Data Stream Management

Yanlei Diao
University of Massachusetts Amherst

Slides courtesy of Michael Franklin and Jianjun Chen
Data Stream Management

Two driving forces:

- A collection of applications where data streams naturally exist but DBMS doesn’t help much
- Advances of sensor technologies
Financial Applications

- Financial services
  - Data feeds: stock tickers, foreign exchange transactions…
  - Data rate: 10’s or 100’s thousands of messages per second
  - Applications:
    - routing trade requests,
    - automating trade strategies,
    - market trend analysis…
  - Stream systems: e.g.,
    - http://www.streambase.com
    - http://www.aleri.com
Network and System Monitoring

- **Network monitoring**
  - Packet traces, network performance measurements…
  - Data rate: gigabits per second
  - Applications:
    - traffic analysis, performance monitoring, router configuration, intrusion detection…
  - Stream systems: e.g.,
    - Gigascope at AT&T

- **System/Application monitoring**…
  - Data: system log, measurements
  - Stream systems: e.g.,
    - Ganglia [http://ganglia.info/](http://ganglia.info/)
Wireless Sensor Networks

- Wireless sensor networks
  - Sensor devices: temperature, light, pressure, acceleration, humidity, magnetic field, ...
  - A set of sensor devices auto-configure themselves into a communication network
  - Applications:
    - environment monitoring
    - habitat monitoring
    - structural monitoring
    - vehicle tracking…
Radio Frequency Identification

- **RFID Technology**
  - Tags: small devices, transmitting a tag id when brought close to an RFID reader
  - Readers: interrogate tags with radio signal; read constantly, over a range, without line-of-sight
  - Applications:
    - supply chain management
    - healthcare
    - pharmaceuticals
    - postal services…
  - Stream systems: e.g.,
    - [http://www.truviso.com](http://www.truviso.com)
Execution Models: A Comparison

Traditional DBMS

- Data at rest
- One-time or periodic queries
- Query-driven execution
- Results returned once or periodically

Data Stream System

- Data in motion, unending
- Continuous, long-running queries
- Data-driven execution
- Results returned continuously in real-time
Latency

Value of Data to Decision-Making

- Time-critical Decisions

- Information Half-Life In Decision-Making

- Traditional “Batch” Business Intelligence

- Reactive

- Actionable

- Preventive/Predictive

- Historical

- Real-Time
- Seconds
- Minutes
- Hours
- Time
- Days
- Months

Slide Courtesy of Michael Franklin
History of Stream Processing

- Early: Active DBs, ECA rules, Triggers, Data Broadcast
- Field took off around 2002
  - ~10 SIGMOD/VLDB/ICDE “stream” papers thru 2001
  - 275+ since then (holding steady at 40-50/yr)
- Lots of Research Systems-Building Projects
  - AT&T – Gigascope
  - Berkeley – TelegraphCQ->HIFI
  - Brandeis, Brown, MIT - Aurora->Borealis
  - Purdue – Nile
  - Stanford – STREAM
  - Wisconsin, Portland State – NiagaraCQ
- Lots of Technology Transfer: e.g.,
Main Issues of Stream Management

- Query languages
- Single query processing
- Multi-query optimization
- Adaptive query processing
- Real-time processing
- Approximate answers
- Out of order processing

- Sliding windows
- Non blocking, windowed operations
- Sharing
- Adaptivity
- Quality of service
- Confidence, sketching, sampling, wavelets...
- Punctuation, order-agnostic
Data Stream Model

- A New Data Model: an infinite sequence of elements
  - An element has a system time and an application time
  - Elements arrive in order of system time (arrival order)
  - If an application wants data in application time, need buffering & sorting
  - If data comes out of order or very late, more work is needed to take care the order. (out of order processing)
Stream Query Languages

- Extensions of SQL: SEQUIN [SLR’ 96], CQL [ABW03], TCQ [CCD +03], GSQL [CJS+03]

- Continuous Query Language (CQL)
  - **FROM** can address both streams and relations
  - **PARTITION BY** creates sub-streams
  - **Window**: extracts a set of tuples from an infinite stream and applies relational processing
    - **ROW**, physical window in number of tuples;
    - **RANGE**, logical window in terms of seconds, minutes, or hours (more generally, positions in an ordered domain)
    - **SLIDE**, window movement.
Examples of CQL

**Sliding window join:**

```
Select  * 
From    S1 [Rows 5], S2 [Rows 10]
Where   S1.a = S2.a
```
Example Windows

**Tumbling**: Non-overlapping windows, e.g., 5 second windows  
<Range 5 seconds  Slide 5 seconds>

**Sliding**: Overlapping windows, e.g., 5 second windows move every 2 seconds  
<Range 5 seconds Slide 2 seconds>

**Landmark**: Growing windows, e.g., extend window every 2 seconds  
<LANDMARK  ADVANCE 2 sec> [Truviso Language]
Example Windows

**Partitioned**: value-based sub-streams

\[<\text{Partition By attr Rows } 2 \text{ Slide } 2>\] partitioned tumbling windows

