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 Varun Singhal
(vsinghal@seas.upenn.edu)

AURORA

- Introduction
- Aurora System Model
- Aurora Optimization
- Runtime Operation
- Aurora System Query model
- Conclusion

Introduction:Traditional DBMSs

- Passive repository: Human-Active, DBMS-Passive(HADP) model
- The current of state of the data is important:
  Previous data needs to be extracted form the log
- Triggers and alerts as second-class citizens
- Perfect synchronization of data elements and exact query answers
- No real-time services from applications

Introduction: Monitoring apps

- Monitoring applications are applications that monitor continuous streams of data
- Active repository: DBMS-Active, Human-Passive (DAHP) model
- History of the data is important: Not only the current state but also the previous history
- Triggers and alerts as the first-class citizens
- Missing or imprecise data, and approximate query answers
- Real-time services required by applications

Introducation: Monitoring apps

- Target Applications :military
  financial analysis
  tracking
  other real-time
  applications

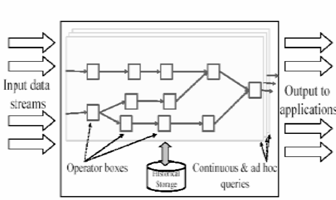
Car Navigation System

- Data(e.g., the location of the car) comes from external sources
- History of the data is required(e.g., display a trajectory of your car in the past 20 minutes)
- Trigger and alert oriented: an alert for the driver when the car is approaching to an intersection
- The location of the car is not always perfectly transmitted due to interferences etc..

Aurora System Model

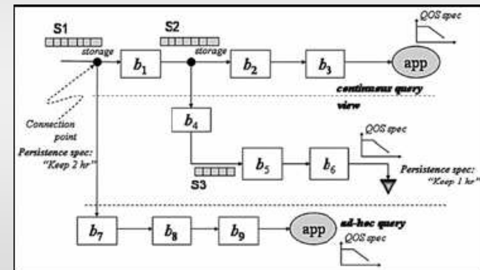
Aurora is a data-flow system that makes use of boxes and arrows paradigm. Its basic job is to process the incoming streams in a way defined by the application administrator.

- Continuous stream data comes
- Flow through a set of operators
- Output to application or materialized
- Multiple streams can be merged



Aurora System Model

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



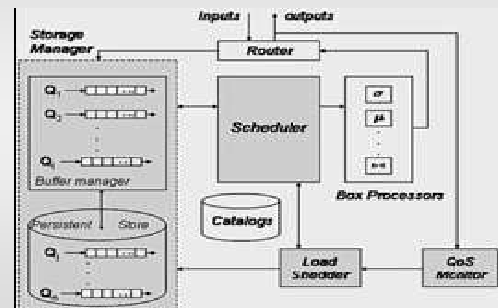
Aurora Optimization

- Dynamic continuous query optimization
  - ❖Inserting projections
  - ❖Combining Boxes
  - ❖Reordering Boxes

    c(bi) +c(bj)*s(bi)

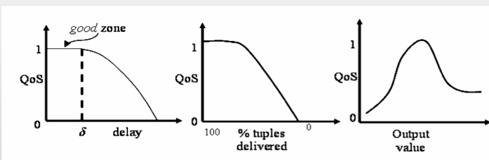
- Ad hoc query optimization

Aurora run-time Architecture



Qos Data Structures

•Response times – output tuples should be produced in a timely fashion, as otherwise QoS will degrade as delays get longer.
•Tuple drops – if tuples are dropped to shed load, then the QoS of the affected outputs will deteriorate.
•Values produced – QoS clearly depends on whether or not important values are being produced.

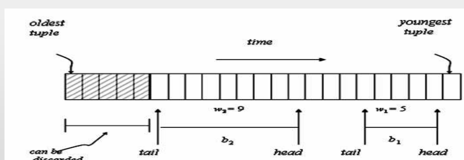


Storage Management

- Queue Management

  Each window operation requires a historical collection of tuples to be stored equal to the size of the window. The storage manager must manage a collection of variable length queues of tuples. There is one queue at the output of each box which is shared by all successor boxes. Each successor box maintains two pointers, head which is the oldest tuple that this box has not processed and tail denotes the oldest tuple that the box needs. Storage manager pages queue blocks into and out of main memory using a novel replacement policy.

- Connection Point Management



Real-time Scheduling

- Train Scheduling

  - Interbox nonlinearity

If buffer space is not sufficient and tuples need to be shuttled back and forth between memory and disk several times throughout their lifetime. Minimize tuple crashing. Scheduler can decide in advance that box b2 is going to be scheduled right after b1 then the overhead of the storage manager can be avoided while transferring b1's output to b2's queue.

