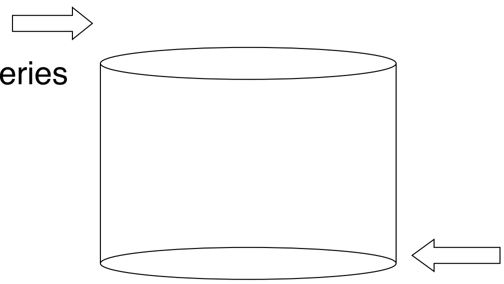
Outline

- Introduction & architectural issues
- Data distribution
- Distributed query processing
- Distributed query optimization
- Distributed transactions & concurrency control
- Distributed reliability
- Data replication
- Parallel database systems
- Database integration & querying
- Peer-to-Peer data management
- ❑ Stream data management
  - ❑ Stream architecture
  - ❑ Query processing
- ❑ MapReduce-based distributed data management

Inputs & Outputs

- Inputs: One or more sources generate data continuously, in real time, and in fixed order
  - Sensor networks – weather monitoring, road traffic monitoring, motion detection
  - Web data – financial trading, news/sports tickers
  - Scientific data – experiments in particle physics
  - Transaction logs – telecom, point-of-sale purchases
  - Network traffic analysis (IP packet headers) – bandwidth usage, routing decisions, security
- Outputs: Want to collect and process the data on-line
  - Environment monitoring
  - Location monitoring
  - Correlations across stock prices
  - Denial-of-service attack detection
- Up-to-date answers generated continuously or periodically

Traditional Database Management System (DBMS)



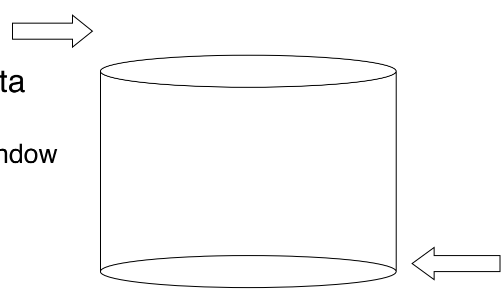
Transient queries

- issued once,
  then forgotten

Persistent data

- stored until deleted
  by user or application

Data Stream Management System (DSMS)



Transient data

- deleted as window
  slides forward

Persistent queries

- generate up-to-date
  answers as time goes on

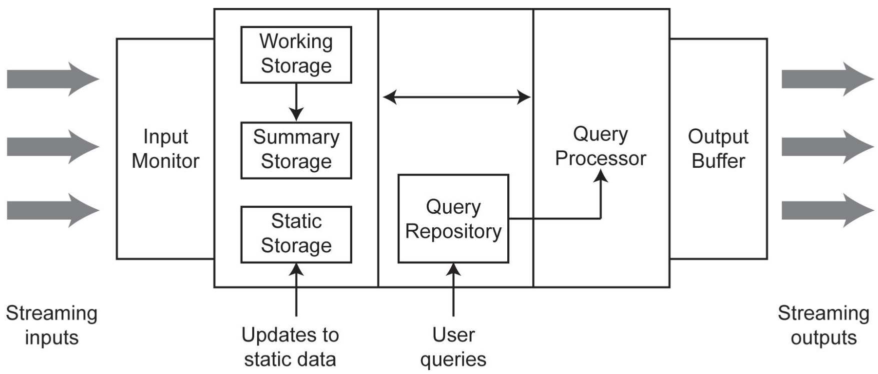
DSMSs – Novel Problems

- Push-based (data-driven), rather than pull-based (query-driven) computation model
  - New data arrive continuously and must be processed
  - Query plans require buffers, queues, and scheduling mechanisms
  - Query operators must be non-blocking
  - Must adapt to changing system conditions throughout the lifetime of a query
  - Load shedding may be required if the system can't keep up with the stream arrival rates

DSMS Implementation Choices

- Application on top of a relational DBMS
  - Application simulates data-driven processing
  - Inefficient due to the semantic gap between the DBMS and the DSMS-like application
- Use advanced features of the DBMS engine
  - Triggers, materialized views, temporal/sequence data models
  - Still based upon query-driven model, triggers don't scale and are not expressive enough
- Specialized DSMS
  - Incorporate streaming semantics and data-driven processing model inside the engine

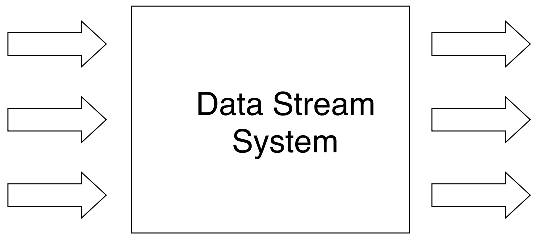
Abstract System Architecture



Stream Data Models

- Append-only sequence of timestamped items that arrive in some order.
- More relaxed definitions are possible
  - Revision tuples
  - Sequence of events (as in publish/subscribe systems)
  - Sequence of sets (or bags) of elements with each set storing elements that have arrived during the sameunit of time.
  - …
- Possible models
  - Unordered cash register
  - Ordered cash register
  - Unordered aggregate
  - Ordered aggregate

