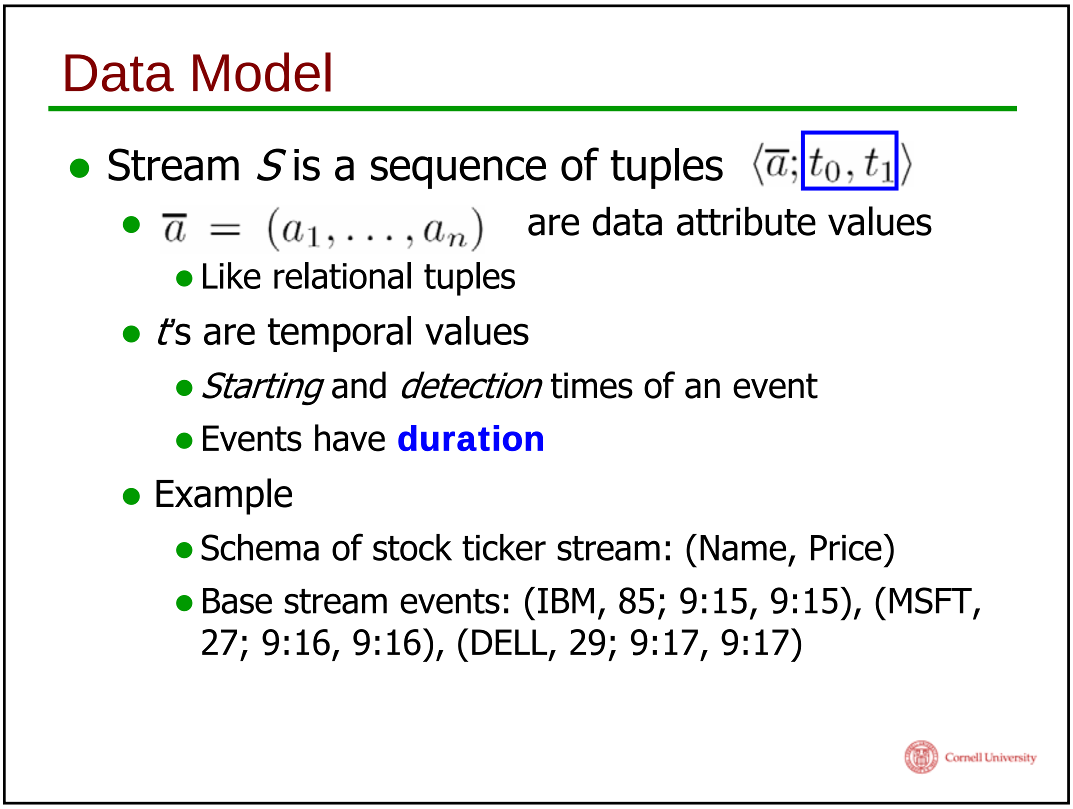
## Data Model

- Stream *S* is a sequence of tuples $\langle \overline{a}; t_0, t_1 \rangle$
  - $\overline{a} = (a_1, \ldots, a_n)$ are data attribute values
    - Like relational tuples
  - *t*'s are temporal values
    - *Starting* and *detection* times of an event
    - Events have duration
  - Example
    - Schema of stock ticker stream: (Name, Price)
    - Base stream events: (IBM, 85; 9:15, 9:15), (MSFT, 27; 9:16, 9:16), (DELL, 29; 9:17, 9:17)



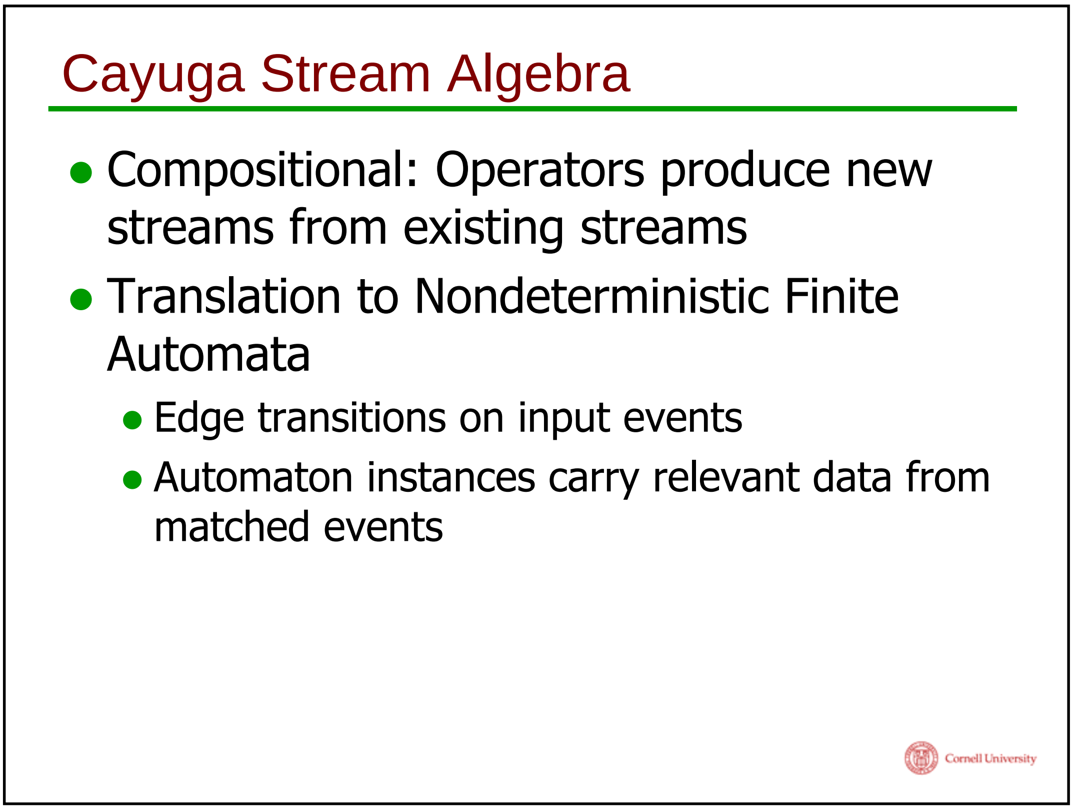
## Data Model

- Stream *S* is a sequence of tuples $\boxed{\langle \overline{a};} t_0, t_1 \rangle$
  - $\boxed{\overline{a} = (a_1, \ldots, a_n)}$ are data attribute values
    - Like relational tuples
  - *t*'s are temporal values
    - *Starting* and *detection* times of an event
    - Events have duration
  - Example
    - Schema of stock ticker stream: (Name, Price)
    - Base stream events: (IBM, 85; 9:15, 9:15), (MSFT, 27; 9:16, 9:16), (DELL, 29; 9:17, 9:17)



## Data Model

- Stream *S* is a sequence of tuples $\langle \overline{a}; \boxed{t_0, t_1} \rangle$
  - $\overline{a} = (a_1, \ldots, a_n)$ are data attribute values
    - Like relational tuples
  - *t*'s are temporal values
    - *Starting* and *detection* times of an event
    - Events have **duration**
  - Example
    - Schema of stock ticker stream: (Name, Price)
    - Base stream events: (IBM, 85; 9:15, 9:15), (MSFT, 27; 9:16, 9:16), (DELL, 29; 9:17, 9:17)

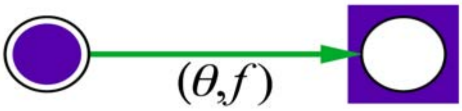


## Cayuga Stream Algebra

- Compositional: Operators produce new streams from existing streams
- Translation to Nondeterministic Finite Automata
  - Edge transitions on input events
  - Automaton instances carry relevant data from matched events

## Operators

- Relational operators (on non-temporal attributes)
  - Selection $\sigma_\theta$
  - Projection $\pi_X$
  - Renaming $\rho_f$
  - Union
- Together these give standard pub/sub



Automaton for $\rho_f \circ \sigma_\theta(S)$

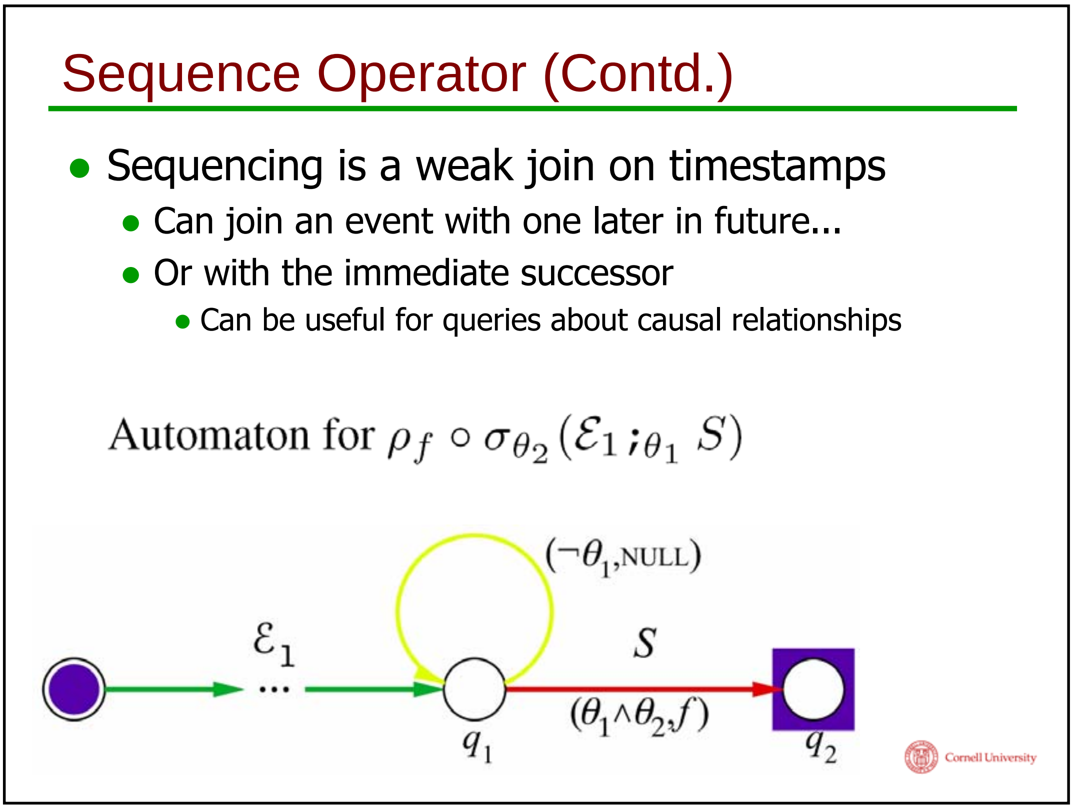
## Sequence Operator

- *Sequence* operator $S_1;_\theta S_2$
- After an event from $S_1$ is detected, match the first event from $S_2$ that satisfies the condition
- Examples
  - IBM price increases by at least $1 in two consecutive sales:

$$\sigma_{p2>p1+1}(\sigma_{n1=IBM}(S_1);_{n2=IBM} S_2)$$

  - Find a stock whose price stays constant in two consecutive sales:

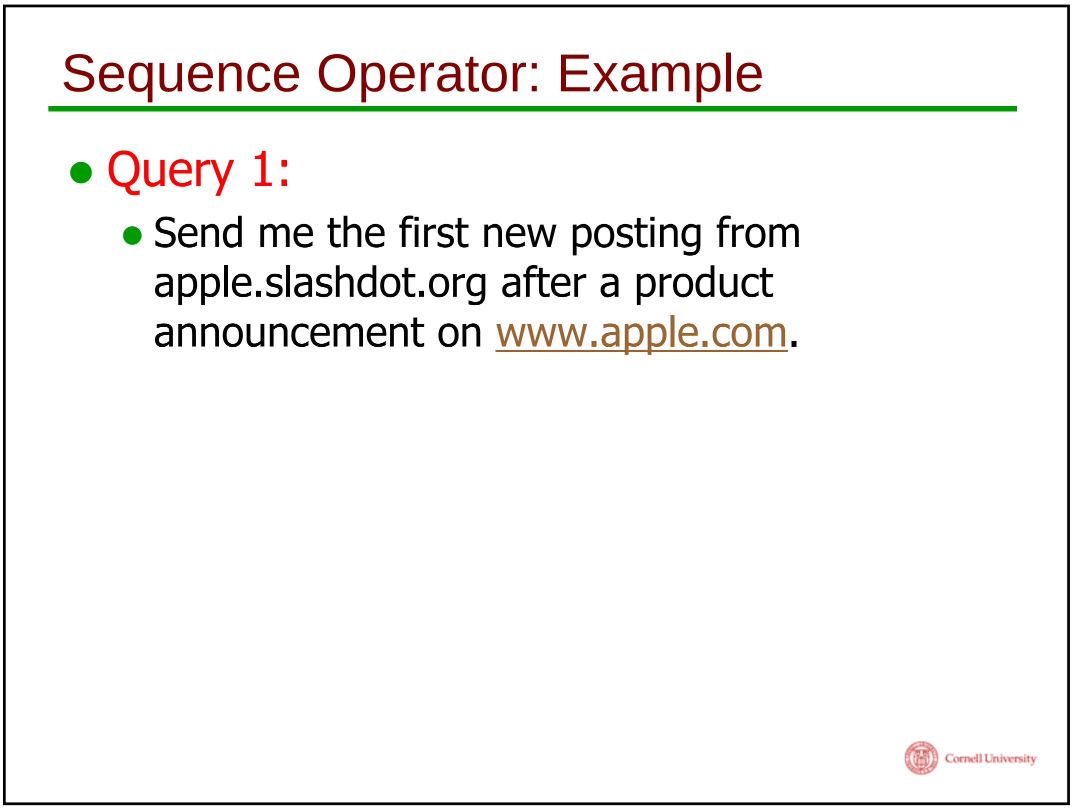
$$\sigma_{p1=p2}(S_1;_{n1=n2} S_2)$$



## Sequence Operator (Contd.)

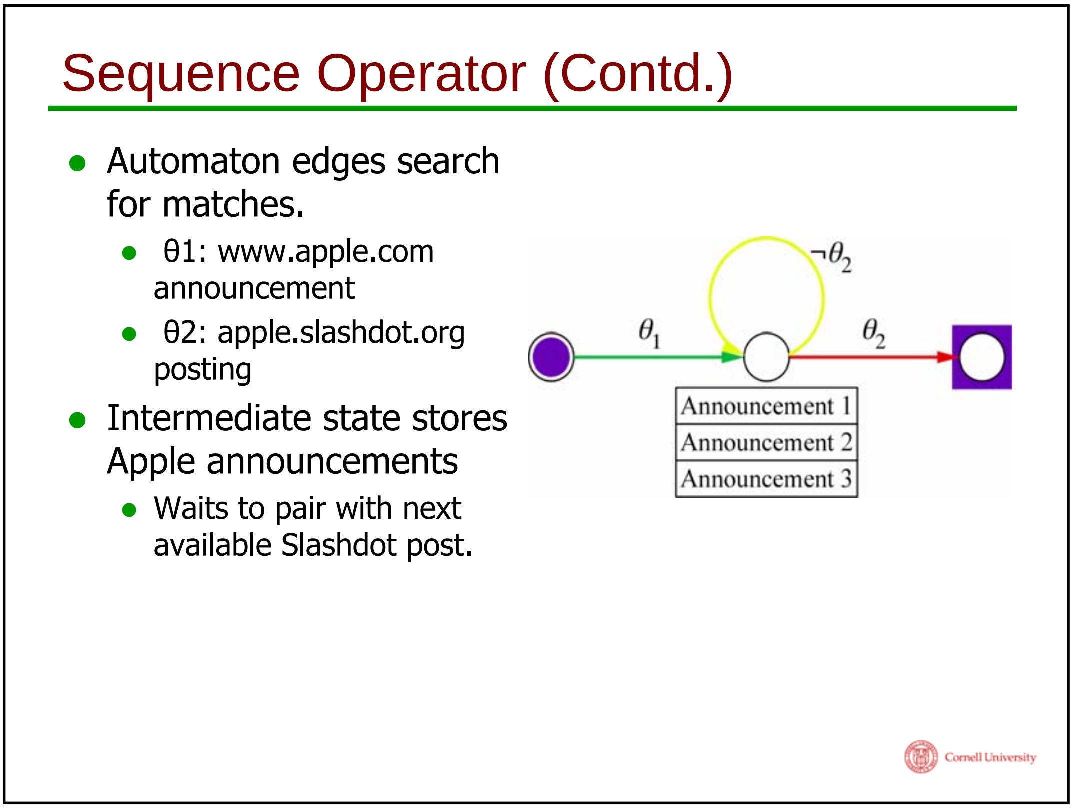
- Sequencing is a weak join on timestamps
  - Can join an event with one later in future...
  - Or with the immediate successor
    - Can be useful for queries about causal relationships

