Big Data Management and NoSQL Databases

Lecture 13. Data Stream Management

PD Dr. Andreas Behrend
Data Stream

• A data stream is a sequence of data tuples.
• Think of standard tuples of relational databases.
• With time information (timestamps)
• One after the other, or in batches, they are generated.

• That means, Data is moving! Continuously generated (assumed infinite!)
• Potentially high pace.

• System has to process data without first storing everything (how would that be possible anyway if stream is infinite?!)
Sensor Networks as Data Streams Origin

- E.g., in Environmental Monitoring

StationStream(timestamp, humidity, solarRadiation, windSpeed, snowHeight)

- Various application scenarios:
  - avalanche risk level computation
  - insights for agriculture
  - air pollution (urban) monitoring
Sample Application

• **The Pothole Patrol**

• Detecting and reporting the surface conditions of roads; using sensors in vehicles

• **Using 3-axis accelerometer+GPS + learning**

Sample Application

- **Environmental monitoring**
- Sensor data management and meta data sharing.
- Across many different types of measurement: (hydrology, alpine monitoring, atmospheric phenomena, earthquakes, ...)

- Also higher level applications like putting sensors and interpretations on maps, computing statistics over streams.

http://www.swiss-experiment.ch
Earthquake News on Twitter
Earthquake News on Twitter
Earthquake News on Twitter
Earthquake News on Twitter

source: http://blog.socialflow.com/
Classic Example: Stock Market

- Real-time analysis of stock marked changes
- Computing statistics over streams, e.g., for decision support
- Opportunities for reacting in real-time
- Even with fully automated means: algorithmic trading
So Far: Databases/NoSQL Datastores

- Data is changing, yes, but this is more due to inserts and update to stored data items
- Historic data is kept
- Queries operate on full data (tables)
- MapReduce is extreme, Write-once & Read-many times
- Data warehousing, too: periodically loading data in store for deep(er) analytics
- Data mining
Traditional Data Management …

- At query time, data is accessed as a whole
- Data is persistently stored
- Queries are ad-hoc (mainly)
Traditional Data Management vs. Data Stream Mgmt

- **Data is moving!** Continuously generated (assumed infinite!)
- **At high pace**
- **Queries are (mainly) continuous** (aka. standing). Registered once, observed “forever”.
- Answer to queries in (near) **real-time** required (often)
- Probabilistic methods for efficiency or considering only part of the stream (**sliding window**)
## DBMS vs. DSMS

<table>
<thead>
<tr>
<th>Database management system (DBMS)</th>
<th>Data stream management system (DSMS)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Persistent data (relations)</td>
<td>Volatile data streams</td>
</tr>
<tr>
<td>Random access</td>
<td>Sequential access</td>
</tr>
<tr>
<td>One-time queries</td>
<td>Continuous queries</td>
</tr>
<tr>
<td>(theoretically) unlimited secondary storage</td>
<td>Limited main memory</td>
</tr>
<tr>
<td>Only the current state is relevant</td>
<td>Consideration of the order of the input</td>
</tr>
<tr>
<td>Relatively low update rate</td>
<td>Potentially extremely high update rate</td>
</tr>
<tr>
<td>Little or no time requirements</td>
<td>Real-time requirements</td>
</tr>
<tr>
<td>Assumes exact data</td>
<td>Assumes outdated/inaccurate data</td>
</tr>
<tr>
<td>Plannable query processing</td>
<td>Variable data arrival and data</td>
</tr>
</tbody>
</table>

Data Stream Model

• **Stream of data items is unbounded** (available memory is not)
• **No way to store entire stream** (how could we, its (probably) not ending)
• To compute query results, need to devise **algorithm with little memory consumption**
Overview of Data Stream Topics

- **Synopses:**
  - concise representations of stream content
  - tailored to tasks, e.g., counting distinct elements
  - usually not exact, but approximations (estimators) of true values.
  - generally useful for representing data compactly
  - We will look at some of them today

- **(Sliding) Windows:**
  - focus of certain recent subset of data
  - computation of functions/joins over window(s) content
  - Will look at CQL language: think “SQL” for streaming data
Data Stream Mining: Teasers

• I tell you integer numbers between 1 and N
• I will tell all but one number

481  324  122  412  871  231  849  447  641 ...