Processing Model



- Stream-in-stream-out
- Problem:
  - Streams have unbounded length (system point of view)
  - New data are more accurate/interesting (user point of view)
- Solution:
  - Windows

Windows

- Based on direction of movement of endpoints
  - Two endpoints can be fixed, moving forward, or moving backward
  - Nine possibilities, interesting ones
    - Fixed window
    - Sliding window
    - Landmark window
- Based on direction of window size
  - Logical (or time-based) window
  - Physical (or count-based) window
  - Predicate window
- Based on windows within windows
  - Elastic window
  - N-of-N window
- Based on window update interval
  - Jumping window
  - Tumbling window

Stream Query Languages

- Queries are persistent
- They may be monotonic or non-monotonic
  - Monotonic: result always grows
    - If $Q(t)$ is the result of a query at time t, given two executions at time t_i and t_j, $Q(t_i) \subseteq Q(t_j)$ for all $t_i > t_j$
  - Non-monotonic: deletions from the result are possible
- Monotonic query semantics:
  - $Q(t) = \bigcup_{t_i=1}^{t} (Q(t_i) - Q(t_{i-1})) \cup Q(0)$
- Non-monotonic query semantics:
  - $Q(t) = \bigcup_{t_i=0}^{t} Q(t_i)$

Declarative Languages

- Syntax similar to SQL + window specifications
- Examples: CQL, GSQL, StreaQuel
- CQL
  - Three types of operators:
    - Relation-to-realtion
    - Stream-to-relation
    - Relation-to-stream
  - Join of one-minute windows on the a-attribute:

```
SELECT *
FROM   S1 [RANGE 1 min], S2 [RANGE 1 min]
WHERE  S1.a=S2.a
```

  - ROWS for count-based windows, RANGE for time-based windows

Declarative Languages (cont'd)

- GSQL
  - Input and output are streams (composability)
  - Each stream should have an ordering attribute (e.g., timestamp)
  - Subset of operators of SQL (selection, aggregation with group-by, join)
  - Stream merge operator
  - Only landmark windows, sliding windows may be simulated
- StreaQuel
  - SQL syntax
  - Query includes a for-loop construct with a variable *t* that iterates over time
  - Sliding window over stream *S* with size 5 that should run for 50 time units:

```
for(t=ST; t<ST+50; t++)
   WindowIs(S, t-4, t)
```


Object-based Languages

- Use abstract data typing and/or type hierarchies
- Examples: Tribeca, Cougar
- Tribeca
  - Models stream contents according to a type hierarchy
  - SQL-like syntax, accepts a stream as input and generates one or more output streams
  - Operations: projection, selection, aggregation (over the entire input stream or over a sliding window), multiplex and demultiplex (corresponding to union and group-by)
- Cougar
  - Model sources as ADTs
  - SQL-like syntax + `$every()` clause to specify re-execution frequency

Procedural Languages

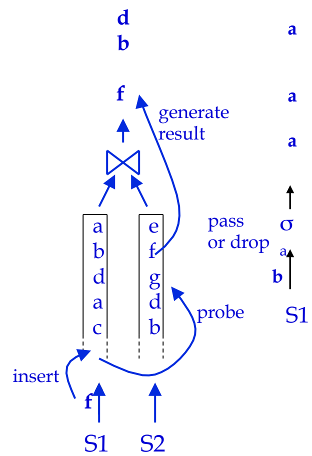
- Let the user specify how the data should flow through the system
- Example: Aurora
- Aurora
  - Accepts streams as inputs and generates output streams
  - Static data sets may be incorporated into query plans via connection points
  - SQuAl algebra
    - Seven operators: projection, union, map, buffered sort, windowed aggregate, binary band join, resample
  - Interface includes
    - Boxes that correspond to operators
    - Edges that connect boxes that correspond to data flow
    - User creates the execution plan

Comparison of Languages

Language/ System	Allowed inputs	Allowed outputs	Novel operators	Supported windows	Execution frequency
CQL/ STREAM	Streams and relations	Streams and relations	Relation-to-stream, stream-to-relation	Sliding	Continuous or periodic
GSQL/ Gigascope	Streams	Streams	Order-preserving union	Landmark	Periodic
StreaQuel/ TelegraphCQ	Streams and relations	Sequences of relations	WindowIs	Fixed, landmark, sliding	Continuous or periodic
Tribeca	Single stream	Streams	Multiplex, demultiplex	Fixed, landmark, sliding	Continuous
SQuAl/ Aurora	Streams and relations	Streams	Resample, map, buffered sort	Fixed, landmark, sliding	Continuous or periodic

Operators over Unbounded Streams

- Simple relational operators (selection, projection) are fine
- Other operators (e.g., nested loop join) are blocking
  - You need to see the entire inner operand
- For some blocking operators, non-blocking versions exist
  - Symmetric hash join