Automaton for $\rho_f \circ \sigma_{\theta_2}(\mathcal{E}_1 ;_{\theta_1} S)$



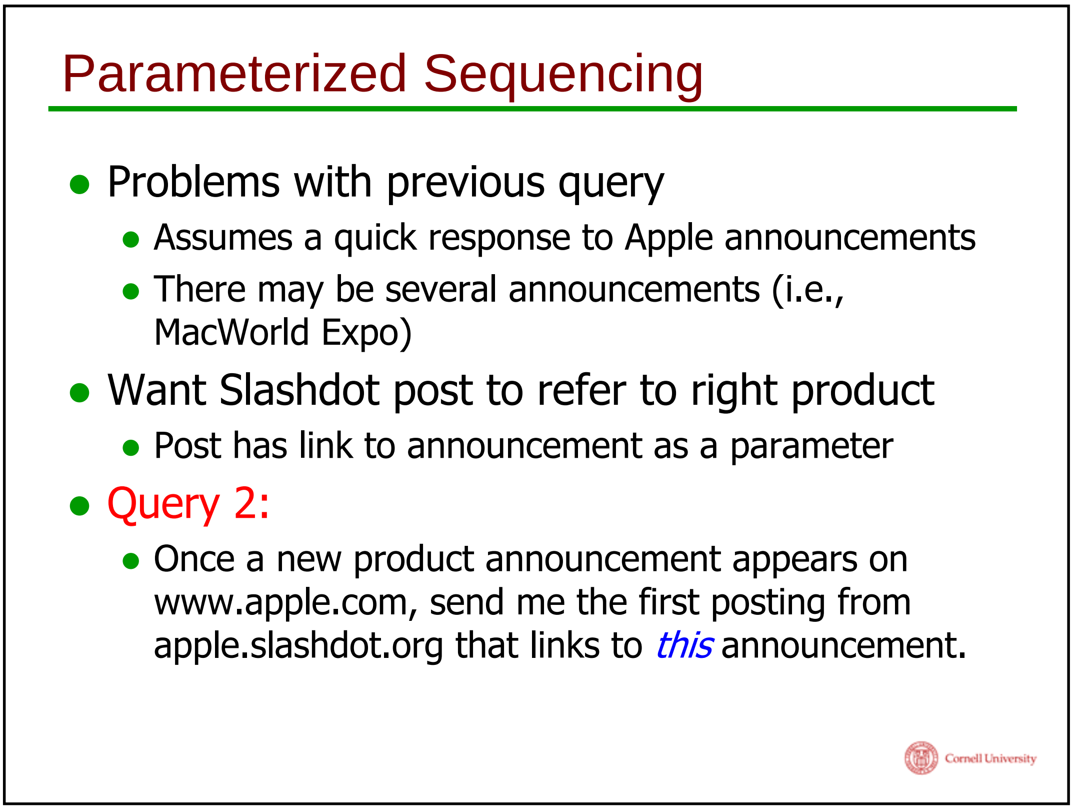
## Sequence Operator: Example

- Query 1:
  - Send me the first new posting from apple.slashdot.org after a product announcement on www.apple.com.



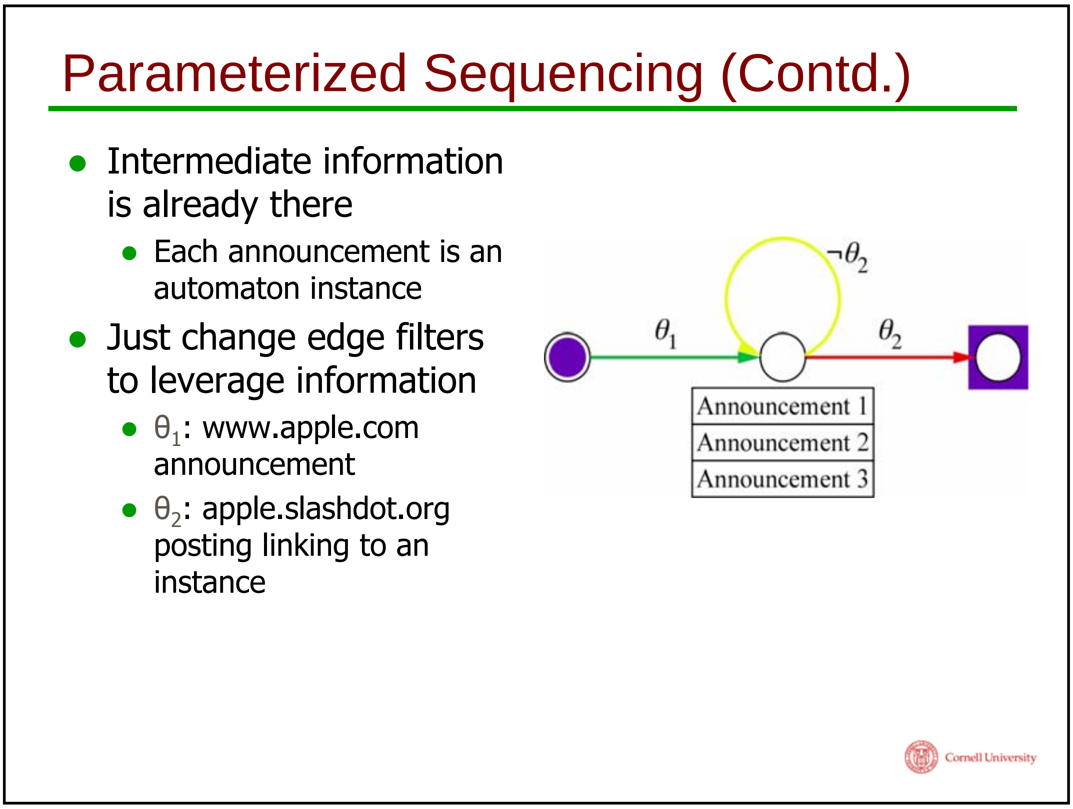
## Sequence Operator (Contd.)

- Automaton edges search for matches.
  - θ1: www.apple.com announcement
  - θ2: apple.slashdot.org posting
- Intermediate state stores Apple announcements
  - Waits to pair with next available Slashdot post.



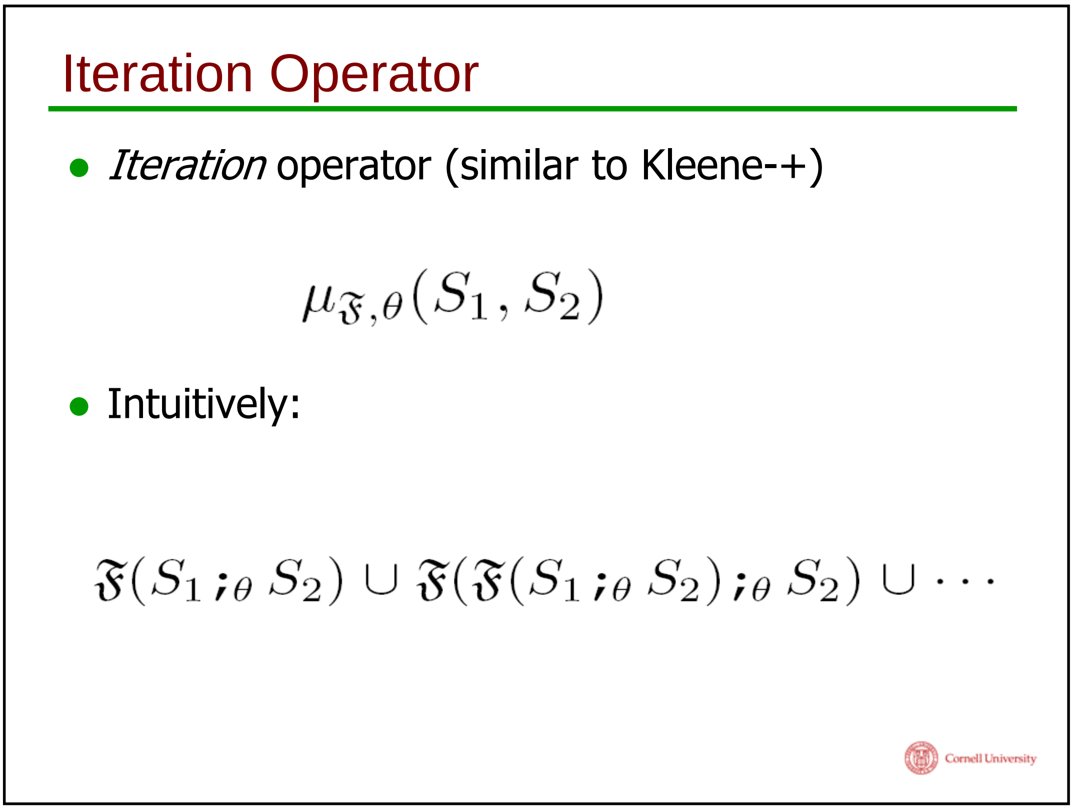
## Parameterized Sequencing

- Problems with previous query
  - Assumes a quick response to Apple announcements
  - There may be several announcements (i.e., MacWorld Expo)
- Want Slashdot post to refer to right product
  - Post has link to announcement as a parameter
- Query 2:
  - Once a new product announcement appears on www.apple.com, send me the first posting from apple.slashdot.org that links to *this* announcement.



## Parameterized Sequencing (Contd.)

- Intermediate information is already there
  - Each announcement is an automaton instance
- Just change edge filters to leverage information
  - $\theta_1$: www.apple.com announcement
  - $\theta_2$: apple.slashdot.org posting linking to an instance



## Iteration Operator

- *Iteration* operator (similar to Kleene-+)

$$\mu_{\mathfrak{F},\theta}(S_1, S_2)$$

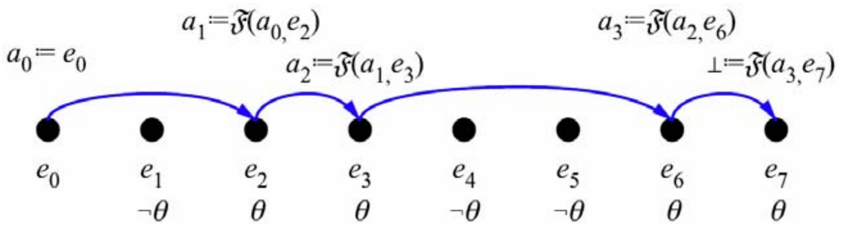
- Intuitively:

$$\mathfrak{F}(S_1 ;_\theta S_2) \cup \mathfrak{F}(\mathfrak{F}(S_1 ;_\theta S_2) ;_\theta S_2) \cup \cdots$$

## Iteration Example

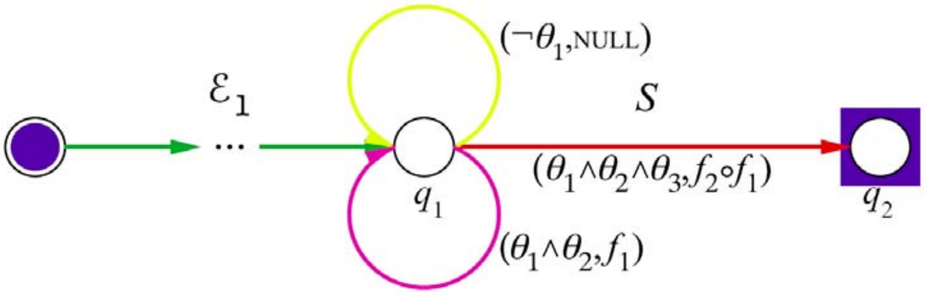
- IBM stock price monotonically increases

$$\mu_{\underbrace{\sigma_{p2>p2.last}}_{\mathfrak{F}},\underbrace{n2=IBM}_{\theta}}(\sigma_{n1=IBM}(S_1), S_2)$$



| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| Name | IBM | MSFT | IBM | IBM | DELL | MSFT | IBM | IBM |
| Price | 85 | 27 | 85.5 | 85.7 | 29 | 27.4 | 85.9 | 85.6 |

## Automaton for Iteration Operator



$(\neg\theta_1, \text{NULL})$
$\mathcal{E}_1$
$S$
...
$(\theta_1 \wedge \theta_2 \wedge \theta_3, f_2 \circ f_1)$
$q_1$
$q_2$
$(\theta_1 \wedge \theta_2, f_1)$

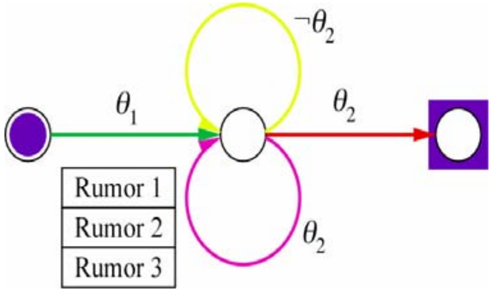
Automaton for $\rho_{f_2} \circ \sigma_{\theta_3}(\mu_{\rho_{f_1} \circ \sigma_{\theta_2}, \theta_1}(\mathcal{E}_1, S))$

## Iteration: Another Example

- Following the spread of crazy Apple rumors...

- Query 3:
  - Send me a sequence of Apple blog postings, in which the first posting is a rumor about an upcoming Apple product announcement, and each later posting is a reference (i.e., contains a direct quote from or a hypertext link to) to the previous.

## Implementing Iteration



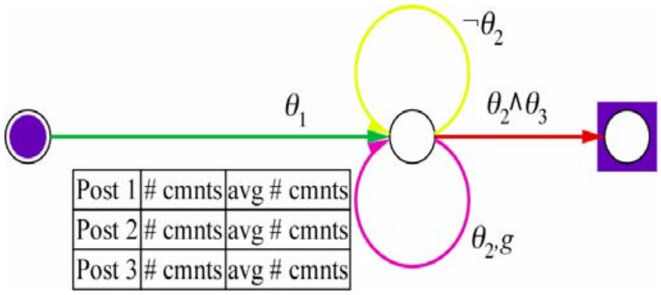
- Similar to parameters sequencing
  - $\theta_1$: Initial Apple rumor
  - $\theta_2$: Rumor that references the previous one
- Purple edge is a *rebind edge*
  - Updates instance information with latest rumor

# Aggregation

- Recall: Iteration also allows for aggregation.
  - Iterate over all posts of this type
  - Keep a running aggregate of some post attribute
    - e.g. current number of comments, average word count, etc...,
  - Implemented like normal aggregates
    - Need initializer, iterator, finalizer

- Query 4:
  - Send me an product review from apple.slashdot.org once it receives an above average number of user comments.

# Implementing Aggregation



| | | |
|---|---|---|
| Post 1 | # cmnts | avg # cmnts |
| Post 2 | # cmnts | avg # cmnts |
| Post 3 | # cmnts | avg # cmnts |

- Rebind edge performs the aggregation
  - *g* is attached to rebind edge to update values
- Note outgoing edge different from rebind edge
  - $\theta_3$: Above average number of comments

## Other Features

- Resubscription
  - Ability for one query to subscribe to the output of another (as a stream)
  - Significantly more expressive
- Extensibility
  - incorporate user-defined datatypes, data mining algorithms, predicates, aggregation functions, ...