  - Intrabox nonlinearity

The cost of tuple processing may decrease as the number of tuples that are available for processing at a given box increases. This reduction in unit tuple processing costs occurs because the total number of box calls needed to process a given number of tuples decreases. Secondly, a box may optimize its execution better with a larger number of tuples available in its queue.

Train Scheduling

Aurora exploits the benefits of interbox and intrabox nonlinearity through train scheduling by

- have boxes queue as many tuples as possible without processing, thereby generating long tuple trains
- process complete trains at once, thereby exploiting intrabox nonlinearity
- pass them to subsequent boxes without having to go to disk, thereby exploiting interbox nonlinearity.

Train scheduling has two goals: its primary goal is to minimize the number of I/O operations performed per tuple. A secondary goal is to minimize the number of box calls made per tuple. T*rain scheduling* to describe the batching of multiple tuples as input to a single box and *superbox scheduling* to describe scheduling actions that push a tuple train through multiple boxes. Both these require the Aurora scheduler to tell each box when to execute and how many queued tuples to process thus increasing the load on the scheduler.

Priority Assignment

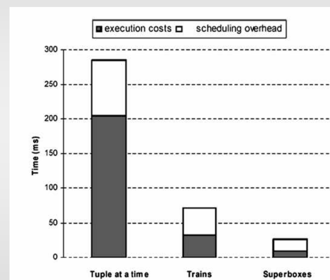
The latency of each output tuple is the sum of the tuple's processing delay and its waiting delay. The processing delay is a function of input tuple rates and box costs, the waiting delay is primarily a function of scheduling. Aurora' s goal is to assign priorities to outputs so as to achieve the per-output waiting delays that maximize the overall QoS.

Aurora currently considers two approaches for priority assignment

- **State-based** approach, assigns priorities to outputs based on their expected utility under the current system state and then picks for execution, at each scheduling instance, the output with the highest utility. In this approach, the utility of an output can be determined by computing how much QoS will be sacrificed if the execution of the output is deferred.

- **Feedback-based** approach continuously observes the performance of the system and dynamically reassigns priorities to outputs, properly increasing the priorities of those that are not doing well and decreasing priorities of the applications that are already in their good zones.

Scheduler Performance



Introspection

Aurora employs static and run-time introspection techniques to predict and detect overload situations.

- **Static Analysis**

  The goal of static analysis is to determine if the hardware running the Aurora network is sized correctly. If insufficient computational resources are present to handle the steady-state requirements of an Aurora network, then queue lengths will increase without bound and response times will become arbitrarily large. In this case either more resources have to be provided or by doing load shedding.

- **Dynamic Analysis**

  This uses delay-based QoS information. Aurora timestamps all tuples from data sources as they arrive. Furthermore, all Aurora operators preserve the tuple timestamps as they produce output Tuples. When Aurora delivers an output tuple to an application, it checks the corresponding delay-based QoS graph for that output to ascertain that the delay is at an acceptable level (i.e., the output is in the *good* zone). If enough outputs are observed to be outside of the good zone, this is a good indication of overload.

Load Shedding

- Dropping tuples

It involves dropping tuples at random points in the network and attempts to minimize the degradation in the overall system QoS. This is accomplished by dropping tuples on network branches that terminate in more tolerant outputs.

- Semantic load shedding by Filtering tuples

Semantic load shedding drops tuples in a more controlled way by dropping less important tuples rather than random ones using filters. It makes use of the Output value QoS specification to decide which tuples have a higher values.

SQuAl: Aurora Query Algebra

- **SQuAl**: Aurora's Stream Query Algebra
- A stream *schema* has the form (**TS**, A1,...,An) where **TS** is the timestamp attribute populated by Aurora, and A1, ..., An are tuple attributes (data fields)
- Thus, a single *tuple* from a stream takes the form ([**TS**=*ts*], A1=*v1*,...,An=*vn*)

SQuAl: Operators

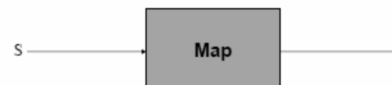
- Three **order-agnostic operators**
  filter, map, union
- Four **order-sensitive operators**
  bsort, aggregate, join, resample
- **Order specification** (for order-sensitive operators)
  O = Order(On A, Slack *n*, GroupBy B1,...,Bm)