• After N-1 numbers I ask: which number was missing?
Data Stream Mining: Teasers (Cont’d)

- **Keep Boolean array of length N:**
  - Mark position for observed number
  - Size required: N
  - Computation at end: N to find missing number
Data Stream Mining: Teasers (Cont’d)

• **Keep Boolean array of length N:**
  – Mark position for observed number
  – Size required: N
  – Computation at end: N to find missing number

• **Much better:**
  – keep sum of numbers: S
  – Missing number is N*(N+1)/2 - S
Counting Occurrences

• Consider a stream of elements $a_i$
  
  ..., $a_2$, $a_{84}$, $a_{41}$, $a_2$, $a_{77}$, $a_{231}$, $a_2$, $a_4$, $a_{54}$, ...

• How often does $a_2$ occur?

• How to implement?
Counting Occurrences

• Consider a stream of elements $a_i$

  ..., $a_2, a_{84}, a_{41}, a_2, a_{77}, a_{231}, a_2, a_4, a_{54}, ...$

• How often does $a_2$ occur?

• How to implement?

  • Keep counter for each id
  • Required space #ids (=N)
  • Not feasible of N is very large
Probabilistic Count'g: Count-Min Sketch

- **Keep 2-dim array** \((h, r)\)
- **\(h\) hash functions** \(h_i\) that map to range \(0...(r-1)\)

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>(h_1)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h_2)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h_3)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(h_4)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Arriving item \(x\).
- **For each** \(j\):  \(\text{array}[j, h_j(x)]++\)

Count-Min Sketch: Insert Example

Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>$h_1(x)$</th>
<th>$h_2(x)$</th>
<th>$h_3(x)$</th>
<th>$h_4(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data stream is $a, b, a, a, c, a, c, \ldots$
Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>$h_1(x)$</th>
<th>$h_2(x)$</th>
<th>$h_3(x)$</th>
<th>$h_4(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data stream is **a, b, a, a, c, a, c, ....**

red = inserted
Count-Min Sketch: Insert Example

Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>h₁(x)</th>
<th>h₂(x)</th>
<th>h₃(x)</th>
<th>h₄(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>... ... ... ... ...</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Data stream is a, b, a, a, c, a, c, ... 
red = inserted
Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>h₁(x)</th>
<th>h₂(x)</th>
<th>h₃(x)</th>
<th>h₄(x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data stream is **a, b, a, a, c, a, c, ....**
red = inserted
Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>$h_1(x)$</th>
<th>$h_2(x)$</th>
<th>$h_3(x)$</th>
<th>$h_4(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data stream is $a, b, a, a, c, a, c, ...$

red = inserted
Consider the following 4 hash functions, for ease of usage, displayed by their value when applied to a, b, or c:

<table>
<thead>
<tr>
<th>x</th>
<th>$h_1(x)$</th>
<th>$h_2(x)$</th>
<th>$h_3(x)$</th>
<th>$h_4(x)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>4</td>
<td>5</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>b</td>
<td>3</td>
<td>5</td>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>c</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Data stream is $a, b, a, a, c, a, c, ...$

red = inserted

Imagine that continues now a bit, then we might end up with ......
Count-Min Sketch: Counting

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>$h_1$</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>$h_2$</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>4</td>
<td>4</td>
<td>8</td>
</tr>
<tr>
<td>$h_3$</td>
<td>8</td>
<td>4</td>
<td>6</td>
<td>7</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>$h_4$</td>
<td>3</td>
<td>1</td>
<td>4</td>
<td>8</td>
<td>7</td>
<td>5</td>
</tr>
</tbody>
</table>

- How often did we see item $a$?
- Recall the hash function values for $a$:
  \[ h_1(a) = 4, \ h_2(a) = 5, \ h_3(a) = 0, \ h_4(a) = 2 \]
Count-Min Sketch: Counting

- How often did we see item a?
- Look at positions: $h_1(a) = 4$, $h_2(a) = 5$, $h_3(a) = 0$, $h_4(a) = 2$
- Take minimum of the corresponding values: Here: 4

Is this estimator generally underestimating or overestimating or can’t we say anything about that?
Count-Min Sketch: Counting

- How often did we see item a?
- Look at positions: $h_1(a) = 4$, $h_2(a) = 5$, $h_3(a) = 0$, $h_4(a) = 2$
- Take minimum of the corresponding values: Here: 4

- Estimate is never underestimating
- Overestimation probabilistically bounded
Continuous Queries
Data Stream Model

• A **stream** $S$ is a (possibly) infinite bag (multiset) of elements $<s, \tau>$ where $s$ is a tuple belonging to the **schema of $S$** and $\tau$ is the **timestamp** of the element.

