Aurora: a new model and architecture for data stream management

Daniel J. Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, Stan Zdonik.

The VLDB Journal, 12(2):120-139, Aug 2003.

Presented by Joel So (j2so@cs.uwaterloo.ca)

Topics to be covered

- Introduction: monitoring applications
- Aurora system model
- Aurora query algebra: SQuAl operators
- Aurora application example
- Aurora architecture & performance
- Related & future works
- Discussion points

Introduction: monitoring applications

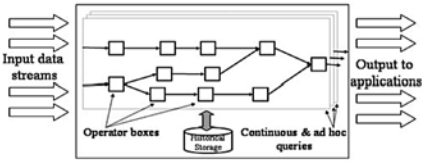
- **Traditional DBMS applications** have different characteristics/profiles than those of **Monitoring Applications** (monitor continuous streams of data)
- Change in paradigm: **HADP → DAHP**
- Other differences: data history relevance, triggering, data precision, real-time requirements
- Examples:
  - **Traditional:** PO systems, hospital record keeping
  - **Monitoring:** stock tickers, military vitality systems

Introduction: monitoring applications

- Traditional DBMSs are not well-suited for implementing monitoring applications
- **Aurora** is a stream management system (prototype) that is designed to support monitoring applications
  - Streams, triggers, imprecise data, real-time requirements

Aurora system model

- **Streams in, streams out**
- **Application concept**: collection of Aurora networks (process orchestrations/flows)
- **Aurora network**: choreography of Aurora "SQuAl" operators and connection points (and sinks) forming queries

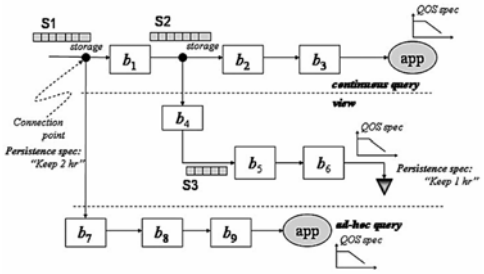


Aurora system model

- **Aurora operators**: filter, map, union, bsort, aggregate, join, resample (more on these later)
- **Connection point**: arc in the network where network branches can be added/deleted dynamically and historical/static data is available (persistence specification OR static data source)
- Other constructs: QoS specifications, order specifications (more on these later)

Aurora system model

- **Query model**: continuous, view, ad-hoc



Aurora query algebra: SQuAl operators

- **SQuAl**: Aurora's Stream Query Algebra
- A stream *schema* has the form (**TS**, A_1,...,A_n) where **TS** is the timestamp attribute populated by Aurora, and A_1, ..., A_n are tuple attributes (data fields)
- Thus, a single *tuple* from a stream takes the form ([**TS**=*ts*], A_1=v_1,...,A_n=v_n)

Aurora query algebra: SQuAl operators

- Three **order-agnostic operators**
  - filter, map, union
- Four **order-sensitive operators**
  - bsort, aggregate, join, resample
- **Order specification** (for order-sensitive operators)
  - O = Order(On A, Slack *n*, GroupBy B_1,...,B_m)

Aurora query algebra: SQuAl operators

- **Filter**(P_1,...,P_m)(S)
  - For each tuple *t* in stream S, route *t* to the first output branch with predicate P_i match

Aurora query algebra: SQuAl operators

- **Map**(B_1=F_1,...,B_m=F_m)(S)
  - For each tuple *t* in stream S, output (B_1=F_1(*t*),...,B_m=F_1(*t*)) where B_i's are mapped attribute names and F_i's are functions over tuples in S

Aurora query algebra: SQuAl operators

- **Union**(S_1,...,S_n)
  - For each tuple in any of streams S_1,...,S_n (all sharing a common schema), output to a single stream (order agnostic, no guarantees)

Aurora query algebra: SQuAl operators

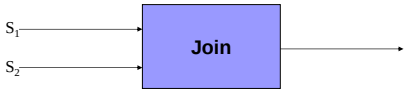
- **BSort**(Assuming O)(S)
  - O = Order(On A, Slack *n*, GroupBy B_1,...,B_m)
  - Bubble sort S over attribute A, *n* passes, using buffer of size *n*+1, output sorted stream

Aurora query algebra: SQuAl operators

- **Aggregate**(F, Assuming O, Size *s*, Advance *i*)(S)
  - O = Order(On A, Slack *n*, GroupBy B_1,...,B_m)
  - Apply aggregating function F to window *W* of S of size *s* (w.r.t. values of A), output result as tuple with schema: (A, B_1,...,B_m)++(F(*W*))
  - Advance window *W* by *i* tuples, repeat