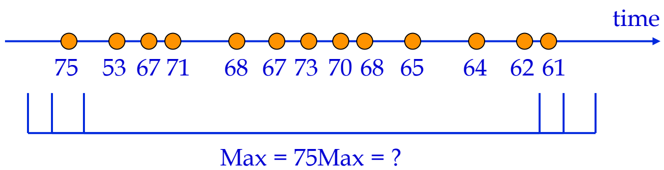


Blocking Operators

- Alternatives if no non-blocking version exists
  - Constraints over the input streams
    - Schema-level
    - Data-level
      - Punctuations
  - Approximation
    - Summaries
      - Counting methods
      - Sketches
  - Windowed operations

Operators over Sliding Windows

- Joins and aggregation may require unbounded state, so they typically operate over sliding windows
- E.g., track the maximum value in an on-line sequence over a sliding window of the last N time units



Operators over Sliding Windows

- Issues
  - Need to store the window so that we "remember what to forget" and when
  - Need to undo previous results by way of negative tuples

Query Processing

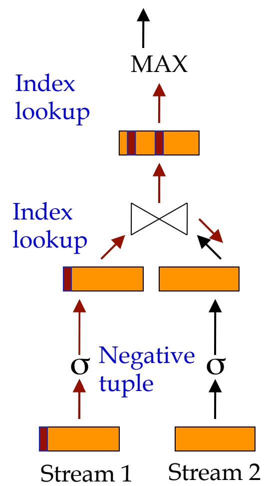
- Queuing and scheduling
  - Queues allow sources to push data into the query plan and operators to pull data when they need them
  - Timeslicing
  - Allowing multiple operators to process one or multiple tuples
- Tuple expiration
  - Removing old tuples from their state buffers and (possibly) update answers
  - Time-based window: simple – when time moves
    - Join results have interesting expiration times
    - Negation operator may force tuples to expire earlier
  - Count-based window: no. of tuples constant → overwrite the oldest tuple with the new arriving tuple

Query Processing (cont'd)

- Continuous query processing over sliding windows
  - Negative tuple approach
  - Direct approach

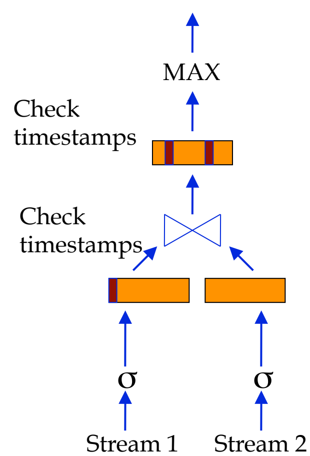
Negative Tuple Approach

- Negative tuples flow through the plan
- Corresponding "real" tuples deleted from operator state
- Updated answer generated, if necessary
- Each tuple is processed twice



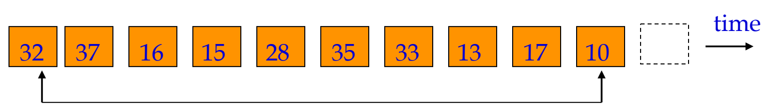
Direct Approach

- No negative tuples
- Operator states are scanned each time window moves
- Updated answer generated, if necessary
- Each tuple is processed once, but state maintenance expensive



Periodic Query Evaluation

- Generate output periodically rather than continuously
- No need to react to every insertion/expiration
- E.g., compute MAX over a 10-minute window that slides every minute
  - Store MAX over each non-overlapping one-minute chunk
  - Take the max of the MAXes stored in each chunk



max = max(10,17,…,37,32) = **37**

DSMS Optimization Framework

- General idea: similar to cost-based DBMS query optimization
- Generate candidate query plans
  - New DSMS-specific rewritings: selections and time-based sliding windows commute, but not selections and count-based windows
- Compute the cost of some of the plans and choose the cheapest plan
  - New cost model for persistent queries:
    - per unit time
    - queries typically evaluated in main memory, so disk I/O is not a concern

Additional DSMS Optimizations – Scheduling

- Scheduling
- Many tuples at a time:
  - Each operator gets a timeslice and processes all the tuples in its input queue
- Many operators at a time:
  - Each tuple is processed by all the operators in the pipeline
- Choice of scheduling strategy depends upon optimization goal
  - Minimize end-to-end latency?
  - Minimize queue sizes?

Additional DSMS Optimizations – Adaptivity

- System conditions can change throughout the lifetime of a persistent query
  - Query workload can change
  - Stream arrival rates can change
- Adjust the query plan on-the-fly
  - Or do away with the query plan and route tuples through the query operators according to some routing strategy
    - Eddies approach

Additional DSMS Optimizations – Load Shedding

- Random load shedding
  - Randomly drop a fraction of arriving tuples
- Semantic load shedding
  - Examine the contents of a tuple before deciding whether or not to drop it
  - Some tuples may have more value than others
- Or, rather than dropping tuples:
  - Spill to disk and process during idle times
  - Shorten the windows
  - Update the answer less often

Additional DSMS Optimizations – Multi-Query Processing

- DBMS: queries are typically issued individually
- DSMS: many persistent queries may be in the system at any given time
  - Some of them may be similar and could be executed together
  - E.g., similar SELECT and WHERE clauses, but different window length in the FROM clause
  - Or, same SELECT and FROM clauses, but different predicate in the WHERE clause