## Example

- Notify me when
  1. for any stock, there is a a very large trade (volume > 10K);
  2. followed by a monotonic decrease in price for at least 10 minutes;
  3. the next quote on the same stock after this monotonic sequence is 5% above the previously seen (bottom) price.
- Intuition: Large sale, followed by price drop, followed by sudden upwards move

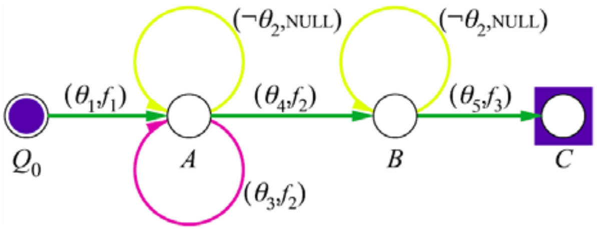
## Example

- Algebra expression: $\sigma_{\theta_5}(\sigma_{\theta_4}(\mu_{\sigma_{\theta_3},\theta_2}(S_1,S_2));_{\theta_2} S_3)$

$$
\begin{aligned}
S_1 &\equiv \rho_{f_1} \circ \pi_{\texttt{name,price}} \circ \sigma_{\theta_1}(S)\\
S_2 &\equiv \rho_{f_2} \circ \pi_{\texttt{name,price}}(S)\\
S_3 &\equiv \rho_{f_3} \circ \pi_{\texttt{name,price}}(S)
\end{aligned}
$$

$$
\begin{aligned}
\theta_1 &\equiv \texttt{vol} > 10,000\\
\theta_2 &\equiv \texttt{company} = \texttt{company.last}\\
\theta_3 &\equiv \theta_2 \wedge \texttt{minP} < \texttt{minP.last}\\
\theta_4 &\equiv \theta_3 \wedge \text{DUR} \geq 10\,\text{min}\\
\theta_5 &\equiv \theta_2 \wedge \texttt{price} > 1.05\,\texttt{minP}\\
f_1 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{maxP})\\
f_2 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{minP})\\
f_3 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{finalP})
\end{aligned}
$$

## Example

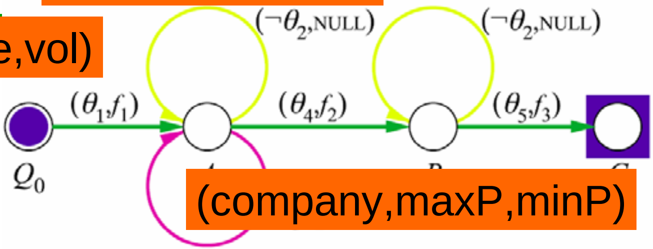


$$
\begin{aligned}
\theta_1 &\equiv \texttt{vol} > 10,000\\
\theta_2 &\equiv \texttt{company} = \texttt{company.last}\\
\theta_3 &\equiv \theta_2 \wedge \texttt{minP} < \texttt{minP.last}\\
\theta_4 &\equiv \theta_3 \wedge \text{DUR} \geq 10\,\text{min}\\
\theta_5 &\equiv \theta_2 \wedge \texttt{price} > 1.05\,\texttt{minP}\\
f_1 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{maxP})\\
f_2 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{minP})\\
f_3 &\equiv (\texttt{name},\texttt{price}) \mapsto (\texttt{company},\texttt{finalP})
\end{aligned}
$$

## Example

(company,maxP)

(name,price,vol)

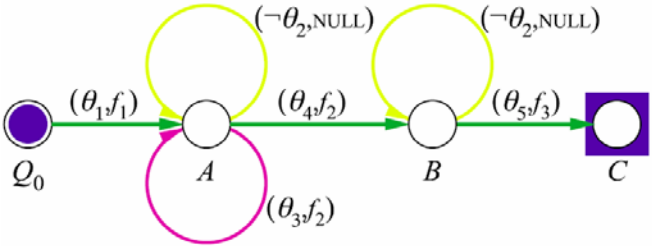


(company,maxP,minP)

(company,maxP,minP,finalP)

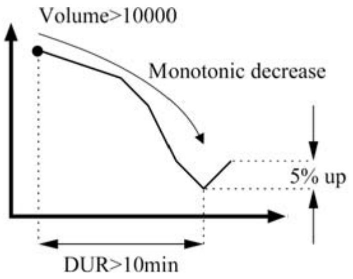
$$\begin{aligned}
\theta_1 &\equiv \texttt{vol} > 10,000 \\
\theta_2 &\equiv \texttt{company} = \texttt{company.last} \\
\theta_3 &\equiv \theta_2 \wedge \texttt{minP} < \texttt{minP.last} \\
\theta_4 &\equiv \theta_3 \wedge \text{DUR} \geq 10\,\text{min} \\
\theta_5 &\equiv \theta_2 \wedge \texttt{price} > 1.05\,\texttt{minP} \\
f_1 &\equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{maxP}) \\
f_2 &\equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{minP}) \\
f_3 &\equiv (\texttt{name}, \texttt{price}) \mapsto (\texttt{company}, \texttt{finalP})
\end{aligned}$$

## Example



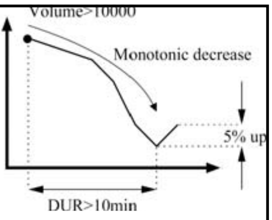
| **Input event**<br>$(\texttt{name}, \texttt{price}, \texttt{vol})$ | Instances at state $A$<br>$(\texttt{company}, \texttt{maxP}, \texttt{minP})$ | Instances at state $B$<br>$(\texttt{company}, \texttt{maxP}, \texttt{minP})$ | Instances at state $C$<br>$(\texttt{company}, \texttt{maxP}, \texttt{minP}, \texttt{finalP})$ |
|---|---|---|---|
| $e_1$ : $\langle$IBM, 90, 15000; 9:10; 9:10$\rangle$ | $I_1$ = $\langle$IBM, 90, NULL; 9:10; 9:10$\rangle$ | | |
| $e_2$ : $\langle$IBM, 85, 7000; 9:15; 9:15$\rangle$ | $I_1$ = $\langle$IBM, 90, 85; 9:10; 9:15$\rangle$ | | |
| $e_3$ : $\langle$Dell, 40, 11000; 9:17; 9:17$\rangle$ | $I_1$ = $\langle$IBM, 90, 85; 9:10; 9:15$\rangle$<br>$I_2$ = $\langle$Dell, 40, NULL; 9:17; 9:17$\rangle$ | | |
| $e_4$ : $\langle$IBM, 81, 8000; 9:21; 9:21$\rangle$ | $I_1$ = $\langle$IBM, 90, 81; 9:10; 9:21$\rangle$<br>$I_2$ = $\langle$Dell, 40, NULL; 9:17; 9:17$\rangle$ | $I_3$ = $\langle$IBM, 90, 81; 9:10; 9:21$\rangle$ | |
| $e_5$ : $\langle$MSFT, 25, 6000; 9:23; 9:23$\rangle$ | $I_1$ = $\langle$IBM, 90, 81; 9:10; 9:21$\rangle$<br>$I_2$ = $\langle$Dell, 40, NULL; 9:17; 9:17$\rangle$ | $I_3$ = $\langle$IBM, 90, 81; 9:10; 9:21$\rangle$ | |
| $e_6$ : $\langle$IBM, 91, 9000; 9:24; 9:24$\rangle$ | $I_2$ = $\langle$Dell, 40, NULL; 9:17; 9:17$\rangle$ | | $I_3$ = $\langle$IBM, 90, 81, 91; 9:10; 9:24$\rangle$ |

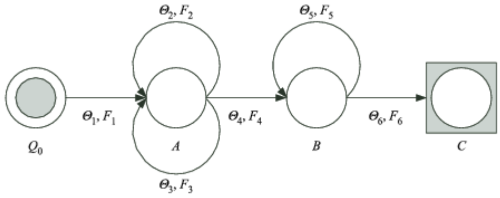
# Cayuga Query Language



```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{Price > 1.05*MinPrice}(
    FILTER{DUR > 10min}(
      (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
      FROM FILTER{Volume > 10000}(Stock))
        FOLD{$2.Name = $.Name, $2.Price < $.Price}
      Stock)
    NEXT{$2.Name = $1.Name}
    Stock)
```

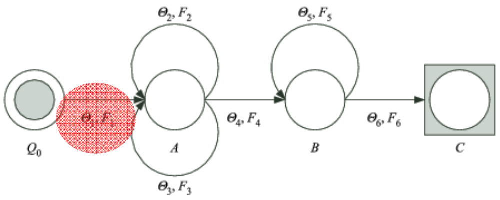
# Cayuga Automata





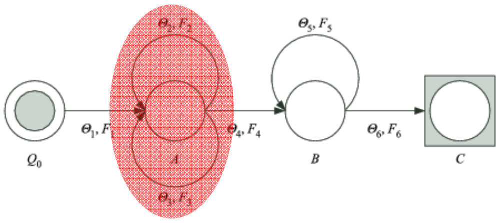
```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{Price > 1.05*MinPrice}(
    FILTER{DUR > 10min}(
      (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
      FROM FILTER{Volume > 10000}(Stock))
        FOLD{$2.Name = $.Name, $2.Price < $.Price}
      Stock)
    NEXT{$2.Name = $1.Name}
    Stock)
```

# Cayuga Automata



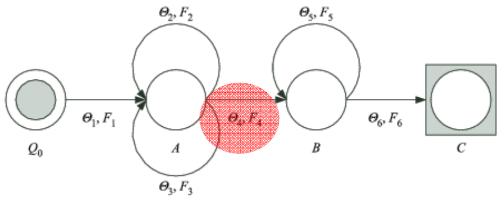
```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
   FILTER{Price > 1.05*MinPrice}(
      FILTER{DUR > 10min}(
        (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
        FROM FILTER{Volume > 10000}(Stock))
           FOLD{$2.Name = $.Name, $2.Price < $.Price}
        Stock)
      NEXT{$2.Name = $1.Name}
      Stock)
```

# Cayuga Automata



```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
   FILTER{Price > 1.05*MinPrice}(
      FILTER{DUR > 10min}(
        (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
        FROM FILTER{Volume > 10000}(Stock))
           FOLD{$2.Name = $.Name, $2.Price < $.Price}
        Stock)
      NEXT{$2.Name = $1.Name}
      Stock)
```

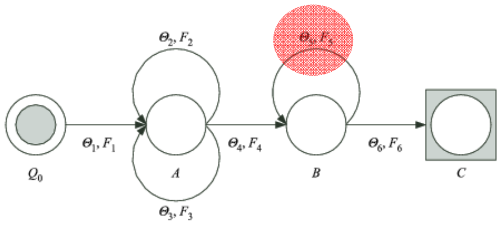
# Cayuga Automata



```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{Price > 1.05*MinPrice}(
    FILTER{DUR > 10min}(
     (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
     FROM FILTER{Volume > 10000}(Stock))
       FOLD{$2.Name = $.Name, $2.Price < $.Price}
     Stock)
    NEXT{$2.Name = $1.Name}
    Stock)
```

Cornell University

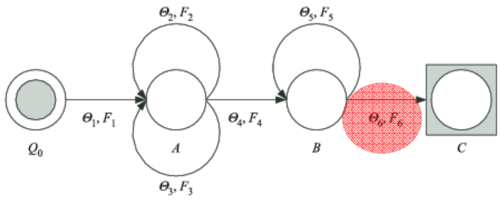
# Cayuga Automata



```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
  FILTER{Price > 1.05*MinPrice}(
    FILTER{DUR > 10min}(
     (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
     FROM FILTER{Volume > 10000}(Stock))
       FOLD{$2.Name = $.Name, $2.Price < $.Price}
     Stock)
    NEXT{$2.Name = $1.Name}
    Stock)
```

Cornell University

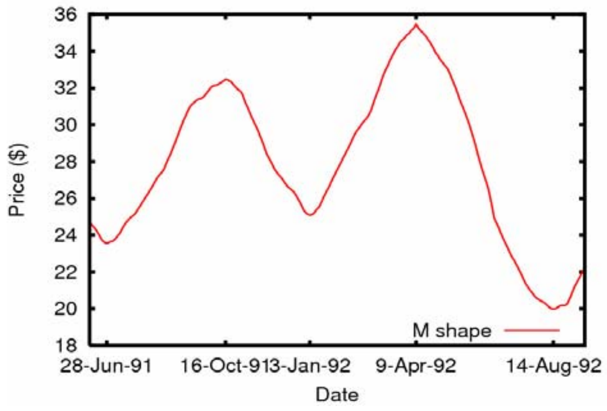
# Cayuga Automata

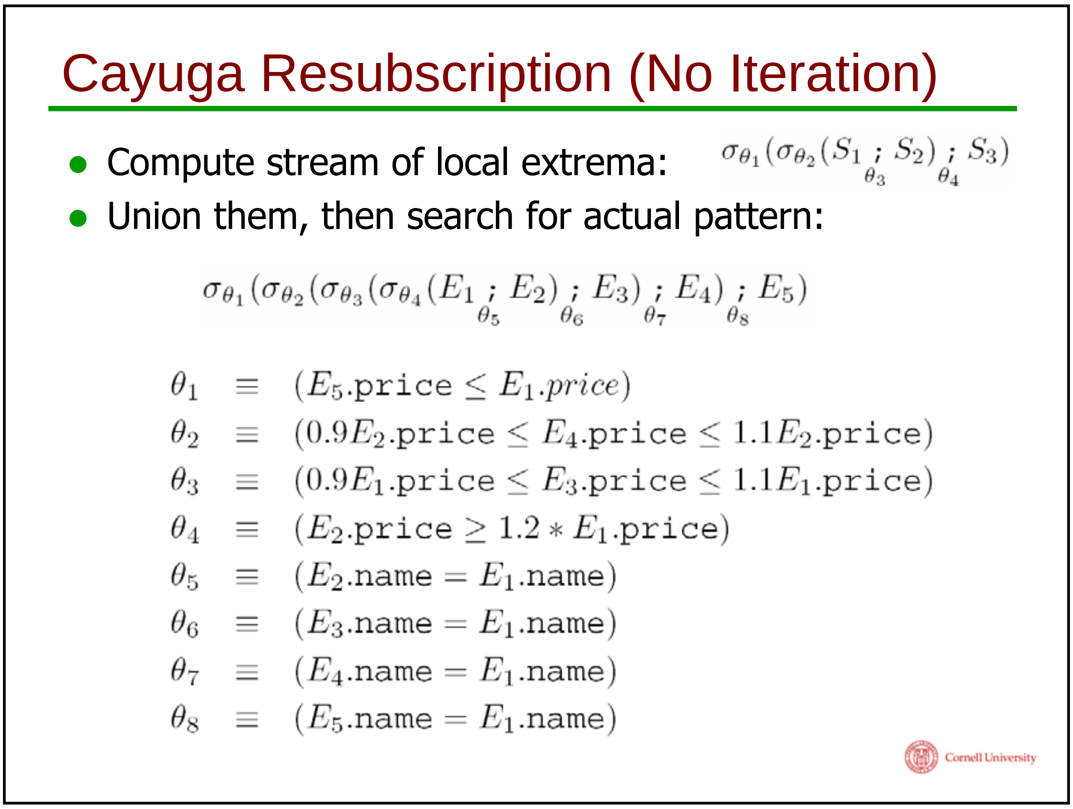


```
SELECT Name, MaxPrice, MinPrice, Price AS FinalPrice
FROM
   FILTER{Price > 1.05*MinPrice}(
      FILTER{DUR > 10min}(
        (SELECT Name, Price_1 AS MaxPrice, Price AS MinPrice
        FROM FILTER{Volume > 10000}(Stock))
           FOLD{$2.Name = $.Name, $2.Price < $.Price}
        Stock)
      NEXT{$2.Name = $1.Name}
      Stock)
```

# Example: Double-Top

- Double-Top query pattern



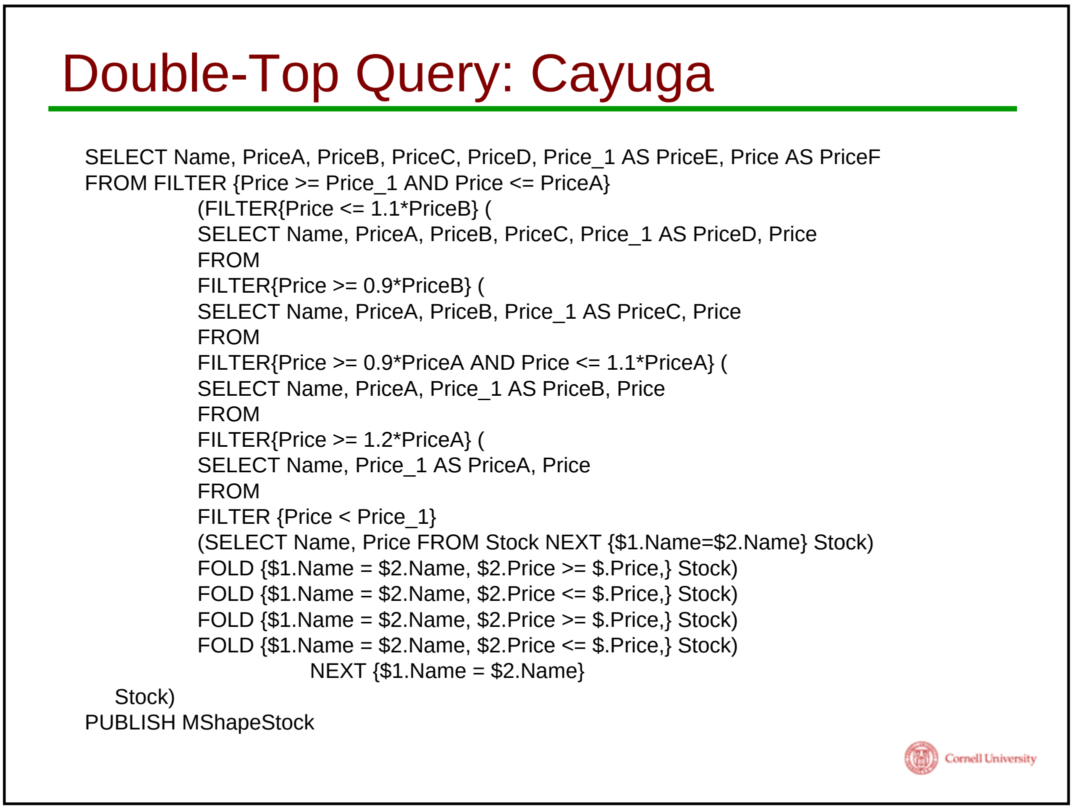


## Cayuga Resubscription (No Iteration)

- Compute stream of local extrema: $\sigma_{\theta_1}(\sigma_{\theta_2}(S_1 \underset{\theta_3}{;} S_2) \underset{\theta_4}{;} S_3)$
- Union them, then search for actual pattern:

$$\sigma_{\theta_1}(\sigma_{\theta_2}(\sigma_{\theta_3}(\sigma_{\theta_4}(E_1 \underset{\theta_5}{;} E_2) \underset{\theta_6}{;} E_3) \underset{\theta_7}{;} E_4) \underset{\theta_8}{;} E_5)$$

$$\begin{aligned}
\theta_1 &\equiv (E_5.\mathtt{price} \leq E_1.price) \\
\theta_2 &\equiv (0.9E_2.\mathtt{price} \leq E_4.\mathtt{price} \leq 1.1E_2.\mathtt{price}) \\
\theta_3 &\equiv (0.9E_1.\mathtt{price} \leq E_3.\mathtt{price} \leq 1.1E_1.\mathtt{price}) \\
\theta_4 &\equiv (E_2.\mathtt{price} \geq 1.2 * E_1.\mathtt{price}) \\
\theta_5 &\equiv (E_2.\mathtt{name} = E_1.\mathtt{name}) \\
\theta_6 &\equiv (E_3.\mathtt{name} = E_1.\mathtt{name}) \\
\theta_7 &\equiv (E_4.\mathtt{name} = E_1.\mathtt{name}) \\
\theta_8 &\equiv (E_5.\mathtt{name} = E_1.\mathtt{name})
\end{aligned}$$