Aurora query algebra: SQuAl operators

- **Join**(P, Size *s*, Left Ass. O_1, Right Ass. O_2)(S_1)(S_2)
  - O_1, O_2 are order specifications for S_1, S_2 respectively
  - Join tuples of S_1 and S_2 where predicate P is satisfied, within a window of size *s* (w.r.t. order attributes of O_1 and O_2)



Aurora query algebra: SQuAl operators

- **Resample**(F, Size *s*, Left Ass. O_1, Right Ass. O_2)(S_1)(S_2)
  - O_1, O_2 are order specifications for S_1, S_2 respectively
  - Interpolate missing points in S_2 as specified in S_1, using the interpolation function F, over a window of size 2*s* (w.r.t. order attributes of O_1 and O_2)

Aurora application example

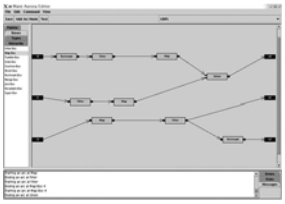
- Financial services application: Aurora case study taken from "Retrospective on Aurora" [1]
- Demonstrates aggregation, user-defined mapping, merging, filtering
- Scenario:
  - Institution receives two ticker feeds: Reuters, Comstock
  - Each feed contains securities from NYSE and NASDAQ
  - Each feed contains some "fast" securities and some "slow" securities
    - Fast securites: ticker data should be available every 5 s
    - Slow securities: ticker data should be available every 60 s

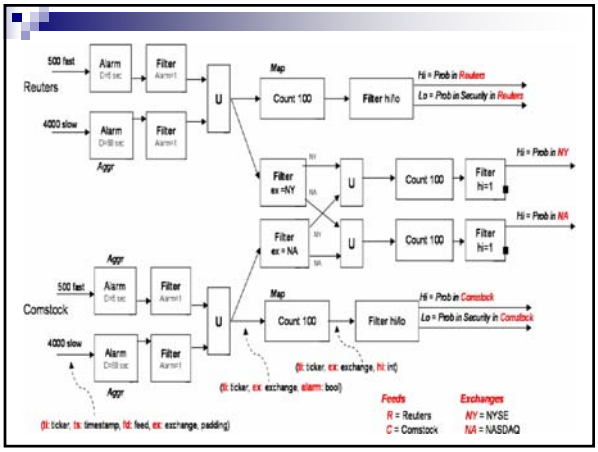
Aurora application example

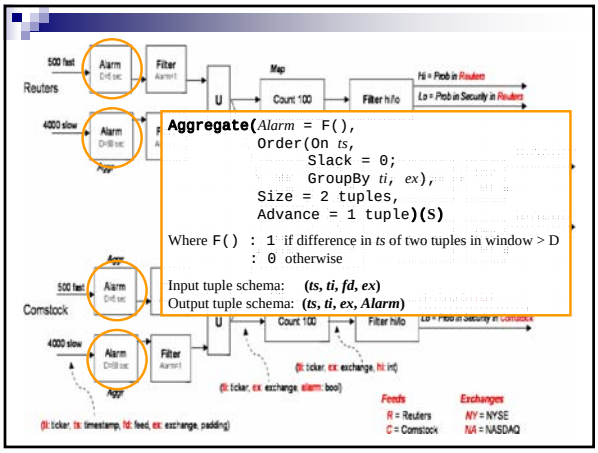
- Monitoring application functionality:
  - Monitor both feeds for ticker data delays
  - Output the following streams:
    1. Delayed securities in Reuters feed (Reuters "low alarm")
    2. Delayed securities in Comstock feed (Comstock "low alarm")
    3. Reuters "high alarm" for every 100 delayed securities in Reuters feed (disable Reuters low alarms on first high alarm)
    4. Comstock "high alarm" for every 100 delayed securities in Comstock feed (disable Comstock low alarms on first high alarm)
    5. NASDAQ "high alarm" for every 100 delayed NASDAQ securities (from either feed, combined)
    6. NYSE "high alarm" for every 100 delayed NYSE securites (from either feed, combined)

