

CMSC 724: Data Streams

Amol Deshpande

University of Maryland, College Park

April 12, 2007

Data Streams

- Why ?
- Much data generated continuously (growing every day)
 - Financial data
 - Sensors, RFID
 - Network/systems monitoring
 - Video/Audio data
 - etc ...
- Need to support:
 - High data rates
 - Real-time processing with low latencies
 - Support for temporal reasoning (time-series operations)
 - Data dissemination
 - Distributed ? (at least data generation)
 - etc...

Examples of Tasks

- **Continuous** (SQL) queries
 - E.g. moving average over last hour every 10 mins
 - SQL extended to support "windows" over streams
 - Proposed extensions: SEQUENCE, CQL, StreamSQL
- Pattern recognition
 - Alert me when: *A*, then *B* within 10 mins
 - How to specify ? StreamSQL has some support
- Probabilistic modeling; Applying financial models
 - Infer hidden variables
 - Remove noise (from measured readings)
 - Do complex analysis to decide whether to *buy*
 - We don't even know how to specify these
- Multimedia data ?
 - Online object detection, activity detection
 - Correlating events from different streams

Data Streams

- Use traditional DBMS ?
- Consider simplest case:
 - Report moving average over last hour every 10 minutes
 - 1. Insert all new items into database
 - 2. Execute the query every 10 minutes
- Not easily generalizable to other tasks
 - E.g. "alert me the moment moving average > 100 " ?
- Typically 1000's of such continuous queries
- Even for one query, too slow and inefficient
 - Doesn't reuse work from previous execution
- Application-level modules typically used for complex tasks

Data Streams

- Triggers ?
 - Similar, but current trigger systems not designed for the required scale
- Publish-Subscribe Systems
 - Similar concepts: Push-based, reactive execution
 - Typically no complex queries
 - Much focus on “dissemination”
- Major research systems (late 90's-early 00's):
 - NiagaraCQ (Wisc), Telegraph, TelegraphCQ (Berkeley)
 - STREAM (Stanford), Aurora, Borealis, Medusa (Brown/Brandeis/MIT)
- Commercial
 - Oracle*Streams, Strembase etc...

Scalable Trigger Processing (Hanesn et al.)

- Goal: Handle millions of triggers
- Triggers: Commonly used for integrity constraint checking, alerts etc. . .

```
create trigger IrisHouseAlert
on insert to house
from salesperson s, house h, represents r
when s.name = 'Iris' and s.spno=r.spno and
r.nno=h.nno
do raise event
NewHouseInIrisNeighborhood(h.hno, h.address)
```

Scalable Trigger Processing (Hanesn et al.)

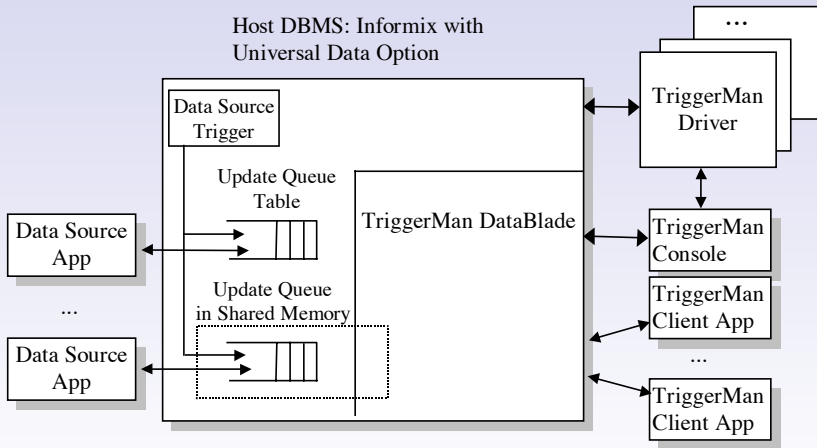
- Goal: Handle millions of triggers
- Triggers: Commonly used for integrity constraint checking, alerts etc. . .

```
CREATE TABLE empauditlog (  
    audit_date      DATE,  
    audit_user      VARCHAR2(20),  
    audit_desc      VARCHAR2(20)  
);  
CREATE OR REPLACE TRIGGER emp_audit_trig  
    AFTER INSERT OR UPDATE OR DELETE ON emp  
DECLARE  
    v_action        VARCHAR2(20);  
BEGIN  
    IF INSERTING THEN  
        v_action := 'Added employee(s)';  
    ELSIF UPDATING THEN  
        v_action := 'Updated employee(s)';  
    ELSIF DELETING THEN  
        v_action := 'Deleted employee(s)';  
    END IF;  
    INSERT INTO empauditlog VALUES (SYSDATE, USER,  
        v_action);  
END;
```