• **Think:** tuples of a relational DBMS extended with timestamp, streaming in.
Data Streams: Example

- Monitoring of highway traffic: $\text{PosSpeedStr(vehicleId, speed, xPos, dir, hwy)}$

- E.g., for:
  - congestion prediction/warning
  - estimates of travel time
  - toll collection
  - ticket for too fast driving
Data Streams: Example

• Environmental Monitoring

StationStream(humidity, solarRadiation, windSpeed, snowHeight)

• Various application scenarios:
  – avalanche risk level computation
  – insights for agriculture
  – air pollution (urban) monitoring
Continuous Queries

• **In contrast to ad-hoc, single time queries** in (relational) DBMS.

• Queries over Streams are considered continuous: *registered once, run “forever”*:
  – “want to stay updated to avalanche risk, not just check once”

• **Also called standing queries or subscriptions** (in publish/subscribe context)

• **For instance:**
  – Compute average temperature.
  – Select all orders of stock “Apple” with quantity larger than 100.
What and How can we Compute DB-Style Queries?

• How to compute average values over an infinite stream? Block forever?

• How to join infinite streams if join partners can arbitrarily arrive (or not)?
What and How can we Compute DB-Style Queries?

• How to compute average values over an infinite stream? Block forever?

• How to join infinite streams if join partners can arbitrarily arrive (or not)?

• Idea: keep window that renders a continuous (infinite) stream a snapshot/static relation
Sliding Window Concept

- Focus attention to **latest values of stream**
- Allows computation of **aggregates**
- **Joins** are computed **across windows** overlaid of other (or same) streams
Sliding Window: Example

• Window of size $W$
  – based on time (=> time-based)
  – or number of tuples inside (count-based)
• Shifted every $t$ by $B$
Sliding Window Aggregates

- Output average for each window when it slides.
- Here:
  - 17.6°C
  - 26.3°C
  - 19.1°C
  - 18.3°C
  - 13.5°C
  - 27.0°C
  - 11.6°C
  - 29.6°C
  - 39.7°C
  - 24.2°C
  - 11.5°C
  - 12.7°C
  - 27.9°C
  .....
Sliding Window Joins

- Join is executed over individual window contents.
Types of Sliding Windows

• **Time based Window**
  – window contains tuples within a certain time range; e.g., Twitter Tweets of the last 10 minutes, stock market values of the last 10 seconds
  – size can arbitrarily change if input rate changes

• **Count-based Window**
  – window contains at any time a fixed amount of items, say, the last 100 Tweets or 10000 last stock trades
  – newly arriving items kick out older ones (once window is filled up), depending on strategy (next slide)
Types of Sliding Windows (Cont’d)

- **Sliding Window**: move window on certain ticks/time, continuous or in blocks
- **Tumbling Window**: create new window for each time range of size W (i.e., non overlapping)

  - At each slide/”tumple” a function can be applied to window content and the result outputted
  - This is also called “trigger”.
Overview of DSMSs

- **STREAM** (Stanford University), **Aurora** (Brandeis/Brown/MIT), **TelegraphCQ** (UC Berkely), **Cayuga** (Cornell), **PIPES** (Uni Marburg), ...

- Large interest also from companies/startups: Oracle, Microsoft, IBM, Streambase

- Lately open-source product for big data distributed streams: Yahoo! S4, **Twitter Storm** (will see in more detail later)
StreamBase Example UI

http://www.streambase.com
STREAM

• **Stanford Stream Data Manager**

• “General purpose” DSMS for streams and stored data

• **CQL**: Declarative query language to phrase continuous queries (SQL like).