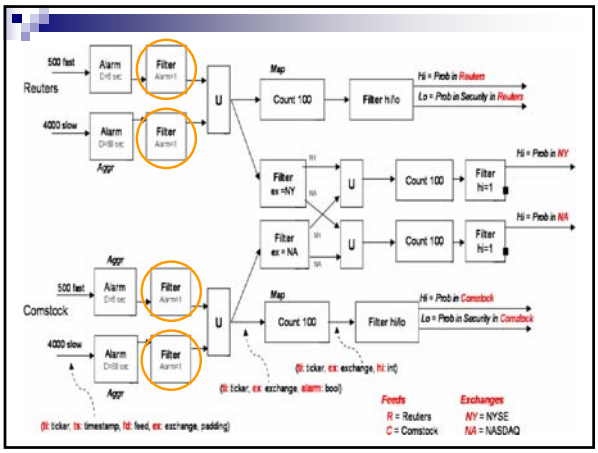
Aurora application example

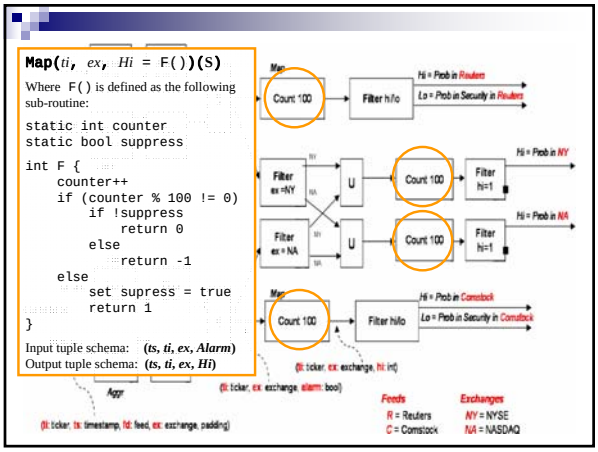
- Aurora prototype provides graphical tools for defining and configuring networks:

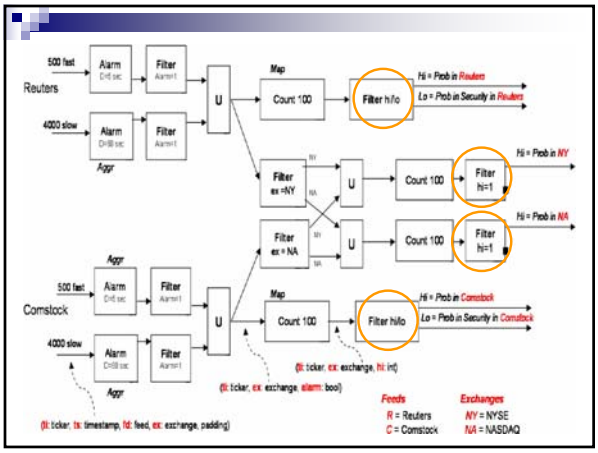






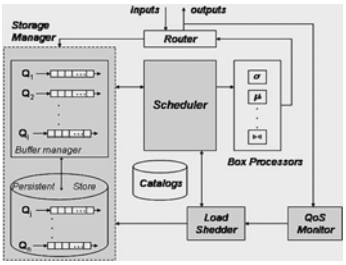

500 fast
Alarm
Filter
Reuters
4000 slow
Alarm
Filter
Aggr
U
Map
Count 100
Filter hi/lo
Hi = Prob in Reuters
Lo = Prob in Security in Reuters
Filter ex =NY
Filter ex = NA
Count 100
Filter hi=1
Hi = Prob in NY
Hi = Prob in NA
500 fast
Alarm
Filter
Comstock
4000 slow
Alarm
Filter
Aggr
Hi = Prob in Comstock
Lo = Prob in Security in Comstock
(tt: ticker, ex: exchange, hi: int)
(tt: ticker, ex: exchange, alarm: bool)
(tt: ticker, ts: timestamp, fd: feed, ex: exchange, padding)
Feeds
R = Reuters
C = Comstock
Exchanges
NY = NYSE
NA = NASDAQ


Map(ti, ex, Hi = F())(S)
Where F() is defined as the following sub-routine:
static int counter
static bool suppress
int F {
counter++
if (counter % 100 != 0)
if !suppress
return 0
else
return -1
else
set supress = true
return 1
}
Input tuple schema: (ts, ti, ex, Alarm)
Output tuple schema: (ts, ti, ex, Hi)
Count 100
Filter hi/lo
Filter ex =NY
Filter ex = NA
U
Count 100
Filter hi=1
Hi = Prob in NY
Hi = Prob in NA
Hi = Prob in Comstock
Lo = Prob in Security in Comstock
(tt: ticker, ex: exchange, hi: int)
(tt: ticker, ex: exchange, alarm: bool)
Feeds
R = Reuters
C = Comstock
Exchanges
NY = NYSE
NA = NASDAQ


500 fast
Alarm
Filter
Reuters
4000 slow
Alarm
Filter
Aggr
U
Map
Count 100
Filter hi/lo
Hi = Prob in Reuters
Lo = Prob in Security in Reuters
Filter ex =NY
Filter ex = NA
Count 100
Filter hi=1
Hi = Prob in NY
Hi = Prob in NA
500 fast
Alarm
Filter
Comstock
4000 slow
Alarm
Filter
Aggr
Hi = Prob in Comstock
Lo = Prob in Security in Comstock
(tt: ticker, ex: exchange, hi: int)
(tt: ticker, ex: exchange, alarm: bool)
(tt: ticker, ts: timestamp, fd: feed, ex: exchange, padding)
Feeds
R = Reuters
C = Comstock
Exchanges
NY = NYSE
NA = NASDAQ

