Systems Infrastructure for Data Science

Web Science Group Uni Freiburg WS 2014/15

Data Stream Processing

Topics

- Model Issues
- System Issues

Data Streams

- Continuous sequences of data elements that are typically:
 - Push-based (data flow controlled by sources)
 - Ordered (e.g., by arrival time, or by explicit timestamps)
 - Rapid (e.g., ~ 100K messages/second in market data)
 - Potentially unbounded (may have no end)
 - Time-sensitive (usually representing real-time events)
 - Time-varying (in content and speed)
 - Unpredictable (autonomous data sources)

Example Applications

• Financial Services



Example:

 Trades(time, symbol, price, volume)

Typical Applications:

- Algorithmic Trading
- Foreign Exchange
- Fraud Detection
- Compliance Checking

Financial Services: Skyrocketing Data Rates

OPRA Message Traffic Projections



Date

[Source: Options Price Reporting Authority, http://www.opradata.com]

Some more up-to-date rates from <u>http://www.marketdatapeaks.com/</u>:

- 4 M mps on January 25, 2013
- 6.65 M mps on October 7, 2011

Low response time critical (think high frequency trading)!

Example Applications

System and Network Monitoring



Example:

Connections(time, srcIP, destIP, destPort, status)

Typical Applications:

- Server load monitoring
- Network traffic monitoring
- Detecting security attacks
 - Denial of Service
 - Intrusion

7

Systems Infrastructure for Data Science

Network Monitoring: Bursty Data Rates



[Source: Internet Traffic Archive, http://ita.ee.lbl.gov/]

Example Applications

Sensor-based Monitoring



Example:

CarPositions(time, id, speed, position)

Typical Applications:

- Monitoring congested roads
- Route planning
- Rule violations
- Tolling

Historical Background

- 1990s: Various extensions to traditional database systems
 - Triggers in Active DB's, Sequence DB's, Continuous Queries, Pub/Sub, etc.
- Early 2000s: Data Stream Management Systems
 - Aurora [Brandeis-Brown-MIT]
 - STREAM [Stanford]
 - TelegraphCQ [UC Berkeley]
 - Many others (NiagaraCQ, Gigascope, Nile, PIPES, ...)
- 2003: Start-ups
 - Aurora -> StreamBase, Inc.
 - -> Borealis (= distributed Aurora)
 - STREAM -> Coral8, Inc.
- 2005: More Start-ups
 - TelegraphCQ -> Truviso, Inc.
- Today: Growing industry interest, open source platforms and standardization efforts

A Paradigm Shift in Data Processing Model



Traditional Data Management

Data Stream Management

DBMS vs. DSMS

- Persistent relations
- Read-intensive
- One-time queries

- Random access
- Access plan determined by query processor and physical DB design

- Transient streams
- Update-intensive
- Continuous queries (a.k.a., long-running, standing, or persistent queries)
- Sequential access
- Unpredictable data characteristics and arrival patterns

Model Issues

- Data models
 - Relational-based vs. XML-based vs Object-based
 - Time, Order and Unboundedness
- Query models
 - Declarative vs. Procedural
 - Window-based Processing

Example Models

- STREAM / CQL [*Stanford*]
 - Relational-based data model
 - Declarative query language (SQL extensions)
- Aurora / SQuAl [Brandeis-Brown-MIT]
 - Relational-based data model
 - Procedural query language (Relational algebra extensions)
- MXQuery [ETH Zurich]
 - XML-based data model
 - Declarative query language (XQuery extensions)

Window-based Processing

- Windows are <u>finite excerpts</u> of a potentially unbounded stream.
- Most streaming applications are interested in the readings of the <u>recent past</u>.
- Windows help us <u>unblock operators</u> such as aggregates.
- Windows help us <u>bound the memory usage</u> for operators such as joins.

Window Example

- Two basic parameters: size and slide
- Example: Trades (time, symbol, price, volume)



Windows: Unblocking Aggregate Operation



Windows: Bounding Join State



STREAM CQL: Continuous Query Language

- SQL for Relation-to-Relation operations
- Additionally:
 - "Stream" as a new data type (in addition to "Relation")
 - Continuous instead of one-time query semantics
 - Stream-to-Relation operations:
 - Window specifications derived from SQL-99
 - Relation-to-Stream operations:
 - Three special operators: Istream, Dstream, Rstream
 - Simple sampling operations on streams

CQL: Streams vs. Relations

- T: discrete, ordered time domain
- A stream S is a possibly infinite bag of elements <s, t>, where s is a tuple with the schema of S and t

 T is the timestamp of the element.
 - Note: Timestamp is not part of the tuple schema!
- A relation R is a mapping from each time instant in T to a finite but unbounded bag of tuples with the schema of R.