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.
Example Query 1

• Two streams:
  – 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 [Range 1 Day], 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

**SQL:**

```
SELECT F.clerk, max(O.cost)
FROM orders O,
     fulfillments F
     [PARTITION BY clerk ROW 5] 10% SAMPLE
WHERE O.orderID = F.orderID
GROUP BY F.clerk
```
CQL: Relations and Streams

• **T**: discrete, ordered time domain

• A **relation** $R$ is a mapping from time $T$ to bag of tuples belonging to the schema of $R$.

• That is, $R(t)$ varies over time

• A **stream** is a set of (tuple, timestamp) elements
Streams ↔ Relations

Window specification

Special operators: Istream, Dstream, Rstream

Any relational query language

Slide based on material from Jennifer Widom.
Stream $\rightarrow$ Relation

- $S[W]$ is a relation: at time $T$ it contains all tuples in window $W$ applied to stream $S$, up to time $T$.

- When $W = \infty$, it contains all tuples in stream $S$ up to time $T$.

- Ways to construct these windows “[W]”
  - Time-based
  - Tuple-based
  - Partitioned
Time-Based Window

• S [Range T]
  – S [Now]
  – S [Range Unbounded]

Examples:
• PosSpeedStr [RANGE 30 Seconds]
• PosSpeedStr [NOW]
• PosSpeedStr [RANGE Unbounded]

Note: variable number of records in the window

Stream with vehicle data on highway:
PosSpeedStr(vehicleId,speed,xPos,dir,hwy)

Slide based on material from Jennifer Widom.
Tuple-Based Window

- $S \text{ [Rows } N\text{]}
  - If tuples form a partial order, ties are broken arbitrarily
  - [Rows Unbounded]

Example:

- PosSpeedStr \[\text{ROWS 1}\]

Stream with vehicle data on highway: PosSpeedStr(vehicleId, speed, xPos, dir, hwy)

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:

- \( \text{PosSpeedStr(vehicleId,speed,xPos,dir,hwy)} \)

- \( \text{PosSpeedStr [PARTITION BY vehicleId ROWS 1]} \)

Slide based on material from Jennifer Widom.
With previous window transform we get a relation, now we can apply any query expressed in SQL – just that we deal now with time-varying relations

Example:

• **SELECT distinct vehicleId**
  • **FROM PosSpeedStr [RANGE 30 Seconds]**

*Computes the active vehicles*
Relation $\rightarrow$ Stream

- $\text{Istream}(R)$ contains a stream element $(r,t)$ whenever $r \in R(t) \setminus R(t-1)$ "Insert stream"
- $\text{Dstream}(R)$ contains a stream element $(r,t)$ whenever $r \in R(t-1) \setminus R(t)$ "Delete stream"
- $\text{Rstream}(R)$ contains a stream element $(r,t)$ whenever $r \in R(t)$ "Relation stream"

Bag (Multiset) semantics
Istream, Dstream, and Rstream

• **Istream(R):** contains all tuples in R that are new within the last time period, i.e., insert stream

• **Dstream(R):** contains all tuples in R which were in the stream before the last period (and not anymore in now), i.e., delete stream

• **Rstream(R):** contains all tuples in R

**Note:** Istream and Dstream are expressible with Rstream and suitable selections/windows. How?
Relation $\rightarrow$ Stream: Examples

SELECT $I_{stream}(\ast)$
FROM PosSpeedStr [$\text{RANGE Unbounded}$]
WHERE speed > 65

SELECT $R_{stream}(\ast)$
FROM PosSpeedStr [$\text{NOW}$]
WHERE speed > 65

sliding window that contains only the last (now) tuples; from that instant in time