## Double-Top Query: Cayuga

```
SELECT Name, PriceA, PriceB, PriceC, PriceD, Price_1 AS PriceE, Price AS PriceF
FROM FILTER {Price >= Price_1 AND Price <= PriceA}
          (FILTER{Price <= 1.1*PriceB} (
          SELECT Name, PriceA, PriceB, PriceC, Price_1 AS PriceD, Price
          FROM
          FILTER{Price >= 0.9*PriceB} (
          SELECT Name, PriceA, PriceB, Price_1 AS PriceC, Price
          FROM
          FILTER{Price >= 0.9*PriceA AND Price <= 1.1*PriceA} (
          SELECT Name, PriceA, Price_1 AS PriceB, Price
          FROM
          FILTER{Price >= 1.2*PriceA} (
          SELECT Name, Price_1 AS PriceA, Price
          FROM
          FILTER {Price < Price_1}
          (SELECT Name, Price FROM Stock NEXT {$1.Name=$2.Name} Stock)
          FOLD {$1.Name = $2.Name, $2.Price >= $.Price,} Stock)
          FOLD {$1.Name = $2.Name, $2.Price <= $.Price,} Stock)
          FOLD {$1.Name = $2.Name, $2.Price >= $.Price,} Stock)
          FOLD {$1.Name = $2.Name, $2.Price <= $.Price,} Stock)
                    NEXT {$1.Name = $2.Name}
  Stock)
PUBLISH MShapeStock
```

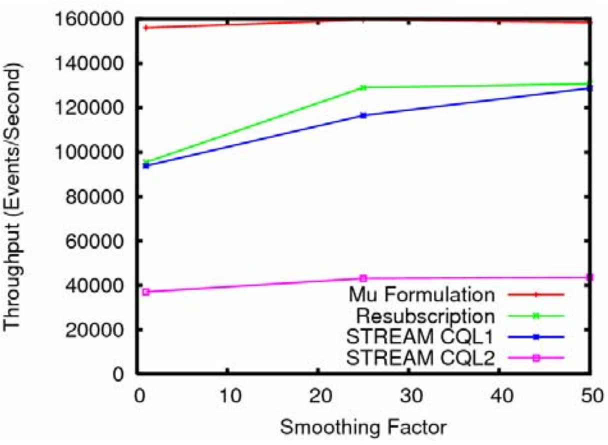
# Double-Top Query: CQL

```
vquery : Rstream (Select S.time, S.name, S.price, (S.price - P.price)
  From Stock [Now] as S, Stock [Partition By P.name Rows 2] as P Where S.name = P.name and S.time > P.time);
vtable : register stream StockDiff (time integer, name integer, price float, pdiff float);

vquery : Rstream (Select P.time, P.name, P.price, P.pdiff
  From StockDiff [Now] as S, StockDiff [Partition By P.name Rows 2] as P Where S.name = P.name and (S.pdiff * P.pdiff) < 0.0);
vtable : register stream Extrema (time integer, name integer, price float, pdiff float);

vquery : Select name, count(*) from Extrema Group By name;
vtable : register relation ExtremaCounter (name integer, seqNo integer);

vquery : Rstream (Select E.name, E.price, E.pdiff, C.seqNo, C.seqNo – 1
  From Extrema [Now] as E, ExtremaCounter as C Where E.name = C.name);
vtable : register stream ExtremaSeq (name integer, price float, pdiff float, seq integer, prevSeq integer);

vquery : Select name, price, seq from ExtremaSeq Where pdiff < 0.0;
vtable : register relation stateA (name integer, price float, seq integer);

vquery : Rstream (Select E.name, E.price, A.price, E.seq
  From ExtremaSeq [Now] as E, stateA as A Where E.name = A.name and E.prevSeq = A.seq and E.price > (A.price * 1.2));
vtable : register relation stateB (name integer, bprice float, aprice float, seq integer);

vquery : Rstream (Select E.name, E.price, B.bprice, B.aprice, E.seq From ExtremaSeq [Now] as E, stateB as B
  Where E.name = B.name and E.prevSeq = B.seq and E.price > (B.aprice * 0.9) and E.price < (B.aprice * 1.1));
vtable : register relation stateC(name integer, cprice float, bprice float, aprice float, seq integer);

vquery : Rstream (Select E.name, E.price, C.cprice, C.bprice, C.aprice, E.seq from ExtremaSeq [Now] as E, stateC as C
  Where E.name = C.name and E.prevSeq = C.seq and E.price > (C.bprice * 0.9) and E.price < (C.bprice * 1.1));
vtable : register relation stateD (name integer, dprice float, cprice float, bprice float, aprice float, seq integer);

query : Rstream (Select E.name, E.price, D.dprice, D.cprice, D.bprice, D.aprice from ExtremaSeq [Now] as E, stateD as D
  Where E.name = D.name and E.prevSeq = D.seq and E.price <= D.aprice);
```

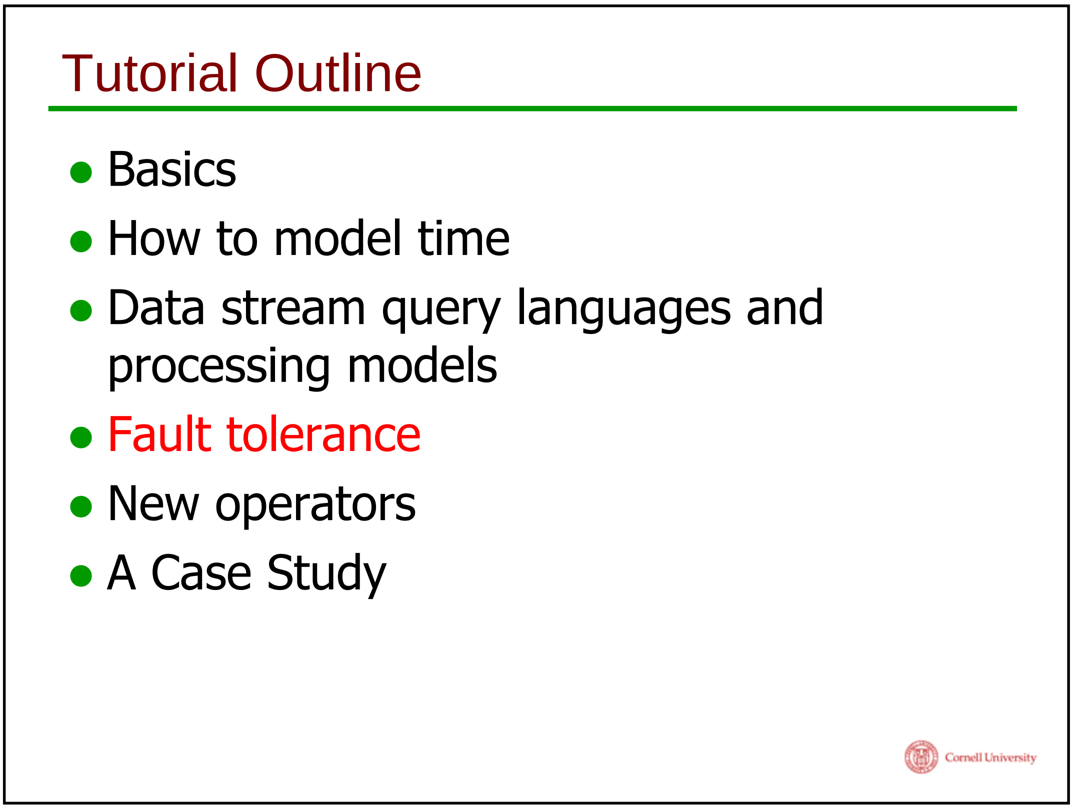
Cornell University

# Example: Double-Top

- Real stock data (24 companies, 112,635 events)

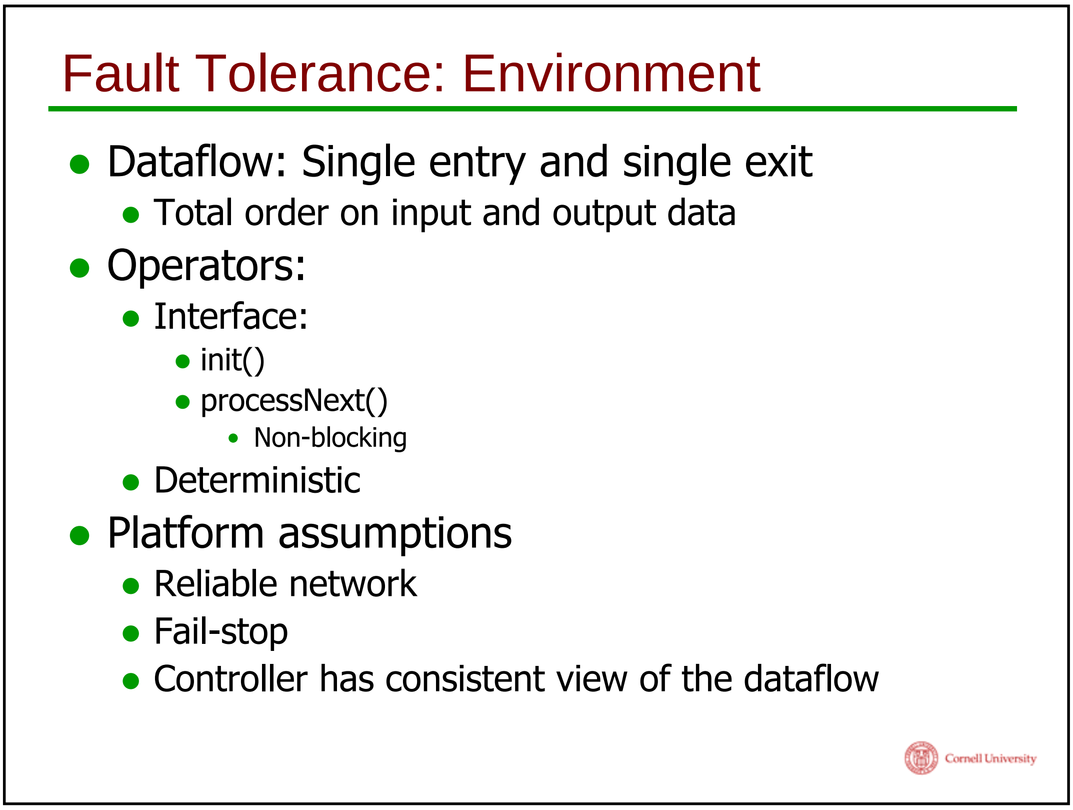


Cornell University



## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
- Fault tolerance
- New operators
- A Case Study



## Fault Tolerance: Environment

- Dataflow: Single entry and single exit
  - Total order on input and output data
- Operators:
  - Interface:
    - init()
    - processNext()
      - Non-blocking
  - Deterministic
- Platform assumptions
  - Reliable network
  - Fail-stop
  - Controller has consistent view of the dataflow

## Example Dataflow

- Analyze network packets from a firewall
- Track minimum and average duration of network sessions on a per source and application basis

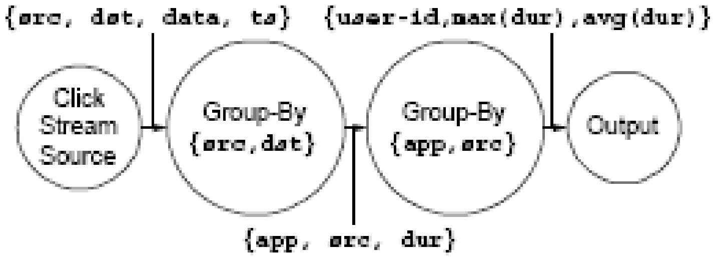


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

## Fault-Tolerance: Basic Approach

- Process pair: Coordinate redundant computation between a primary and a secondary
- Properties of the resulting dataflow:
  - Loss-free: No data in the input sequence is lost
  - Duplicate-free: No data in the input sequence is duplicated

# Fault-Tolerance: Basic Approach (2)

- Main idea:
  - Add new operators that connect existing operators in the replicated dataflow
    - Add boundary operators
  - Assign a sequence number to each tuple
  - Coordinate consumer and producer operators to store, ack, and flush data accordingly

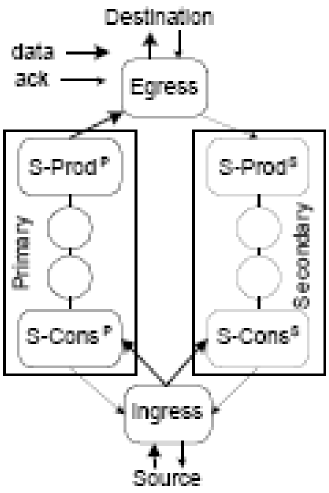


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

# Fault-Tolerance: Basic Approach (3)

- Assumption: Ingress and Egrees do not fail
- Notation:
  - P: Primary
  - S: Secondary

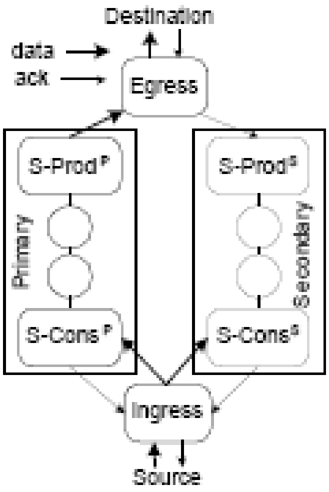


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

## Normal Case Protocol

- Ingress forwards data to both S-Cons$^P$ and S-Cons$^S$; discards it once it has received acks from both
- Only S-Prod$^P$ forwards the result to Egress
- Egress acks results to S-Prod$^S$, which then discards the data

- Note: S-Prod$^S$ could have dangling sequence numbers

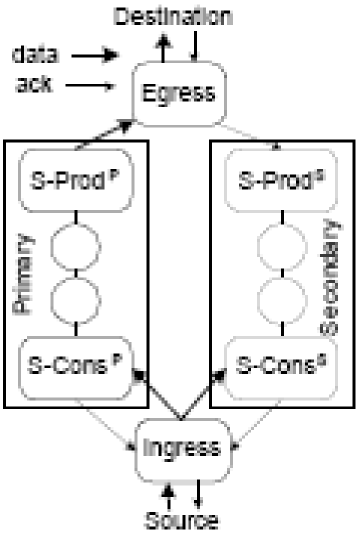


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

## Take-Over During Failure

- Assume wolg primary fails
- S-Prod$^S$ starts sending to Egress
  - Buffer at S-Prod$^S$ only has unacked tuples or dangling sequence numbers

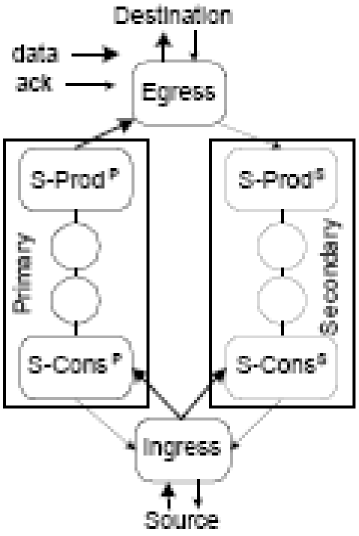


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

## Catch-Up

- S-Cons$^S$ [which is now the new primary!] quiesces the dataflow
- Send state to new secondary
  - API: getState() and installState()
- Fold in new secondary

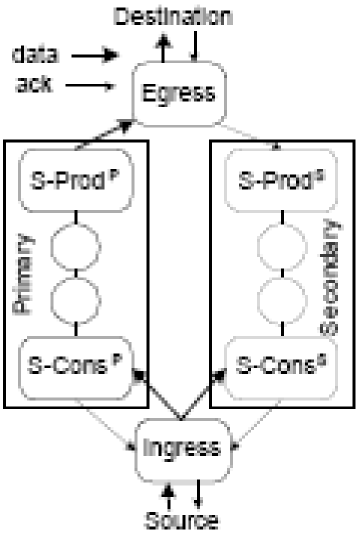


Figure from Highly Available, FaultTolerant, Parallel Dataflows, SIGMOD 2004.