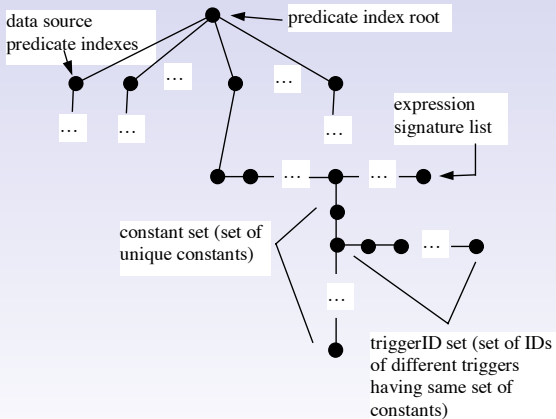
Scalable Trigger Processing (Hanesn et al.)

- Approach:
 - Identify unique “expression signatures” (based on data sources and attributes involved)
 - Group the triggers into “equivalence” classes based on their signatures
 - Use efficient main memory data structures to quickly find triggers that match
- Many similarities to AI Rule systems

Triggerman



Triggerman



Triggers

- Precursor to data streams work
- Event-driven as opposed to query driven
- Can handle pub-sub applications well
- Can identify quickly queries that should be executed
- But, no discussion on how to execute those queries efficiently
 - E.g. “moving average” query
 - Every new tuple will satisfy the query
 - Trigger action (compute moving avg) will be invoked per new tuple
 - No sharing of work from previous execution
 - No sharing of work between multiple triggers
 - E.g. If one person wants moving average over last hour, other person over last two hours

Data Streams: Some Major Concepts

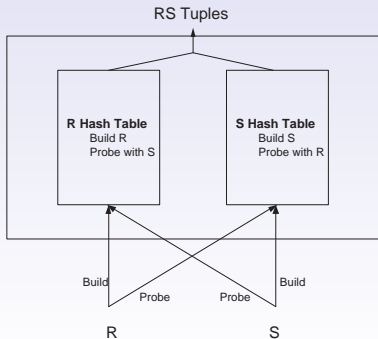
- New non-blocking operators
 - Symmetric hash join, MJoin, XJoin, Eddy etc...
- Adaptivity
 - Dealing with unpredictability
- Sharing/Multi-query optimization
 - 1000's of queries; must share execution
- Load shedding
 - Bursty data: Too much to handle at some times
- Declarative languages
 - Especially for pattern recognition, modeling etc
- Theoretical developments
 - "One-pass" algorithms

Query execution

- Duality between queries and data
 - Traditional: Apply queries to data
 - Streams: Apply data to queries
- New operators
 - Symmetric hash join, XJoins
 - MJoin
- Predicate indexes
- Push vs Pull Execution
- Execution using a router
 - E.g. using an *eddy*

Query execution: Symmetric Hash Join

- Handles updates from both tables
- Produces results immediately
- Can be extended to handle “sliding windows”
 - Remove tuples that fall out
- XJoin: Extends to handle memory limits



Query execution: Eddy

- A *tuple router* that sits between all operators
- Makes decisions of where to send the tuples
- Can encapsulate the logic, so easier to handle these things
- Tricky to reason about in many cases
- Break into two pieces:
 - Mechanism: Set limits on what an eddy can do
 - Where not-to route etc...
 - Policy: Do optimizations within the limits
 - For efficiency etc...

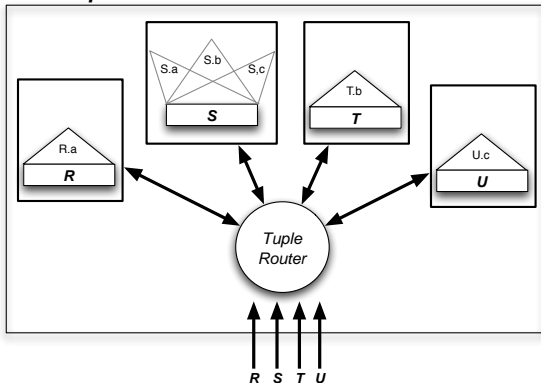
Query Execution: MJoin

- Extension of symmetric hash join to multiple inputs
- Quite useful in stream processing
- Optimization policies; Comparisons to using binary joins

Example Query

```
SELECT *  
FROM R, S, T, U  
WHERE R.a = S.a  
      AND S.b = T.b  
      AND S.c = U.c
```

MJoin Operator



Aurora: Monitoring Streams

- Brown/Brandeis/MIT
- Monitoring applications over continuous data streams
 - Alerts/triggers first class objects
 - Query processing over history/time-series management
 - Poor support in most DBMSs
 - Latency-critical
 - If can't provide data right now, no use
 - Aggressive use of approximate query processing, load shedding etc. . .
 - Quality of service ??