Slide based on material from Jennifer Widom.
Query Results at Time T

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

```
SELECT F.clerk, max(O.cost)
FROM O [∞], 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
Examples (Cont’d)

SELECT Istream(F.clerk, max(O.cost))
FROM O [∞], 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 result (<clerk, max>, T), whenever <clerk, max> changes from T-1

Slide based on material from Jennifer Widom.
Examples (Cont’d)

• What is the following query doing?

    SELECT Istream(Avg(A)) FROM S [Range 5 seconds]

    Emit 5-second moving average on every timestep, but output is generated only if average changes (Istream!)

• To emit a result on every timestep

    SELECT Rstream(Avg(A)) FROM S [Range 5 seconds]

• To emit a result on every second

    SELECT Rstream(Avg(A)) FROM S
    [Range 5 seconds Slide 1 second]

Slide based on material from Jennifer Widom.
Query Execution in STREAM

- 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
More Topics

• Seen only formal model and standard concepts of data stream management systems
• There is of course much more to it
• Implementation, optimization (e.g., equivalences), load shedding, ...
• Would be an own entire lecture by itself.
• Next, distributed data stream management systems
Query Processing

• Many problems to be addressed resemble conceptually the same issues that arise in traditional RDBMS

• Goals of DSMS are different in many aspects, though.
  – Continuous queries
  – Push-based data model
  – Aim at real-time processing
  – Need for memory efficient algorithms
  – Handle overload to guarantee real-time processing; load shedding
  – Sharing of intermediate results (multi query optimization)
Implementation and Processing

• Query is compiled into **query execution plan** (similar to what is known from RDBMS lectures)

• Recall differences from DBMS and DSMS; data is actively streaming in.

• What does this imply for the implementation?
Push vs. Pull

• Two fundamentally different ways operators (nodes in a query plan) interact

• **Pull**: Consuming operator actively retrieves results of producer.

• **Push**: Producer push results to consumer.
Pull

• We all know that from DBMS (think JDBC or operator trees) or Java Iterators

```java
ResultSet rset = Statement.executeQuery("Select * from ....");
while (rset.next()) {
    rset.getInteger(1);
    ... 
}
```

```
SELECT c.plate, p.lastname
FROM people p JOIN cars c ON p.id=c.owner
WHERE c.plate LIKE 'KL-%'
```

“OPEN, NEXT, CLOSE”
Push

- Stream processing is by design mainly data-driven
- **Operators register at other operators**
- When new tuples are generated, they are actively pushed to registered operators

- Creating a **directed acyclic graph** (DAG), e.g., called **topology** in later system
STREAM: Simple Query Plan

Slide courtesy of Jennifer Widom.
Query Plans in STREAM

- **Operators**
  - do the actual processing;
  - e.g., join, selection, window, ...

- **Queues**
  - connect operators

- **Synopses**
  - store operator states. For instance, the hash table of a hash-based join
Queues

• A queue connects a tuple producing operator $O_P$ and its consuming operator $O_C$

• Conceptually FIFO buffer

• Elements inserted and retrieved in timestamp order

• Shared Queues: multiple consumers for one producer possible
Operator Decoupling

- Queues allow decoupling of operators
- Consumers read from queue
- Producers write to queue
Distributed DSMS

• Conceptually, distributed data stream management systems behave/look like centralized ones

• **STREAM** (seen before)
• **Borealis** (Brandeis U, Brown U, MIT)
• **Global Sensor Networks** (EPFL)
• ...


Karl Aberer et al.: Infrastructure for Data Processing in Large-Scale Interconnected Sensor Networks. MDM 2007: 198-205
Distributed DSMS (Cont’d)

• In spirit of the beginning of the lecture on MapReduce / NoSQL, we look at very recent distributed DSMS for big data (stream) processing
  – Yahoo! S4 (now Apache)
  – Twitter (Apache) Storm

• Many concepts are also generic. Conceptually, e.g., the operator interfaces and topologies.
(Generic) Aims

- Guaranteed data processing
- Fault tolerance
- Horizontal scalability
- Enable high-level programming

• Sounds like MapReduce/Hadoop? Well ...
Apache Storm

- Sometimes referred to as “the realtime Hadoop”
- Fault tolerant, distributed stream processing system. Developed by N. Marz (now Twitter) in 2011
- Widely used by companies
- Data stream operators are (can) be put on different nodes; replicated operators of same kind for scalability.
Trident

• Guess what? **There is a high-level abstraction on top of Storm.**

```java
TridentTopology topology = new TridentTopology();
TridentState wordCounts =
    topology.newStream("spout1", spout)
        .each(new Fields("sentence"),
            new Split(), new Fields("word"))
        .groupBy(new Fields("word"))
    .persistentAggregate(new MemoryMapState.Factory(),
            new Count(), new Fields("count"))
    .parallelismHint(6);
```

https://github.com/nathanmarz/storm/wiki/Trident-tutorial
Literature

- Jürgen Krämer, Bernhard Seeger: Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34(1) (2009)