## This Can Be Made Formal

Idea: Specify behavior as a state-machine

Variables:

Buffer: Array of (sn, tuple, mark)

del: {REC|PRIM|SEC}

status[src], status[dest] =
{ACTIVE|DEAD|STDBY|PAUSE}

conn[src], conn[dest] =
{SEND|RECV|ACK|PAUSE}

dest = {PRIM,SEC}

## State Machine of Normal Processing

| | |
|---|---|
| Not B.full() | {t = processNext();<br>B.put(t.sn,t,del)} |
| status[dest]=ACTIVE and<br>SEND in conn[dest] | {t=B.peed(dest);<br>send(dest,t);<br>B.advance(dest);} |
| status[dest]=ACTIVE and<br>SEND in conn[dest] and<br>ACK not in conn[dest] | {t=B.peed(dest);<br>send(dest,t);<br>B.advance(dest);<br>B.ack(t.sn, dest, del);} |
| status[dest] = ACTIVE and<br>ACK in conn[dest] | {sn=recv(dest);<br>B.ack(t.sn, dest, del);} |

## Rollback Recovery

Passive standby:

- Into the stream of data, add delta-checkpoint messages. When an operator processes such a message, it captures the change in state since the last checkpoint
  - Secondary does not do much processing

Upstream backup:

- Log all the data at upstream nodes
  - Queue trimming by eliminating data that cannot contribute to the current state
  - Need to find the earliest data item that contributed to the current state

## Tutorial Outline

- Basics
- How to model time
- Data stream query languages and processing models
  - STREAM and CQL
  - Cayuga
- Fault tolerance
- New operators
  - Change detection
- A Case Study

## Types of Change

- Transient change.
  - Bursts in a datastream (later talk)
- Specific patterns (intrusion detection).
  - Particular sequence of data.
  - Change in mean or variance.
- Long-term change of unknown character.
  - Change in distribution.
  - Power/generality.

## Model for Long-term Change

- $X_1, X_2, X_3, \ldots$
  - $X_i \sim D_i$
  - $X_i$ are independent random variables
- Detect when

  $D_1 = D_2 = \ldots D_i \quad D_{i+1} = D_{i+2} = \ldots$
- Two sample case, $\neq$
  - $S_1 = \{Y_1, \ldots, Y_n\} \sim D_a$, $S_2 = \{Z_1, \ldots, Z_m\} \sim D_b$
- Continuous/Discrete distributions.

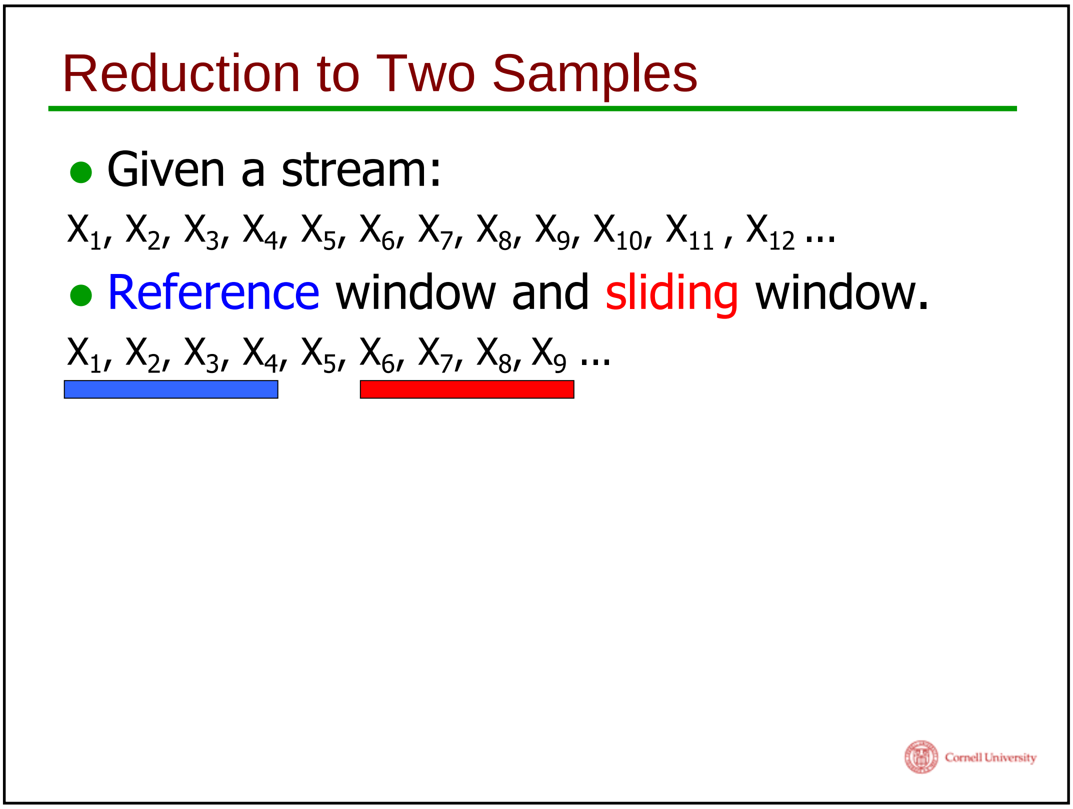
## Reduction to Two Samples

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underbrace{X_1, X_2, X_3, X_4,}_{🟦} \underbrace{X_5, X_6, X_7, X_8}_{🟥} \ldots$



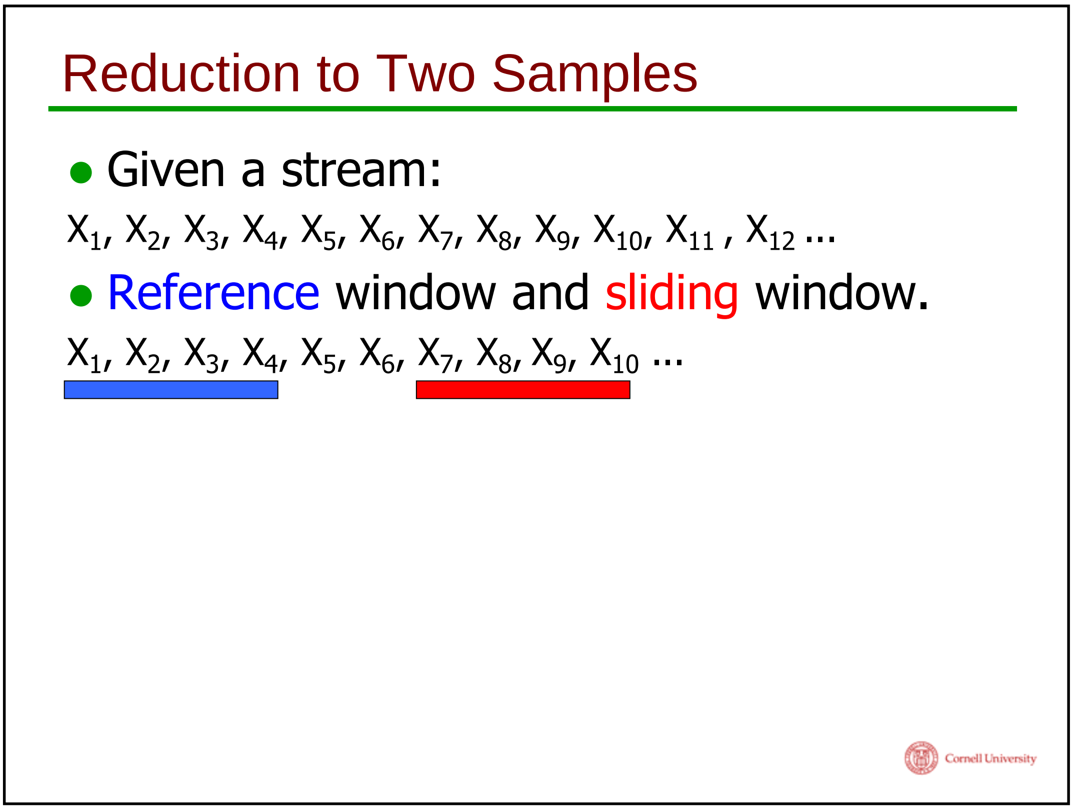
## Reduction to Two Samples

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underbrace{X_1, X_2, X_3, X_4}, X_5, \underbrace{X_6, X_7, X_8, X_9} \ldots$



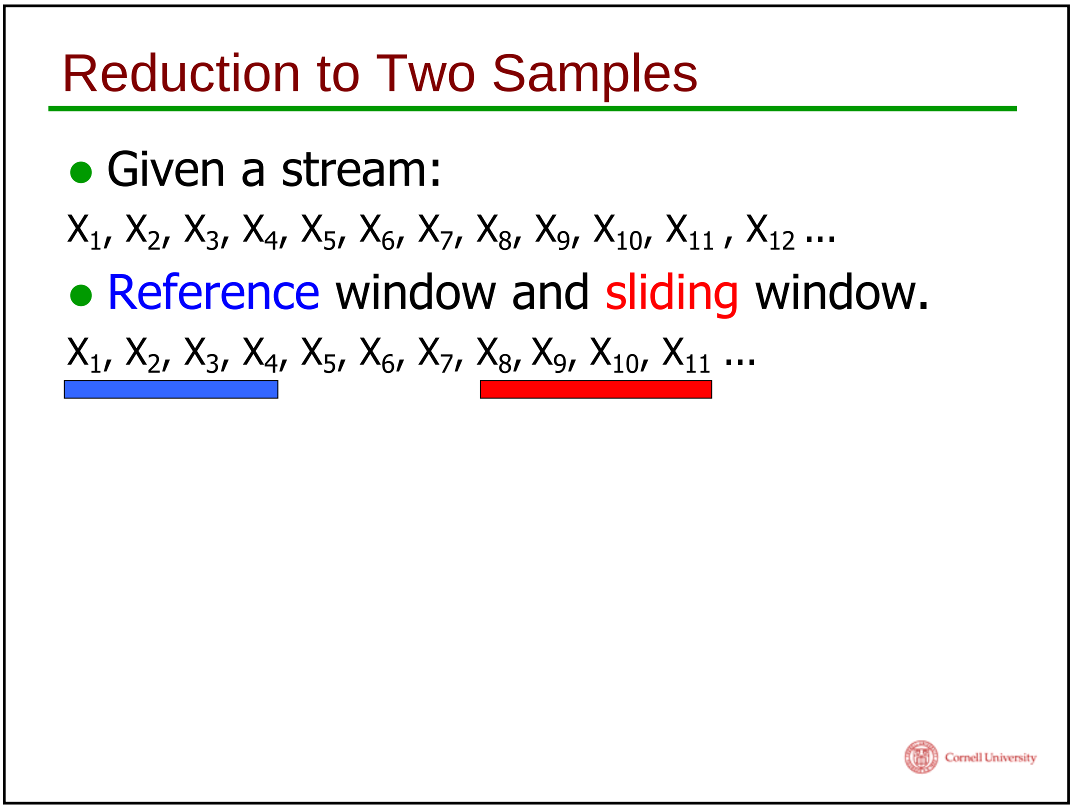
## Reduction to Two Samples

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underbrace{X_1, X_2, X_3, X_4}, X_5, X_6, \underbrace{X_7, X_8, X_9, X_{10}} \ldots$



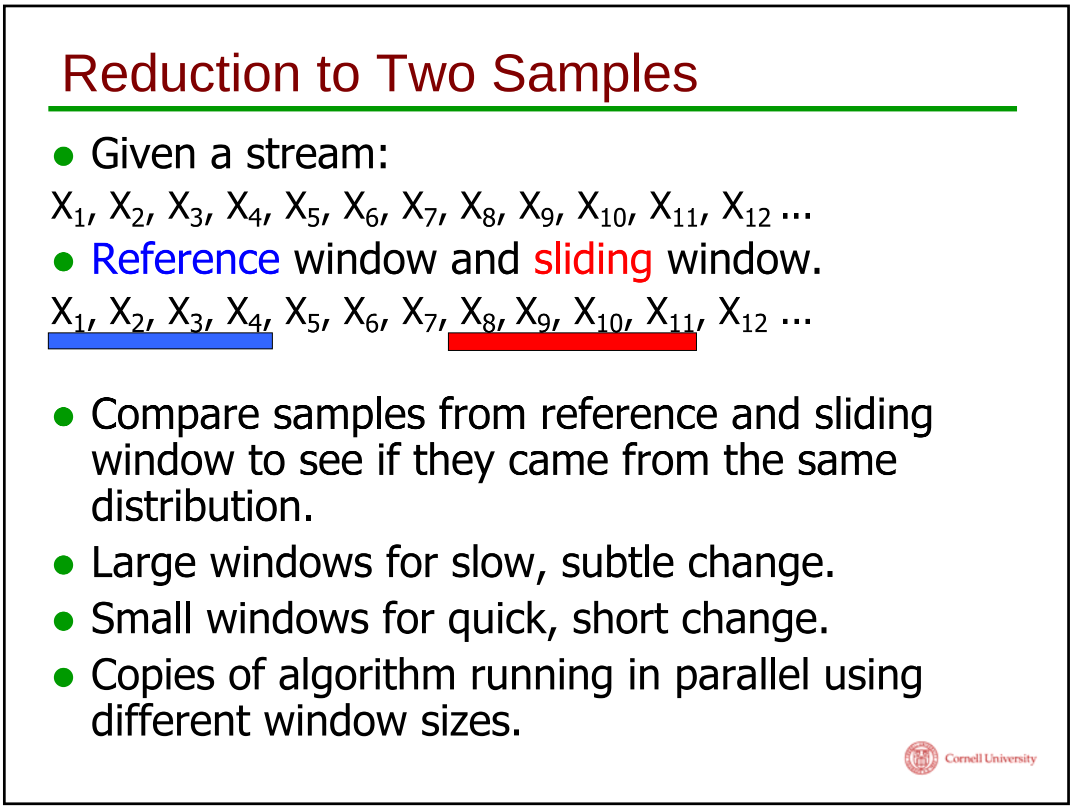
## Reduction to Two Samples

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underline{X_1, X_2, X_3, X_4,} X_5, X_6, X_7, \underline{X_8, X_9, X_{10}, X_{11}} \ldots$



## Reduction to Two Samples

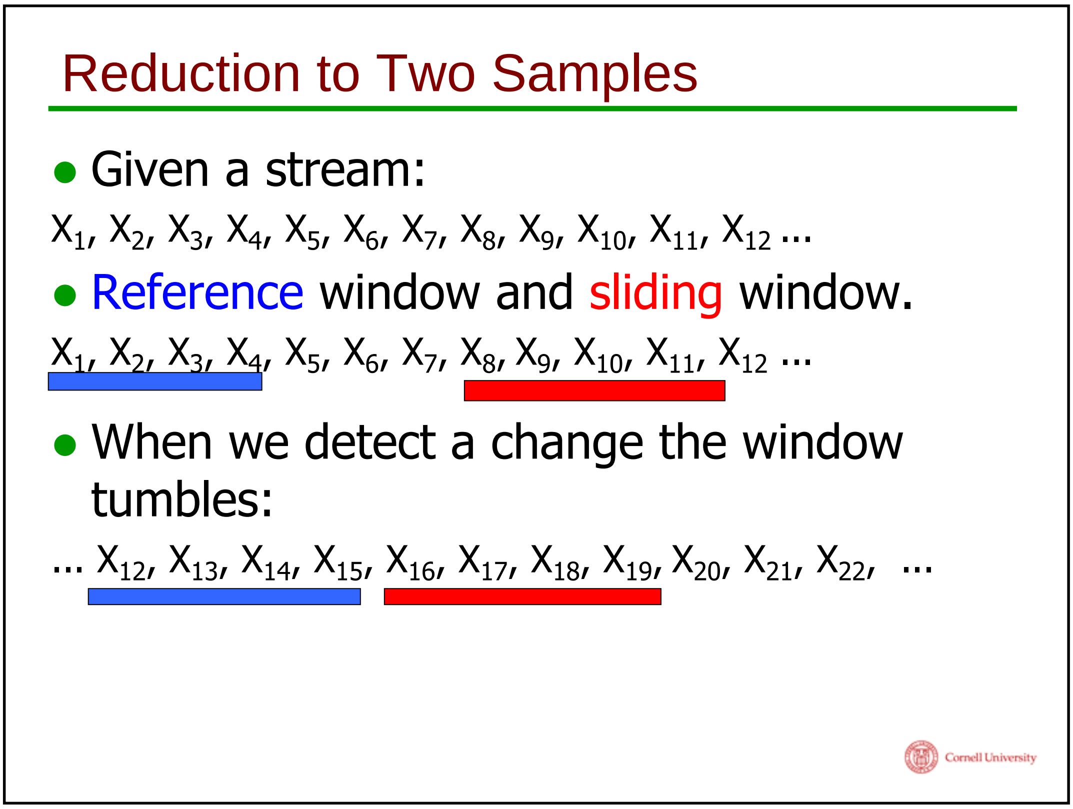
- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underline{X_1, X_2, X_3, X_4,} X_5, X_6, X_7, \underline{X_8, X_9, X_{10}, X_{11}}, X_{12} \ldots$

- Compare samples from reference and sliding window to see if they came from the same distribution.
- Large windows for slow, subtle change.
- Small windows for quick, short change.
- Copies of algorithm running in parallel using different window sizes.