CQL: Continuous Query Semantics

- Time "advances" from t-1 to t, when all inputs up to t-1 have been processed.
- For a query producing a stream:
 - At time t ext{ T, all inputs up to t are processed and the continuous query emits any new stream result elements with timestamp t.
- For a query producing a relation:
 - At time t ext{ T, all inputs up to t are processed and the continuous query updates the output relation to state R(t).

CQL: Mappings between Streams and Relations



Stream-to-Stream = Stream-to-Relation + Relation-to-Stream

CQL: Stream-to-Relation Operators

- Time-based sliding windows
 - FROM S[RANGE T]
- Tuple-based sliding windows
 - FROM S[ROWS N]
- Partitioned windows
 - FROM S[PARTITION BY A₁, ..., A_k RANGE T]
 - FROM S[PARTITION BY A₁, ..., A_k ROWS N]
- Windows with a "slide" parameter
 - FROM S[RANGE T SLIDE L]
 - FROM S[ROWS N SLIDE L]
 - FROM S[PARTITION BY A₁, ..., A_k RANGE T SLIDE L]
 - FROM S[PARTITION BY A₁, ..., A_k ROWS N SLIDE L]

CQL: Relation-to-Stream Operators

Insert stream

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

• Delete stream

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

Relation stream

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

• SELECT Istream(..), SELECT Dstream(..), SELECT Rstream(..)

CQL: Example Queries

Trades (time, symbol, price, volume) NYSE_Trades (time, symbol, price, volume) SWX_Trades (time, symbol, price, volume)

Streaming Filter

SELECT Istream(*) FROM Trades[RANGE Unbounded] WHERE price > 20

Streaming Aggregation SELECT Istream(Count(*)) FROM Trades[PARTITION BY symbol RANGE 10 Minutes SLIDE 1 Minute]

Sliding-window Join

SELECT Istream(*) FROM NYSE_Trades[RANGE 10 Minutes], SWX_Trades[RANGE 10 Minutes] WHERE NYSE_Trades.symbol = SWX_Trades.symbol

CQL: Example Query Execution

Query:
SELECT Istream(*)
FROM S[ROWS 1]
WHERE <Filter>

Stream: S(A)

 Operations: LastRow: S-to-R Filter: R-to-R Istream: R-to-S

Assumption:
(a₀), (a₂), (a₄)
satisfy the filter.

Time	S	LastRow	Filter	Istream
0	$\langle (a_0), 0 \rangle$	(a_0)	(<i>a</i> ₀)	$\langle (a_0), 0 \rangle$
1	$\langle (a_0), 0 \rangle$ $\langle (a_1), 1 \rangle$	(a_1)	ϕ	$\langle (a_0), 0 \rangle$
2	$ \begin{array}{c} \langle (a_0), 0 \rangle \\ \langle (a_1), 1 \rangle \\ \langle (a_2), 2 \rangle \end{array} $	(<i>a</i> ₂)	(a ₂)	$\langle (a_0), 0 \rangle$ $\langle (a_2), 2 \rangle$
3	$ \begin{array}{c} \langle (a_0), 0 \rangle \\ \langle (a_1), 1 \rangle \\ \langle (a_2), 2 \rangle \\ \langle (a_3), 3 \rangle \end{array} $	(<i>a</i> ₃)	ϕ	$\langle (a_0), 0 \rangle$ $\langle (a_2), 2 \rangle$
4	$ \begin{array}{c} \langle (a_0), 0 \rangle \\ \langle (a_1), 1 \rangle \\ \langle (a_2), 2 \rangle \\ \langle (a_3), 3 \rangle \\ \langle (a_4), 4 \rangle \end{array} $	(<i>a</i> ₄)	(a 4)	$\langle (a_0), 0 \rangle$ $\langle (a_2), 2 \rangle$ $\langle (a_4), 4 \rangle$
:	:	:	:	:

Aurora SQuAI: Stream Query Algebra

- A stream is an append-only sequence of tuples with a uniform schema.
- The system stamps each tuple with its time of arrival.
- Disorder is allowed.
- Queries are represented with data-flow diagrams consisting of operators.
- Order-agnostic operators:
 - Filter, Map, Union
- Order-sensitive operators:
 - BSort, Aggregate, Join, Resample

SQuAI: Operators

- Filter applies a predicate on each stream tuple.
- Map applies a function on each stream tuple. (* extensibility)
 e.g., projection
- Union merges two or more streams into one.
 - "order-preserving" version also exists.
- **BSort** is a buffer-based approximate sort.
 - equivalent to n-pass bubble sort
- Aggregate applies window functions to sliding windows over its input. (* extensibility)
- Join applies a predicate to pairs of tuples from two input streams that are within a certain window distance from each other.
- **Resample** applies an interpolation function on a stream to align it with another stream.

SQuAI: Example Query



- Boxes and arrows data-flow diagram instead of a declarative specification.
- Same query can also be written in STREAM CQL as a nested query.

SQuAI: Slack & Timeout Parameters

- **Slack** is a stream parameter to specify the degree of disorder in that stream.
 - Out of order tuples beyond the slack parameter are simply discarded.
- Timeout is a parameter for sliding window operators to specify the maximum time period that a window is allowed to remain open.
 - Delayed tuples beyond the timeout parameter are simply discarded.

Streaming XQuery

- Extend existing turing-complete processing language
- Benefit: Data Model already sequence-based, no mapping needed
- Extend for infinite sequences, define formal semantics for existing operators
- Define predicate-based window operator to produce finite sequences, can be fully nested
- Time not part of data model, operate on item values
- No implicit constraints
- Limitation: FLWOR semantics difficult for join

Common Window Types

- Sliding window
 - A window that slides (i.e., both of its end-points move) as new stream tuples arrive.
- Tumbling window
 - A sliding window for which window size = window slide (i.e., consecutive windows do not overlap).
- Landmark window
 - A window which is moving only on one of its endpoints (usually the forward end-point).

Common Window Types

- Time-based window
 - A window whose size and content is determined by tuples that arrived within a "time period".
 - Note: The actual size of such a window may depend on the stream arrival rate.
- Tuple-based window (a.k.a., count-based window)
 - A window whose size and content is determined by the number of tuples arrived.
 - Note: The actual size is always fixed.
- Semantic window (a.k.a., predicate-based window)
 - A window whose size and content is determined by the tuple contents.
 - Note: Time-based window is a very simple form of semantic window when the time field carried in the tuple is used for windowing.

A Final Note on Window Execution Semantics

- Currently, there is no standard model for defining and executing stream windows.
 - Example: Even "time-based window" works differently in different systems, producing different query results.
- Example differentiators:
 - What triggers window state change? (e.g., time in STREAM vs. tuple arrival in Aurora)
 - When is a window result reported? (e.g., at window close in Aurora vs. at each window state change in Coral8)

Time in DSMS

- "A window of 30 seconds, starting every 5 seconds"
- What is the precise meaning of these time values?
- Two main approaches to handle time:
 - System Time: take 30 seconds of execution time
 - Application Time: 30 seconds of data time fields
- System Time leads to non-determistic results
- Application Time might cause system-time delays
- => Heartbeats to synchronize
- Application Time desirable, in practice often system time
- Other time aspects:
 - Point in Time or Time Period
 - Start, End, ...

Stream Constraints

- Metadata about streams that can be used for their optimized processing, in particular:
 - to reduce, bound, eliminate memory state
 - could be an alternative to windowing
- Metadata can be affect to static and dynamic parts of stream processing
- Schema-level constraints
 - Clustering (e.g., contiguous duplicates)
 - Ordering (e.g., slack parameter in SQuAI)
 - Referential integrity (e.g., timestamp synchronization)
 - In relaxed form: k-constraints (k: adherence parameter)
- Data-level constraints
 - Punctuations
 - Partitions
 - Pattern

Punctuations

- Punctuations are special annotations embedded in data streams to specify the end of a subset of data.
 No more tuples will follow that match the punctuation.
- A punctuation is represented as an ordered set of patterns, where each pattern corresponds to an attribute of a tuple.

Patterns: *, constants, ranges [a, b] or (a b), lists {a, b, ..}, Ø
Example: < item id, buyer id, bid >

< {10, 20}, *, * > => all bids on items 10 and 20.