Observations

In this paper, we present Aurora, a new DBMS that is currently under construction. This paper introduces monitoring applications. Monitoring applications are applications that monitor continuous streams of data. The fact that a software system must process and react to continual inputs from many sources (e.g., sensors) rather than from human operators requires one to re-think the fundamental architecture of a DBMS for this application area.

Traditional DBMSs have been oriented toward business data processing, and consequently are designed to address the needs of these applications. The Assumptions they made are:-

- They have assumed that the DBMS is a passive repository storing a large collection of data elements and that humans initiate queries and transactions on this repository. We call this a Human-Active, DBMS-Passive (HADP) model.
- They have assumed that the current state of the data is the only thing that is important.
- The third assumption is that triggers and alerters are second-class citizens.
- Fourth, DBMSs assume that data elements are synchronized and that queries have exact answers. In many stream-oriented applications, data arrives asynchronously and answers must be computed with incomplete information. Lastly, DBMSs assume that applications require no real-time services.

Because of the high volume of monitored data, monitoring applications would benefit from DBMS support. Existing DBMS systems, however, are ill suited for such applications since they target business applications. First, monitoring applications get their data from external sources (e.g., sensors) rather than from humans issuing transactions. Second, monitoring applications require data management that extends over some history of values reported in a stream, and not just over the most recently reported values. Third, most monitoring applications are trigger-oriented. The trigger processing required in this environment far exceeds that found in traditional DBMS applications. Fourth, stream data is often lost, stale, or intentionally omitted for processing reasons. Lastly, many monitoring applications have real-time requirements.

Key contributions and IDEA

Monitoring applications are very difficult to implement in traditional DBMSs. First, the basic computation model is wrong: DBMSs have a HADP model while monitoring applications often require a DAHP model.
First, he can encode the time series as current data in normal tables. In this case, assembling the historical time series is very expensive because the required data is spread over many tuples, thereby dramatically slowing performance. Alternately, he can encode time series information in binary large objects to achieve physical locality, at the expense of making queries to individual values in the time series very difficult. If a monitoring application had a very large number of triggers or alerters, then current DBMSs would fail because they do not scale past a few triggers per table. No DBMS has built-in facilities for approximate query answering.Aurora data is assumed to come from a variety of data sources such as computer programs that generate values at regular or irregular intervals or hardware sensors. We will use the term data source for either case. In addition, a data stream is the term we will use for the collection of data values that are presented by a data source. Each data source is assumed to have a unique source identifier and Aurora timestamps every incoming tuple to monitor the quality of service being provided. The basic job of Aurora is to process incoming streams in the way defined by an application administrator. Aurora is fundamentally a data-flow system and uses the popular boxes and arrows paradigm found in most process flow and workflow systems. Hence, tuples flow through a loop-free, directed graph of processing operations (i.e., boxes).

Ultimately, output streams are presented to applications, which must be programmed to deal with the asynchronous tuples in an output stream. Aurora can also maintain historical storage, primarily in order to support ad-hoc queries.

Aurora contains built-in support for eight primitive operations for expressing its stream processing requirements, windowed, slide, tumble, latch, resample, filter and drop, map, groupby, join. Other desirable idioms for stream processing can be expressed as compositions of Aurora's built-in primitives.

Aurora supports continual queries (real-time processing), views, and ad-hoc queries all using substantially the same mechanisms.

Aurora will support a hierarchical collection of groups of boxes. A designer can begin near the top of the hierarchy where only a few superboxes are visible on the screen. A zoom capability is provided to allow him to move into specific portions of the network, by replacing a group with its constituent boxes and groups.

Stream-oriented operators that constitute the Aurora network, on the other hand, are designed to operate in a data flow mode in which data elements are processed as they appear on the input.

Dynamic Continuous Query Optimization: the optimizer will select a portion of the network for optimization. Then, it will find all connection points that surround the subnetwork to be optimized. It will hold all input messages at upstream connection points and drain the subnetwork of messages through all downstream connection points. The optimizer will then apply the following local tactics to the identified subnetwork.

- Inserting Projections. To project out all unneeded attributes.
- Combining boxes,
- Reordering boxes
- Reordering the operations in a conventional relational DBMS to an equivalent but more efficient form is a common technique in query optimization.

The initial boxes in an ad-hoc query can pull information from the B-tree associated with the corresponding connection point(s). When the historical operation is finished, Aurora switches the implementation to the standard push-based data structures, and continues processing in the conventional fashion. The basic purpose of Aurora run-time network is to process data flows through a potentially large workflow diagram.
The job of the Aurora Storage Manager (ASM) is to store all tuples required by an Aurora network. There are two kinds of requirements. First, ASM must manage storage for the tuples that are being passed through an Aurora network, and secondly, it must maintain extra tuple storage that may be required at connection points. Aurora attempts to maximize the perceived QoS for the outputs it produces. QoS, in general,

is a multidimensional function of several attributes of an Aurora system. These include:

- 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 important values are being produced or not.

Real-Time Scheduling: scheduling in Aurora is a complex problem due to the need to simultaneously address several issues including large system scale, real-time performance requirements, and dependencies between box executions. Furthermore, tuple processing in Aurora spans many scheduling and execution steps (i.e., an input tuple typically needs to go through many boxes before potentially contributing to an output stream) and may involve multiple accesses to secondary storage. Basing scheduling decisions solely on QoS requirements, thereby failing to address end-to-end tuple processing costs, might lead to drastic performance degradation especially under resource constraints. To this end, Aurora not only aims to maximize overall QoS but
also makes an explicit attempt to reduce overall tuple execution costs.
Aurora exploits the benefits of non-linearity in both inter-box and intra-box tuple processing primarily through train scheduling, a set of scheduling heuristics that attempt to (1) have boxes queue as many tuples as possible without processing thereby generating long tuple trains; (2) process complete trains at once thereby exploiting intra-box non-linearity; and (3) pass them to subsequent boxes without having to go to disk thereby exploiting inter-box non-linearity.

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.
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.
Dynamic Analysis:- Even if the system has sufficient resources to execute a given Aurora network under expected conditions, unpredictable, long-duration spikes in input rates may deteriorate performance to a level that renders the system useless.
Load Shedding:- When an overload is detected as a result of static or dynamic analysis, Aurora attempts to reduce the volume of Aurora tuple

processing via load shedding. It can be done in two ways by using Load Shedding by Dropping Tuples or by Semantic Load Shedding by Filtering Tuples. This paper also discusses related work and the implementation status of the work.