## Reduction to Two Samples

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}$ …

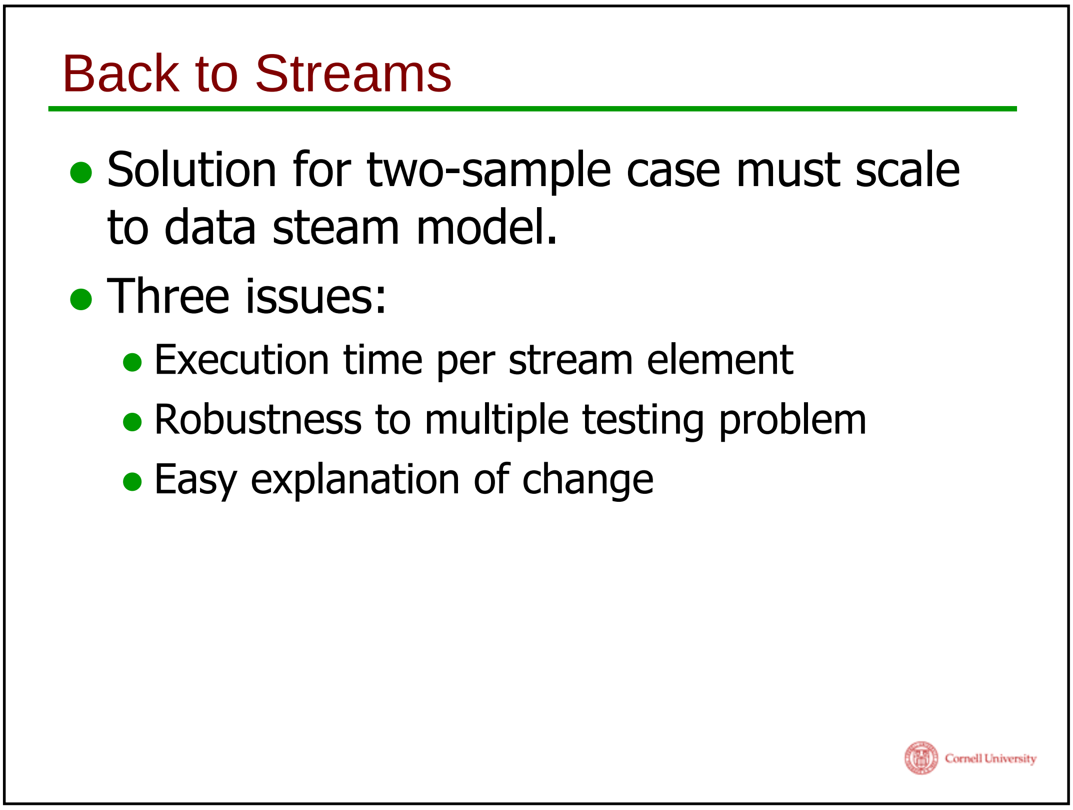
- Reference window and sliding window.

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12}$ …

- When we detect a change the window tumbles:

… $X_{12}, X_{13}, X_{14}, X_{15}, X_{16}, X_{17}, X_{18}, X_{19}, X_{20}, X_{21}, X_{22},$ …

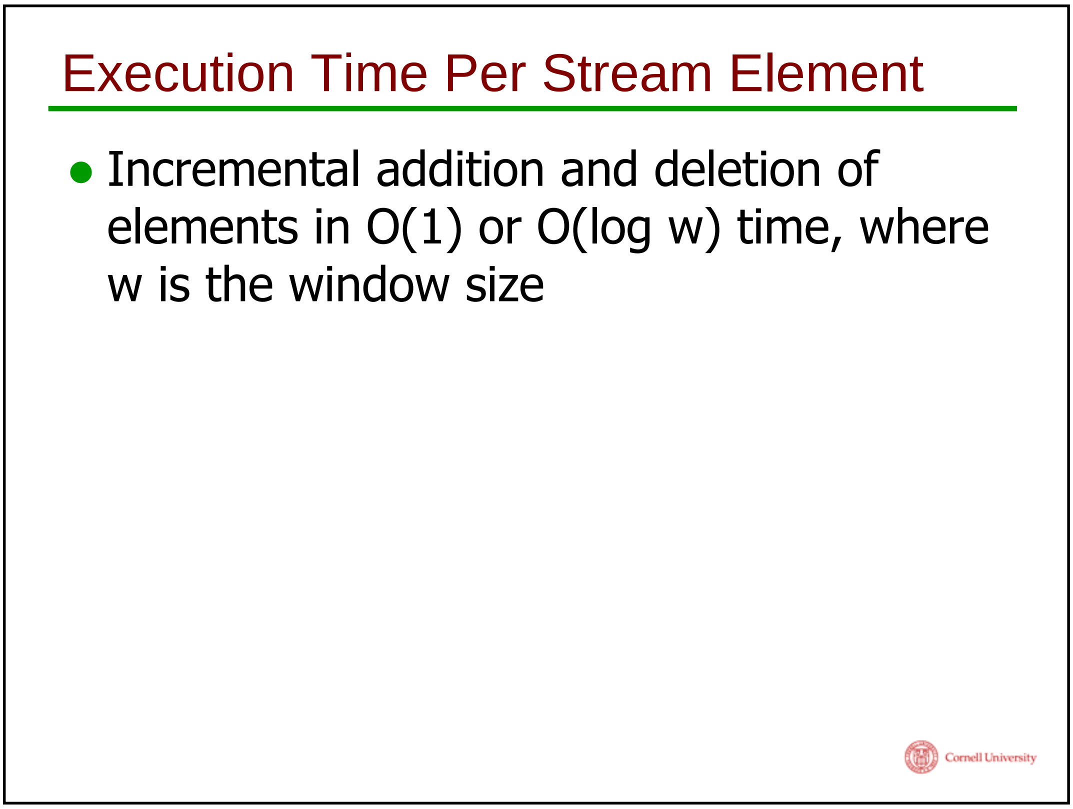
Cornell University



## Back to Streams

- Solution for two-sample case must scale to data steam model.
- Three issues:
  - Execution time per stream element
  - Robustness to multiple testing problem
  - Easy explanation of change

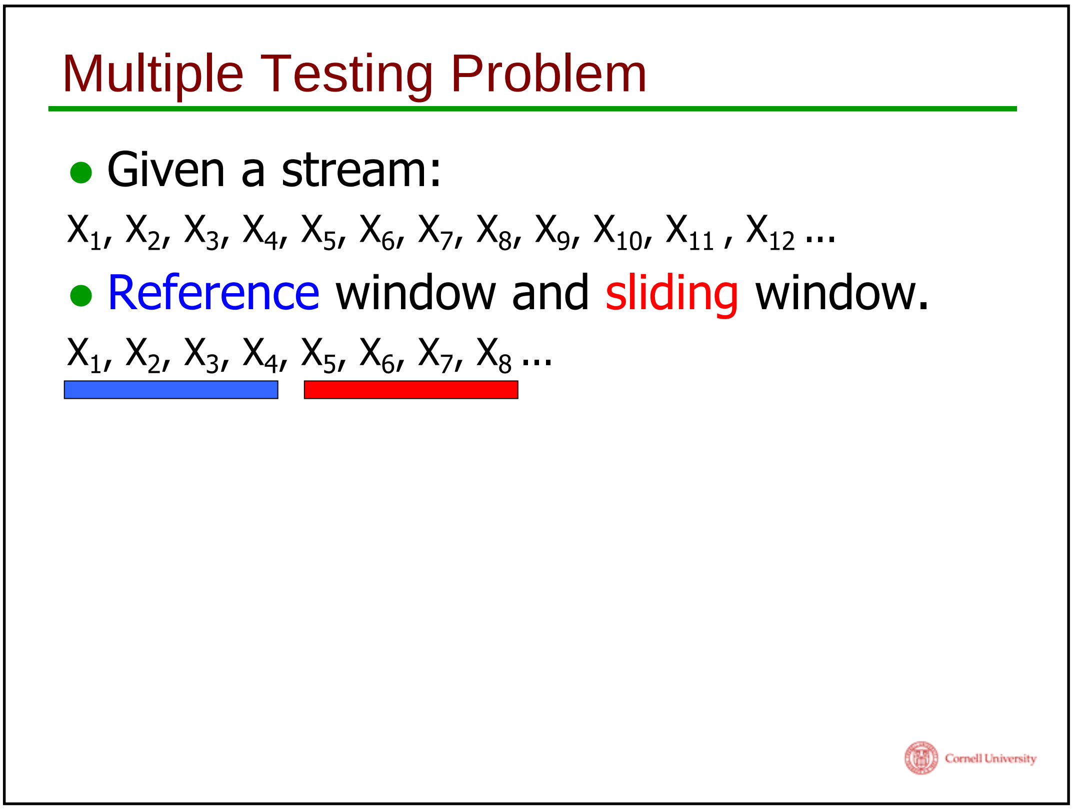
Cornell University



## Execution Time Per Stream Element

- Incremental addition and deletion of elements in O(1) or O(log w) time, where w is the window size

Cornell University



## Multiple Testing Problem

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$\underbrace{X_1, X_2, X_3, X_4,}\ \underbrace{X_5, X_6, X_7, X_8}\ \ldots$

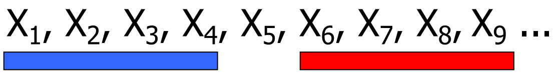
Cornell University

## Multiple Testing Problem

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.



$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9 \ldots$

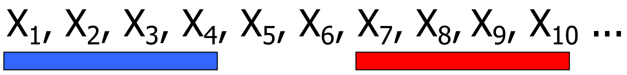
Cornell University

## Multiple Testing Problem

- Given a stream:

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.



$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10} \ldots$

Cornell University

## Multiple Testing Problem

- Given a stream:

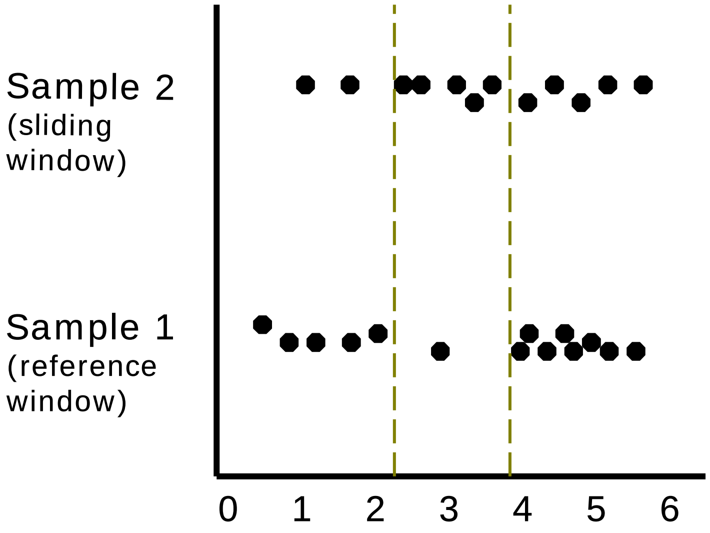
$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11}, X_{12} \ldots$

- Reference window and sliding window.

$X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, X_9, X_{10}, X_{11} \ldots$

Cornell University

## Simple Explanation of Change



Cornell University

## Simple Explanation of Change

- User might be interested in a collection A of regions of the input space.
  - Intervals of the form: [a, b]
  - Initial segments: (-∞, b]
- Regions as queries
  - Count number of points in this area.
  - Estimate probability of this area.
- Point out region C∈A with most significant change in probability (or top K).

- Compare: Decision trees with axis-orthogonal splits versus DTs with linear splits

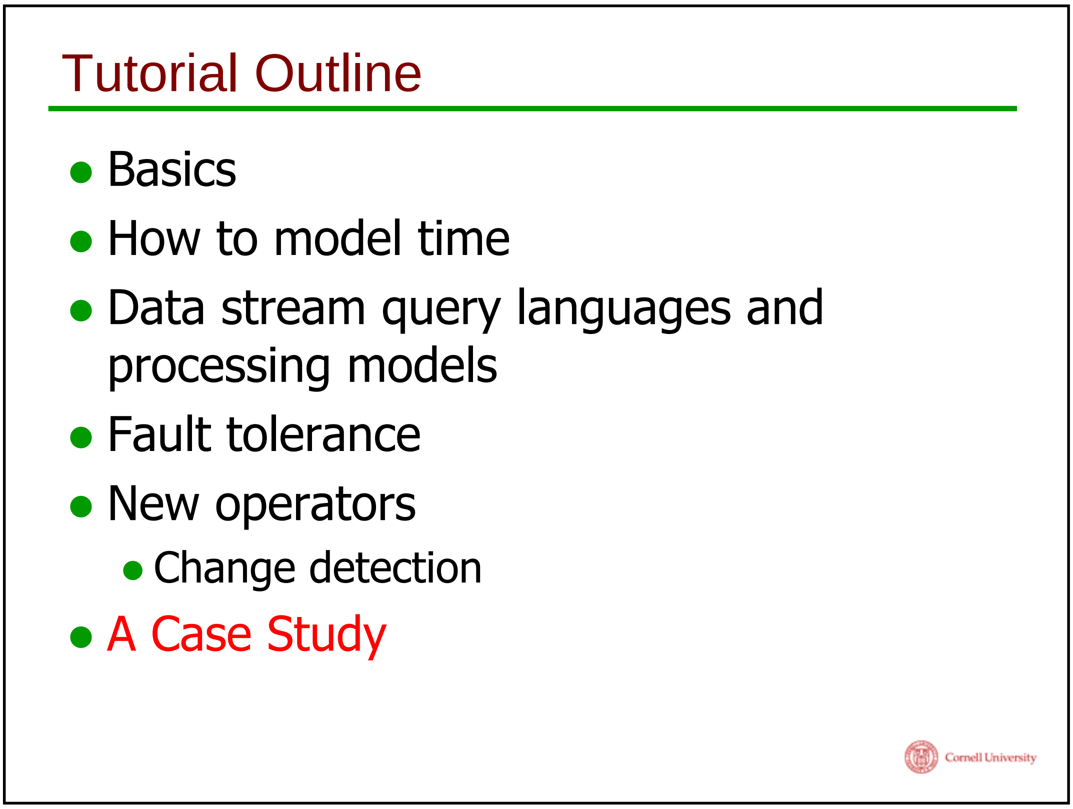
## Algorithm

- Regions:
  - Initial segments (-∞, a]
  - Incremental maintenance by keeping tuples in window sorted in-memory

- Comparison function:

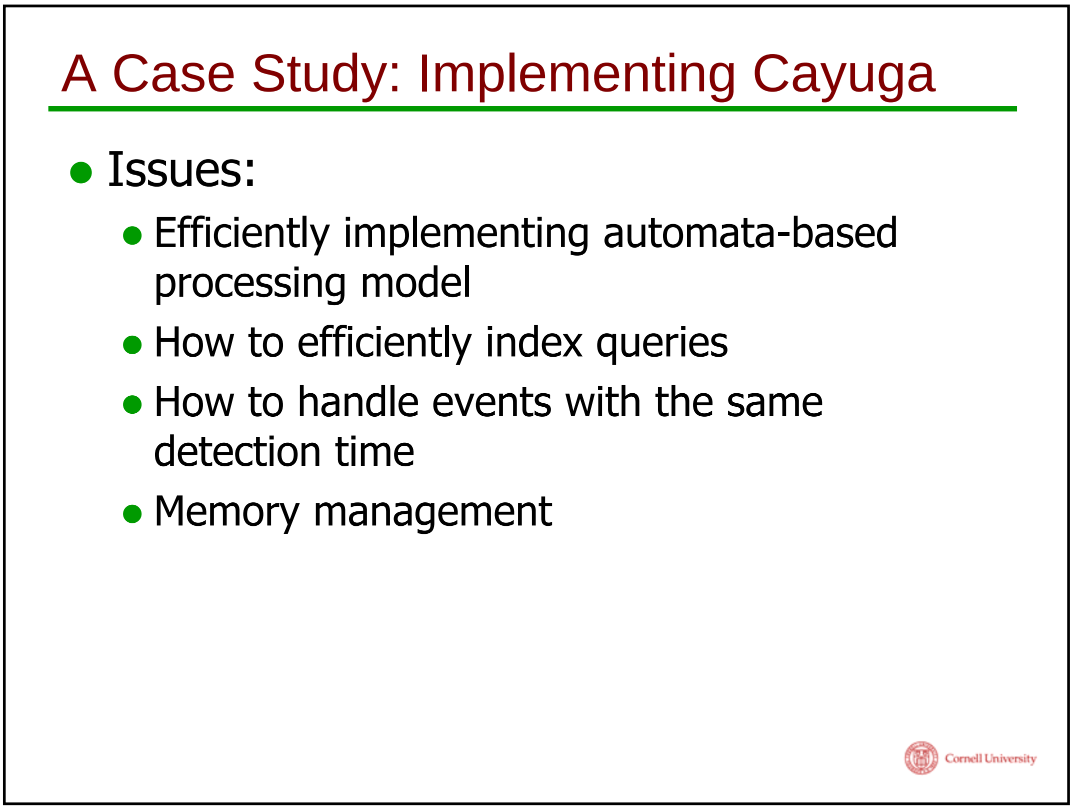
$$\phi = \max \frac{|F_2(c)-F_1(c)|}{\sqrt{\min\{F_1(c)+F_2(c), 2-F_1(c)-F_2(c)\}}}$$

$$\max | F_2(c) - F_1(c) |$$



## Tutorial Outline

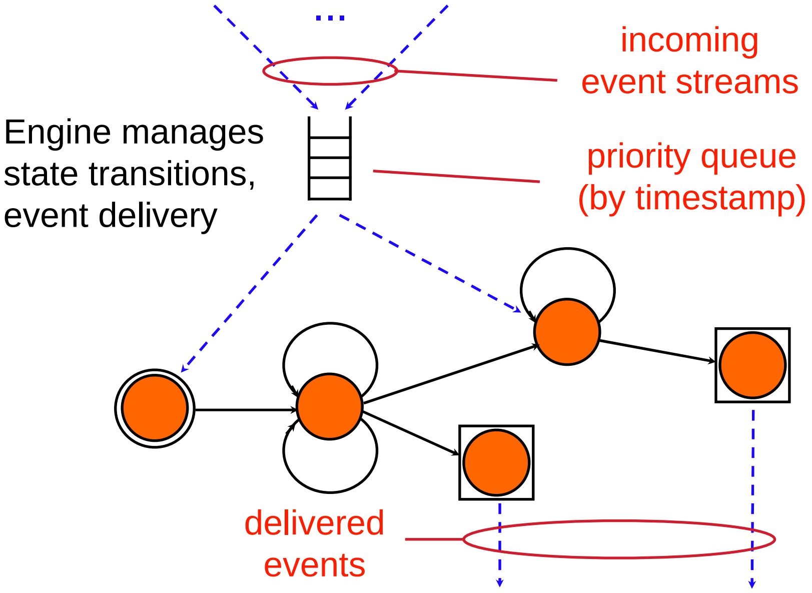
- Basics
- How to model time
- Data stream query languages and processing models
- Fault tolerance
- New operators
  - Change detection
- A Case Study