SQuAl: Operators

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

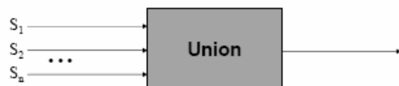
SQuAl: Operators

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



SQuAl: Operators

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



SQuAl: Operators

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

SQuAl: Operators

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

Suppose one wishes to compute an hourly average price (*Price*) per stock (*Sid*) over a stream of stock quotes that is known to be ordered by the time the quote was issued (*Time*).This can be expressed as:

Aggregate [Avg (Price),
AssumingOr der (On Time, GroupBy Sid),
Size 1 hour,
Advance 1 hour]

which will compute an average stock price for each stock ID for each hour

1. (MSF,1:00,$20)
2. (INT,1:00,$16)
3. (IBM,1:00,$24)
4. (IBM,1:15,$20)
5. (IBM,1:30,$23)
6. (MSF,1:30,$24)
7. (INT,1:30,$12)
8. (IBM,1:45,$13)
9. (IBM,2:00,$17)
10. (INT,2:00,$16)
11. (MSF,2:00,$22)

...

MSF:1-1:59 Tups: 1,6

MSF:2-2:59 Tups: 11,…

INT:1-1:59 Tups: 2,7

INT:2-2:59 Tups: 10,…

IBM:1-1:59 Tups: 3,4,5,8

IBM:2-2:59 Tups: 9,…

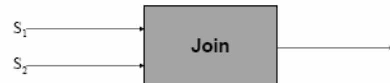
1. (IBM,1:00,$20)
2. (INT,1:00,$14)
3. (MSF,1:00,$22)

...

Input Stream *Windows* *Output Stream*

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)



Platoon X

1. (1,2:00,3)
2. (2,2:00,1)
3. (3,2:00,1)
4. (1,2:05,4)
5. (2,2:05,2)
6. (3,2:05,1)
7. (1,2:10,4)
8. (2,2:10,3)
9. (3,2:10,1)
10. (1,2:15,2)
11. (2,2:15,4)
12. (3,2:15,1)

…

Platoon Y

1. (10, 1:55, 3)
2. (11, 1:55, 4)
3. (12, 1:55, 5)
4. (13, 1:55, 6)

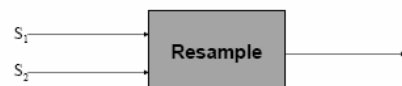
Join (P, Size 10 min, Assuming Left O, Assuming Right O)

1. (1,2:00,3,10,1:55,3)
2. (1,2:00,3,10,2:05,3)
3. (1,2:05,4,11,1:55,4)
4. (1,2:05,4,11,2:05,4)
5. (1,2:05,4,10,2:15,4)
6. (1,2:05,4,11,2:15,4)
7. (1,2:10,4,11,2:05,4)
8. (1,2:10,4,10,2:15,4)
9. (1,2:10,4,11,2:15,4)
10. (2,2:10,3,10,2:05,3)
11. (2,2:15,4,11,2:05,4)
12. (2,2:15,4,10,2:15,4)
13. (2,2:15,4,11,2:15,4)
14. (2,2:15,4,10,2:25,4)
15. (3,2:15,1,13,2:25,1)

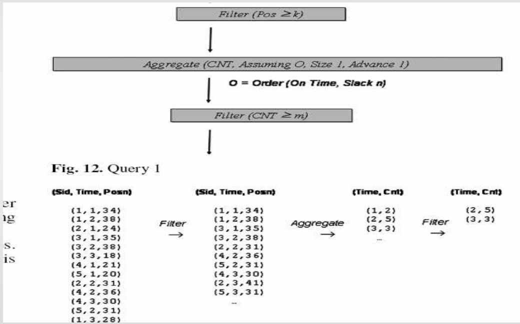
…

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 $2s$ (w.r.t. order attributes of O_1 and O_2)

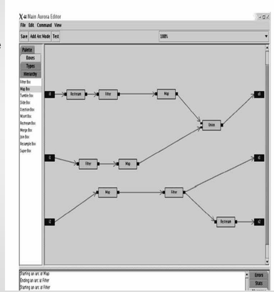


Produce an output whenever m soldiers are across some border k at the same time (with border crossing detection determined by the predicate Pos ≥ k).



Implementation

The prototype has a Java-based GUI that allows construction and execution of Aurora networks.The current interface supports construction of arbitrary Aurora networks, specification of QoS graphs, stream-type inferencing, and zooming. Users construct an Aurora network by simply dragging and dropping operators from the operator palette (shown on the left of the GUI) and connecting them



Conclusion

- Aurora is a new rising star in DBMS
- More demand for monitoring applications
- Future directions:

  Aurora* for distributed processing

  More efficient data handing algorithm for missing and/or imprecise data that is common in sensor network