Aurora

- A data-flow system: *boxes and arrows* paradigm
- Operators/boxes:
 - windowing ops: *slide, tumble, latch, resample*
 - relational ops: *select/filter, join, groupby*
 - arbitrary op: *map*
 - Join: only on time-stamps; not arbitrary joins
 - Can simulate by cross-product followed by filter
- Quality of service fundamentally integrated

Aurora

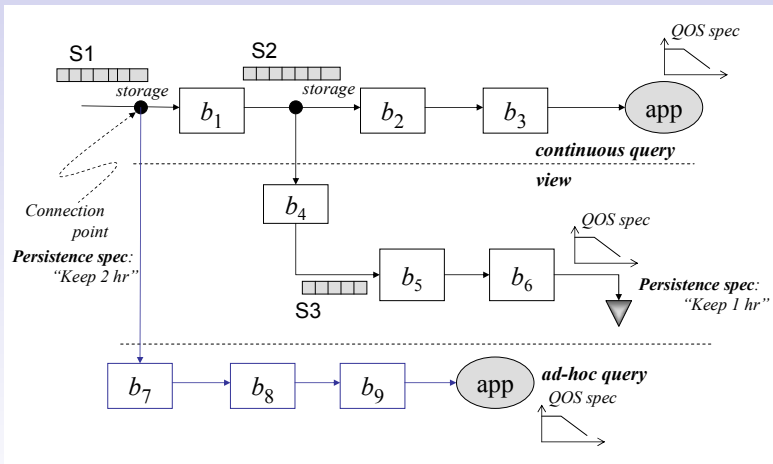


Figure 2: Aurora query model

- Processing tasks/queries
 - Continuous queries: output continuously desired
 - Views
 - Ad hoc queries
- Queries specified using a graphical user interface
- Declarative query language ?
- Optimization:
 - Divide into subnetworks at *connection points*
 - Reoptimize each subnetwork independently

- QoS: delay-based vs drop-based vs value-based
- Aside - Value-based Preferences:
 - CONTROL: Interactive query processing [Hellerstein et al.]
 - Interesting issues in pushing user preferences down the query tree
- QoS: Actual adoption somewhat unclear
 - Users typically only specify “step” functions
 - Too complex otherwise
 - Hard to compare across users etc. . .
 - Dropping may not be acceptable for many applications

Aurora: Query Processing

- Scheduler/router controls what gets processed
- Based on QoS/Priorities
- Uses “batch” (train) scheduling for performance (as with later implementations of eddies)
 - To exploit intra-box and inter-box non-linearity
- Load shedding
 - Drop tuples if too much workload
 - As close to the data sources as possible

Other systems

- NiagaraCQ (Wisconsin)
 - Early work on data streams
- TelegraphCQ (Berkeley)
 - Based on eddies; implemented in PostgreSQL
 - Focus on adaptivity and sharing issues
 - Declarative querying interface: SQL-type
- STREAM (Stanford)
 - Addressed many issues including optimization, language design, approximate query answering, memory constraints etc. . .
- Much other work..

Brief Aside: Sketches

- One-pass algorithms: You can only look at each data item once
- Goal: Compute some aggregate of interest
- Question: What is the amount of space needed if the data size is N ?
 - For exact or *approximate* computation
- Examples:
 - 1. Average: $O(1)$ (number of entries, total sum)
 - 2. Median:
 - Exact: Space complexity = N
 - Approximate: $O(\frac{1}{\epsilon} \log^2 \epsilon N)$, with ϵ error

AMS Sketches

- Alon, Matias, Szegedy: Space Complexity of Approximating the Frequency Moments; STOC 1996
- Consider a stream: (1, 2, 3, 1, 5, 2, 1, 3, 4)
- Let m_i be the frequency of i in the stream
 - $m_1 = 3, m_2 = m_3 = 2, m_4 = m_5 = 1.$
- Frequency moment $F_k = \sum_{i=1}^n m_i^k$
 - $F_0 = 5 =$ number of distinct elements in the stream
 - $F_1 = 9 =$ total number of elements in the stream
 - $F_2 = 19 =$ comes in up many places (e.g. self-join size of a relation)
- How to compute ?
 - Exact computation: $O(n)$, where n is the number of distinct elements, not the size of stream
 - Approximate: AMS Result: Can approximate F_0, F_1, F_2 in logarithmic space, requires $O(n^{\Omega(1)})$ space for others