## A Case Study: Implementing Cayuga

- Issues:
  - Efficiently implementing automata-based processing model
  - How to efficiently index queries
  - How to handle events with the same detection time
  - Memory management

## Architecture of the NFA Engine



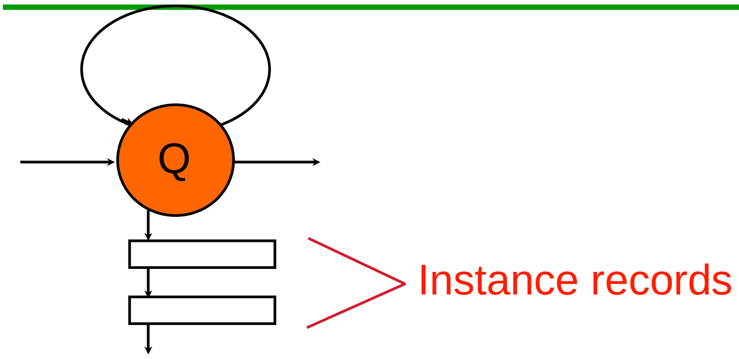
Engine manages state transitions, event delivery

incoming event streams

priority queue (by timestamp)

delivered events

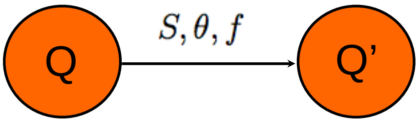
## Automaton States



Instance records

- Instance record contains stored data for a (nondeterministic) instance of the state
- All instances of a state have the same schema (known at query compile time)

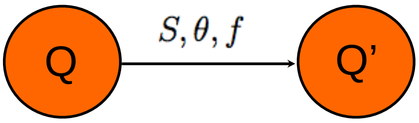
## Automaton Edges



- S identifies a *stream* of incoming events
- θ is a *selection predicate* on Schema(Q) X Schema(S)
- f is the *instance map:*
  - Schema(Q) X Schema(S) → Schema(Q')

Cornell University

## Automaton Edges II



- All edges leaving a given state Q have the same stream S
- θ and f are compiled code for an interpreter we designed for this purpose

Cornell University

## Optimization Goals

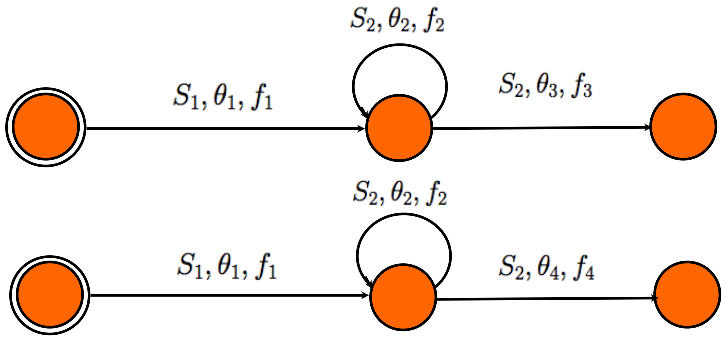
- Minimize
  - # automaton states
  - # instance records / state

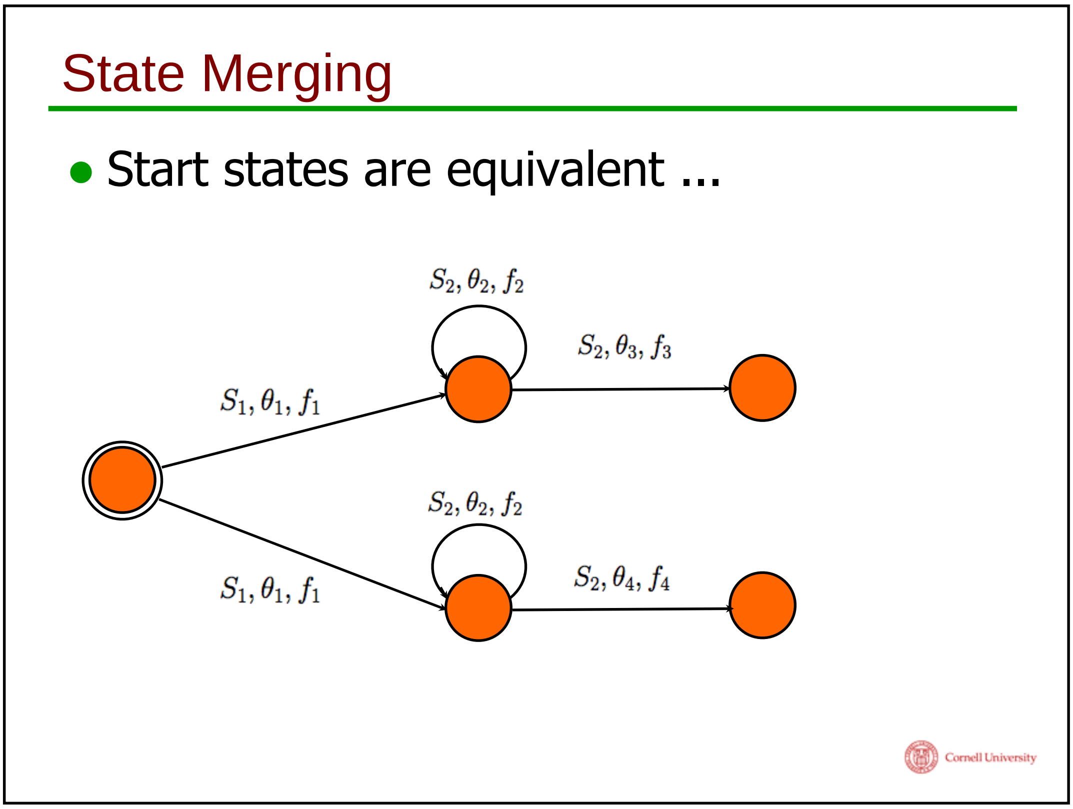
→ *Merging* techniques (states and instances)

- Quickly find
  - states affected by an input event
  - edges that are traversed

## State Merging

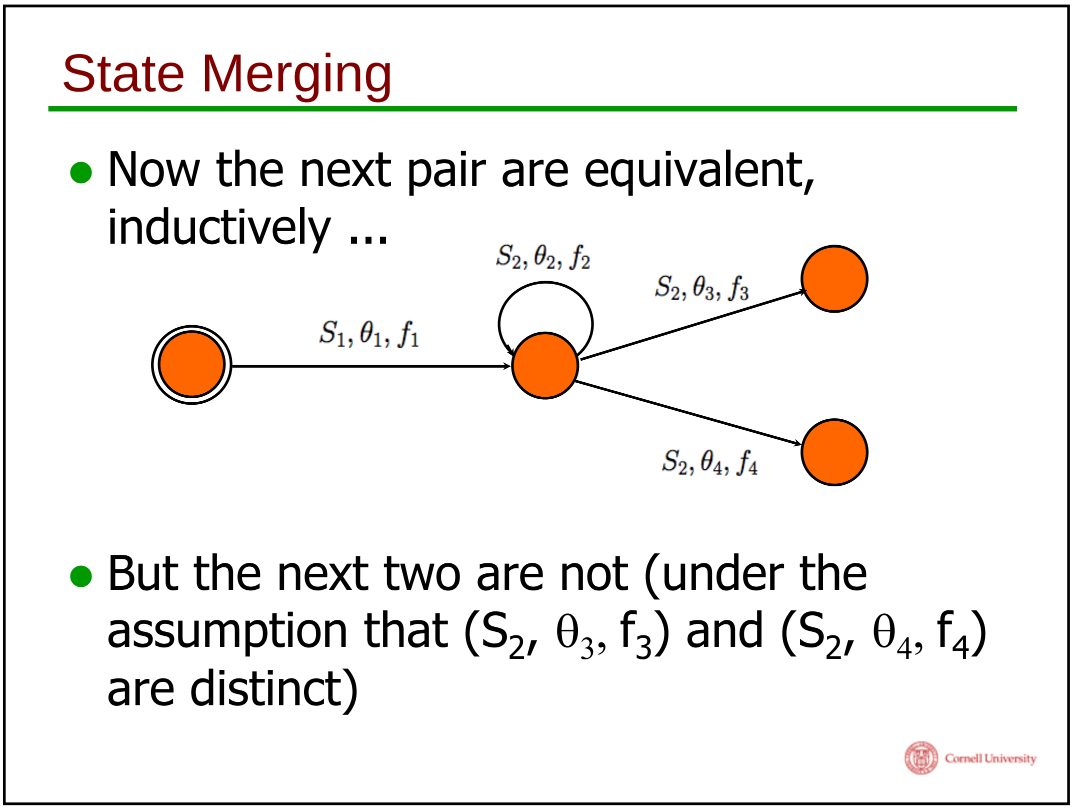
- Roughly: two states are equivalent if they have identically labeled entering edges from equivalent states





## State Merging

- Start states are equivalent ...



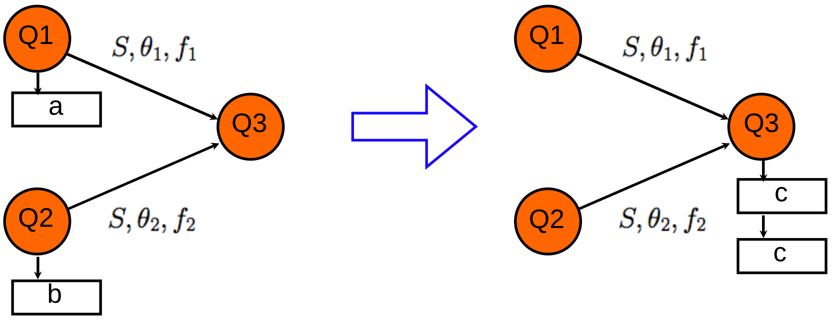
## State Merging

- Now the next pair are equivalent, inductively ...

- But the next two are not (under the assumption that ($S_2$, $\theta_3$, $f_3$) and ($S_2$, $\theta_4$, $f_4$) are distinct)

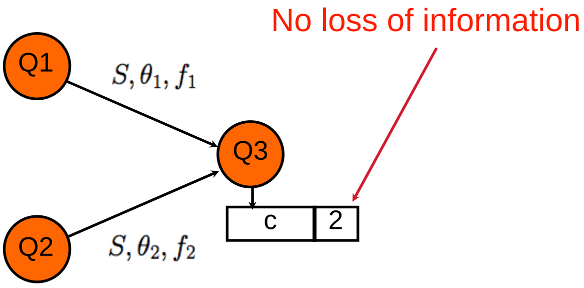
## Instance Merging

- Event e arrives on S, traverses edges from Q1 and Q2, both can produce equal instance records at Q3!



## Instance Merging II

- Clearly we want to merge instances if we can do it efficiently ... indexing (later) helps!
- Can yield exponential reduction in number of instance records
- Similar to dynamic programming or function caching



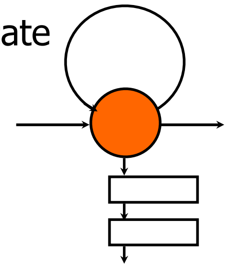
## Indexing

- Index maps event to set of predicates that are satisfied by it
  - essentially a standard pub/sub engine
- Choice: static or dynamic predicates
  - Static
    - easy to maintain
    - conservative approximation
  - Dynamic
    - precise
    - high update rate as instances change

Cornell University

## Indexing: Affected Nodes

- A node is affected if at least one instance does not traverse a filter edge
  - for most input events, most nodes are not affected!
  - nodes that are unaffected require no processing at all
- Construct index that yields (approx) the nodes affected by input event
  - e.g. can use static part of predicate
- Global index per input stream

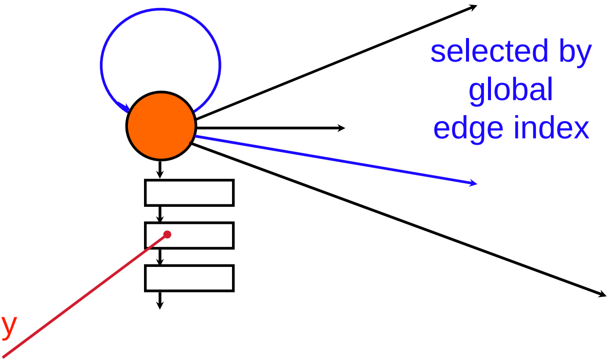


Cornell University

## Indexing: Forward/Rebind

- Conceptually ...
  - maps an event to (approx) the set of instance-edge pairs that satisfy it
- In Cayuga engine
  - global index (static parts of predicates) produces candidate edges
  - per-node index produces candidate instances
  - join the two results, testing predicates