Slide Courtesy of Michael Franklin
Formal Semantics

- At time $t$, there is a window $R_t$, run query to obtain $Q(R_t)$, $t=1,2,3\ldots$

- Collapse the results at adjacent time points based on identical values

- Use Istream and Dstream to ensure that the output also conforms to the stream model
Main Issues of Stream Management

- Query languages
- Single query processing
- Multi-query optimization
- Adaptive query processing
- Real-time processing
- Approximate answers
- Out of order processing

- Sliding windows
- Non blocking, windowed operations
- Sharing
- Adaptivity
- Quality of service
- Confidence, sketching, sampling, wavelets...
- Punctuation, order-agnostic
Single Query Processing

- **Blocking query operator**
  
  "A query operator that is unable to produce the first tuple of its output until it has seen its entire input." [BBD+02]
  
  - Sorting?
  - Sort-merge join?
  - Nested-loops join?
  - Hash join?

- **Non blocking query operator**
  
  "The operator does not stage data (either in memory or on disk) without producing results for a long time." [UF01]
  
  - produces the initial result early
  - returns result tuples incrementally as they become available
(1) Join: First A Blocking Operator

- Assume first that each stream is finite and fits in memory.
- Traditional Hash Join blocks when one input stalls.
Non-Blocking Join Operator

- **Symmetric Hash Join** processes tuples as they arrive.
- Produces all tuples in the join and no duplicates.
- Blocks only if both inputs stall.

Symmetric Hash Join

<table>
<thead>
<tr>
<th>Hash Table A</th>
<th>Hash Table B</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source A</td>
<td>Source B</td>
</tr>
</tbody>
</table>

Build

Probe
Non Blocking Join with Limited Memory

- SHJ requires both inputs to be memory resident.
  - Unrealistic in stream processing

- **XJoin** extends SHJ to work for long (but finite) streams with limited memory [UF00].
  - Partition each input using a hash function.
  - When allocated memory is used up, flush a partition to disk. At each point, a partition = disk-resident data (older) + memory-resident data (newer).
  - Join processing continues on memory-resident data.
  - Disk-resident tuples are handled in background.
XJoin: Handling the Partitions

Memory-resident partitions of source A

Memory-resident partitions of source B

Disk-resident partitions of source A

Disk-resident partitions of source B

SOURCE-A

SOURCE-B

hash(Tuple A) = 1

hash(Tuple B) = n
Stage 1: Memory-to-Memory Joins

Partitions of source A

Partitions of source B

Tuple A
hash(record A) = i

SOURCE-A

Tuple B
hash(record B) = j

SOURCE-B

Output
Stage 2: Disk-to-Memory Joins

Partitions of source A

Partitions of source B

Output
Three Stages of XJoin

- **Stage 1** - Symmetric hash join (memory-to-memory) with partitions

- **Stage 2** - Disk-to-memory
  - Separate thread - runs when stage 1 blocks (both input stall).
  - Stage 1 and 2 interleave until all tuples have been received.

- **Stage 3** - Clean up stage
  - Stage 1 misses pairs that were not in memory concurrently.
  - Stage 2 misses pairs when both are on disk, and may not get to run to completion.
  - Stage 3 joins all the partitions (memory-resident and disk-resident portions) of the two sources.
Windowed Joins

- **Windowed joins**
  - Use windows to obtain a finite set from an infinite stream
  - Basis is non-blocking symmetric hash join
  - Add support for windows
    - Window size: in number of tuples or logical time
    - Sliding of window: smooth movement or hopping
    - Maintain right state (tuples residing in the window) for the join; need to prune **expired tuples** from hash tables!

- [MAF+03, GO03, KNV03, MLA04]
(2) Windowed Aggregates

- Windowed aggregates:
  - Use windows to obtain a finite set from an infinite stream
  - Apply MIN, MAX, SUM, COUNT, or AVG to each window
  - Avoid scanning all tuples in each window for time efficiency
  - Avoid storing all tuples in each window for space efficiency
  - What is the time/space complexity for each aggr operator?

- Windowed SUM/COUNT
  - Symmetry between expired tuples and new tuples [SLR96]

- Windowed MIN/MAX/TOP-K
  - A naïve solution may have to store all tuples in each window
  - How can we do better?
  - [LMT+05, KWF+06, MBP+06, ZYC+07, YSR+11]
“Predictability” Property of Windows

- When a tuple arrives, can predict
  1) to which future windows the tuple will belong
  2) whether the tuple belongs to the partial result (e.g., top-k) of each window

- These insights lead to a “predicted” top-k result for each of the future windows

- When a new tuple arrives,
  - it belongs to its own set of current and future windows
  - it is also used to update the previously “predicted” top-k result for each relevant window
Figure 1: (Predicted) Top-k results four consecutive windows at time of \( W_0 \) (slide size = 4 objects)

Figure 2: Updated predicted top-k results of four consecutive windows at time of $W_1$ (slide size = 4 objects)