Aurora architecture & performance

- Aurora run-time architecture:



Aurora architecture & performance

- Scheduler optimizations
  - **Train scheduling:** maximize the number of tuples to be processed by a box before processing begins (i.e., tuple train creation)
    - Exploits intrabox nonlinearity
  - **Superbox scheduling:** maximize the number of boxes to process the tuple train consecutively (i.e., tuple train push-through)
    - Exploits interbox nonlinearity

Aurora architecture & performance

- Query optimizations
  - **Insert map operators** to project out unneeded attributes at the earliest point in the network
  - **Combine boxes** wherever possible
  - **Reorder commutative boxes** based on historical statistics:
    - Cost: $c(b)$ is the expected time for box b to process a tuple
    - Selectivity: $s(b)$ is the expected number of output tuples from box b, per input tuple
    - Net cost of adjacent boxes b_i and b_j: $c(b_i) + c(b_j)xs(b_i)$
    - Reorder if b_i and b_j are commutative and $(c(b_j) + c(b_i)xs(b_j)) < (c(b_i) + c(b_j)xs(b_i))$

Aurora architecture & performance

- Introspection: overload detection
  - **Static analysis** using c(b), s(b), and tuple production rate r(d) for each data source
    - System capacity must be > minimum aggregate capacity calculated from c(), s(), r() values
  - **Dynamic analysis** using delay-based QoS information
    - Delay (latency) easily calculated with tuple timestamps
    - How many outputs are experiencing delays above the acceptable threshold (as per QoS spec)?

Aurora architecture & performance

- QoS specifications
  - User-defined for each network output stream
  - Latency (delay) graph
    - Specifies how much delay is tolerable
  - Drop % graph
    - Specifies what % of dropped tuples is tolerable
  - Output value utility graph
    - Defined for specific tuple attribute(s)
    - Specifies a QoS rating from 0-1 for ranges of possible values
    - I.e., defines what attribute value ranges are most important

Aurora architecture & performance

- Load shedding
  - Performed when overload detected
  - Load shedder inserts “drop boxes”
    - System-level operators that *randomly* drop tuples from a stream (at a specified rate)
    - Load shedder determines (based on drop % graphs) how many tuples can be dropped to minimize the overall decrease in QoS resulting from the load shedding
    - Drop boxes are placed as far upstream in network as possible

Aurora architecture & performance

- Semantic load shedding
  - Using output value utility graphs, determine which range of tuple values is least important
  - Filter out these tuples (drop) as far upstream as possible by inserting filter operators
  - Non-trivial:
    - Output value utility graphs are based on tuple values at the output of the network
    - Filter predicates upstream must be based on upstream (pre-processed) tuple values
    - **Backward interval propagation:** use operator inverse functions
    - **Forward interval propagation:** estimate and tune (by trial-error) filter predicates at upstream split points (or sources) and test downstream results
  - In practice, use both techniques for propagating filter predicates until sufficient load has been shed (and overall QoS, including value-based, is healthy)

Related & future works

- Related works:
  - Competing/similar technologies: NiagaraCQ, Fjords, Tribeca, STREAM
  - Real-time and temporal DBMSs
- Future works:
  - Findings from case studies [1]
    - E.g., open windows, SQL update & read operators, support for XML feeds, continued focus on performance
  - Core Aurora system frozen and becoming commercial
  - Follow-on system, called Borealis [2], in the works
    - *Distributed* stream-processing system built on core stream-processing functionality from Aurora

Discussion points

- Can input stream tuple production rates (i.e., $r(d)$) be used for scheduling optimization, in addition to capacity analysis?
- How are stateful operators (e.g., user-defined counters, or long-running aggregations) persisted for failover scenarios? What implications (e.g. node affinity) would this have in a distributed Aurora architecture?
- What about adapters and their implications on throughput, tuple ordering, etc.?

Additional references

- 1. H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik. Retrospective on Aurora. *The VLDB Journal*, 13:370–383, 2004.
- 2. D. Abadi, Y. Ahmad, M. Balazinska, U. Cetintemel, M. Cherniack, J-H. Hwang, W. Lindner, A. Rasin, N. Tatbul, Y. Xing, and S. Zdonik. *The design of the Borealis stream processing engine*. In *Proc. 1st Biennial Conf. on Innovative Data Syst. Res.*, 2005.