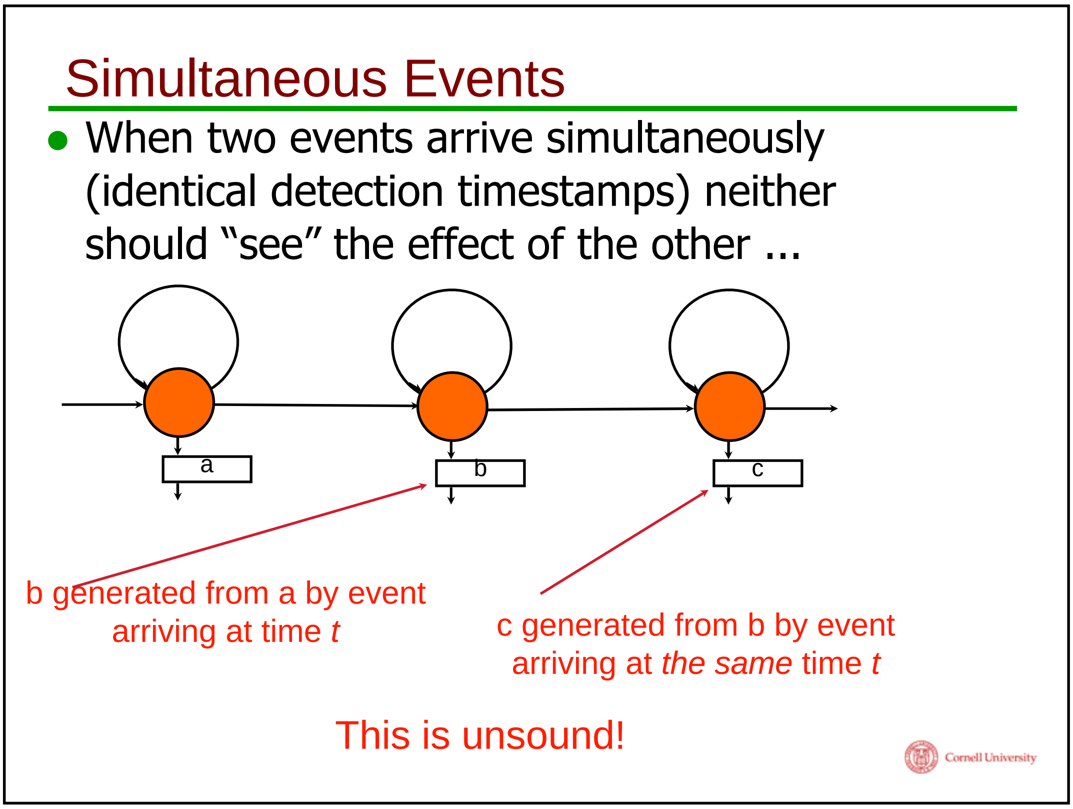
## FR Indexing II



selected by global edge index

selected by per-node instance index

Join the sets together and check predicate satisfaction



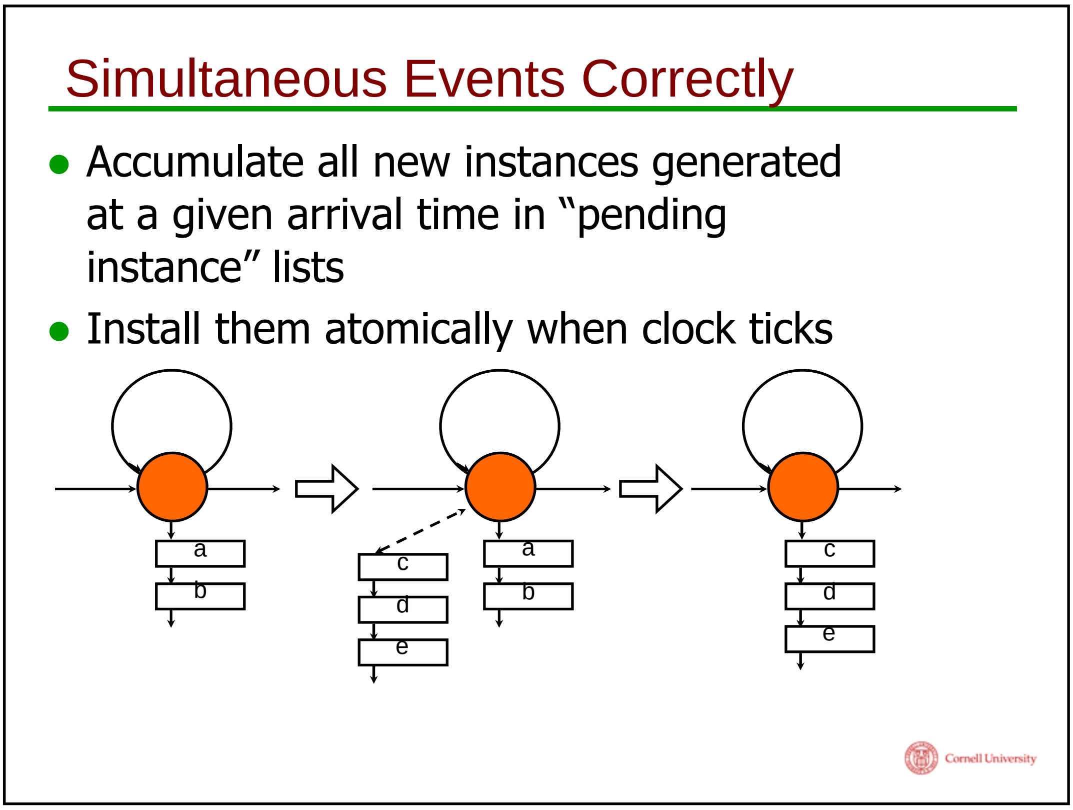
## Simultaneous Events

- When two events arrive simultaneously (identical detection timestamps) neither should “see” the effect of the other …

b generated from a by event arriving at time *t*

c generated from b by event arriving at *the same* time *t*

This is unsound!

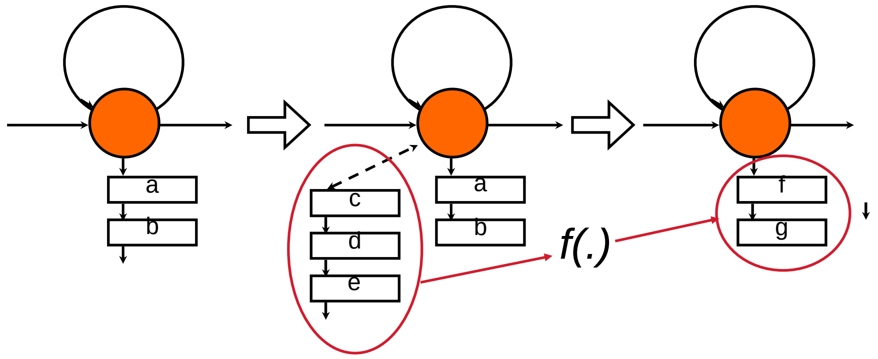


## Simultaneous Events Correctly

- Accumulate all new instances generated at a given arrival time in “pending instance” lists
- Install them atomically when clock ticks

## Aggregation of Pending Instance Lists

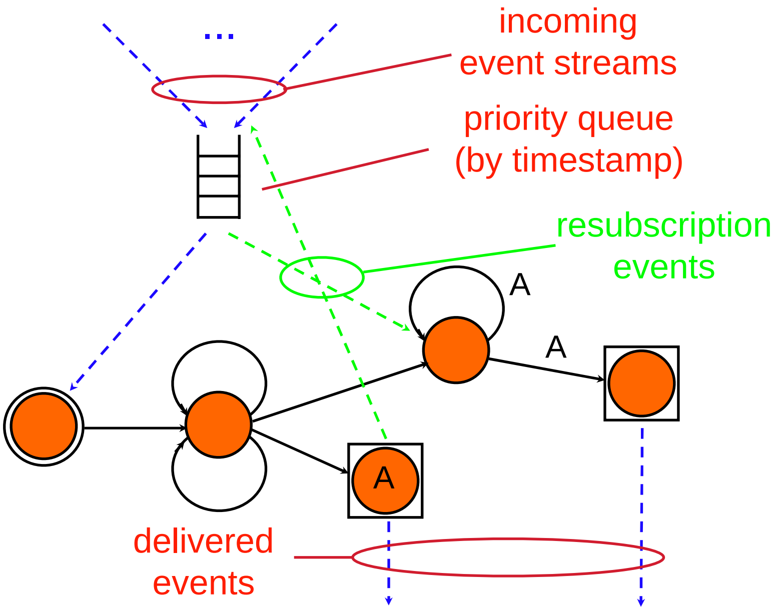
- Apply an aggregate computing function *f* to pending instances at installation time
- A "spatial" aggregate at a point in time



## Application to Resubscription

- Recall resubscription treats the output of an automaton final state as another input stream ...
- Just treat instances added to the pending list of a final state as if they were new events!

## Resubscription



incoming event streams

priority queue (by timestamp)

resubscription events

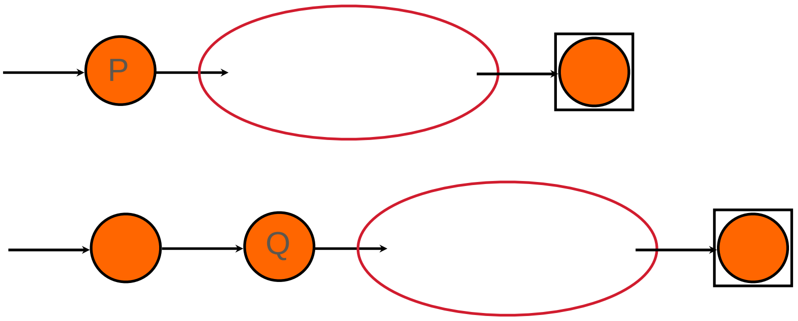
delivered events

## Resubscription

- It was a problem before we figured out how to do it correctly!
- Now
  - Gives us expressiveness of the entire stream algebra
  - Common subexpression elimination
  - Dynamically disable resubscription states
- Note: This will be harder once we distribute Cayuga across a cluster

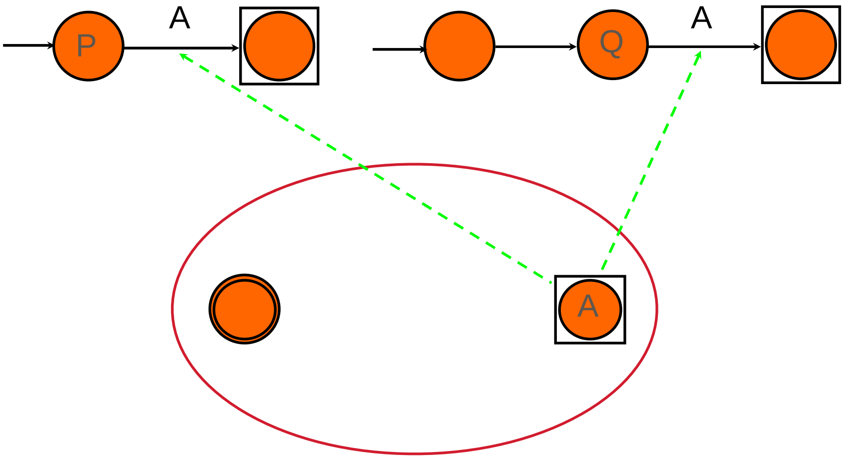
## Resubscribing to Common Subexpressions

- Common subexpressions where states P and Q are not equivalent ...



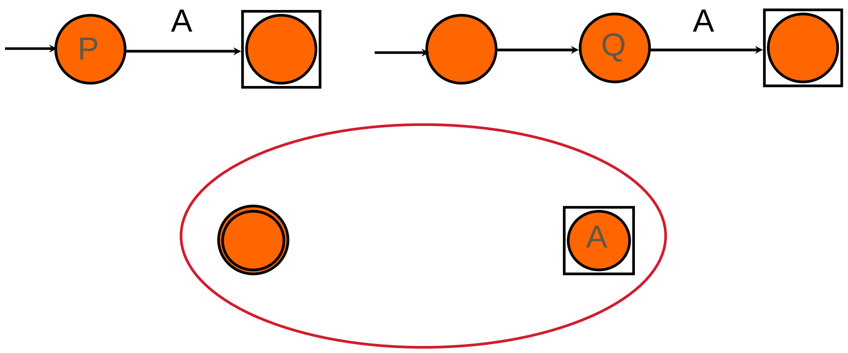
## Common Subexpressions II

- Convert to



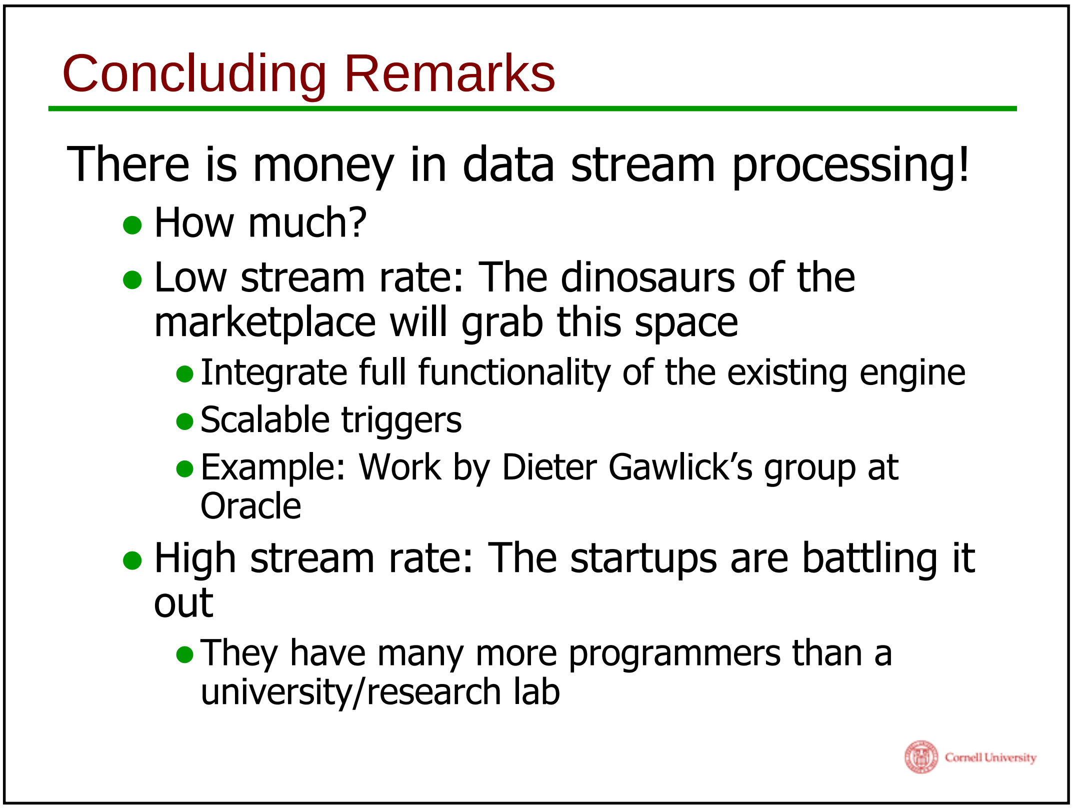
## Common Subexpressions III

- When neither P nor Q is occupied, states of the resubscription machine (determined by static analysis) can be *disabled*.



## In Summary: We Covered …

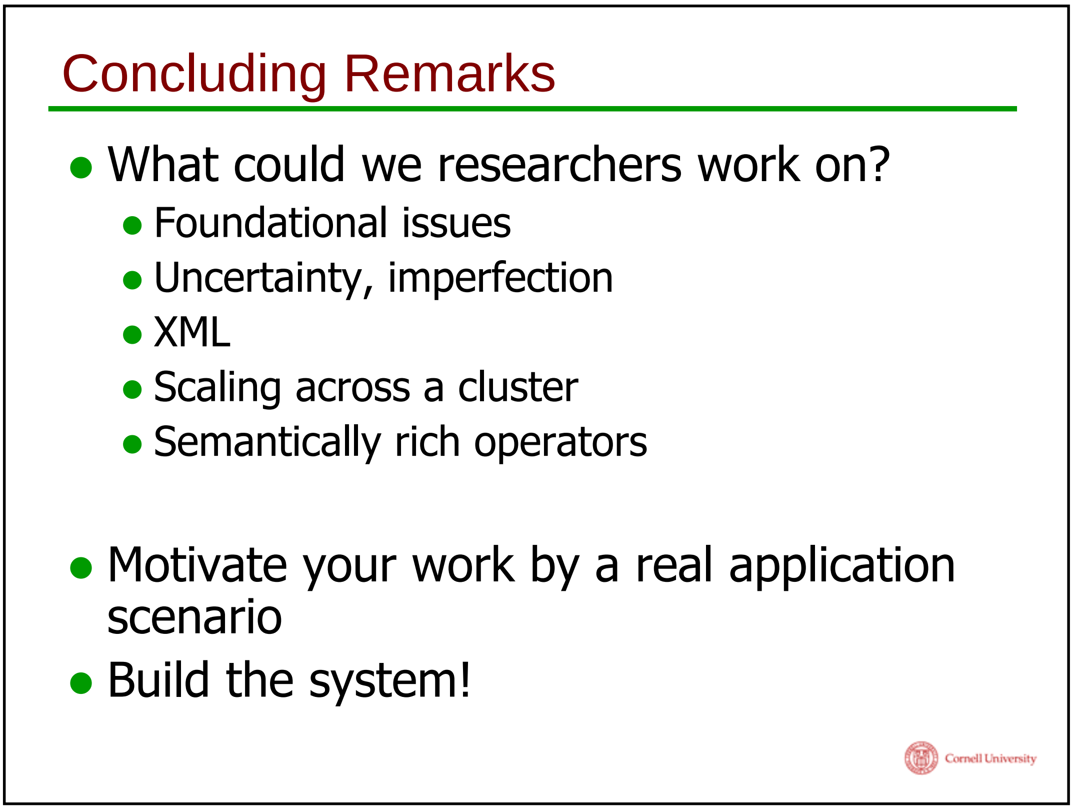
- Basics
- How to model time
- Data stream query languages and processing models
  - STREAM and CQL
  - Cayuga
- Fault tolerance
- New operators
  - Change detection
- A Case Study



## Concluding Remarks

There is money in data stream processing!

- How much?
- Low stream rate: The dinosaurs of the marketplace will grab this space
  - Integrate full functionality of the existing engine
  - Scalable triggers
  - Example: Work by Dieter Gawlick's group at Oracle
- High stream rate: The startups are battling it out
  - They have many more programmers than a university/research lab



## Concluding Remarks

- What could we researchers work on?
  - Foundational issues
  - Uncertainty, imperfection
  - XML
  - Scaling across a cluster
  - Semantically rich operators
- Motivate your work by a real application scenario
- Build the system!

## Thank you!

Cornell:

Alan Demers, Mingsheng Hong, Dan Kifer, Mirek Riedewald, Walker White

Stanford:

STREAM Team, Jennifer Widom

Microsoft:

Roger Barga, Jonathan Goldstein

## Questions?

johannes@cs.cornell.edu
http://www.cs.cornell.edu/johannes