Linear Road: Benchmarking Stream-Based Data Management Systems

by

Richard S. Tibbetts, III

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

October 27, 2003

Author . . . .
Department of Electrical Engineering and Computer Science
October 27, 2003

Certified by . . . .
Michael Stonebraker
Adjunct Professor
Thesis Supervisor

Accepted by . . . .
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Linear Road: Benchmarking Stream-Based Data Management Systems

by

Richard S. Tibbetts, III

Submitted to the Department of Electrical Engineering and Computer Science
on October 27, 2003, in Partial Fulfillment of the
Requirements for the Degree of
Master of Science in Computer Science and Engineering

Abstract

This thesis describes the design, implementation, and execution of the Linear Road benchmark for stream-based data management systems. The motivation for benchmarking and the selection of the benchmark application are described. Test harness implementation is discussed, as are experiences using the benchmark to evaluate the Aurora engine. Effects of this work on the evolution of the Aurora engine are also discussed.

Streams consist of continuous feeds of data from external data sources such as sensor networks or other monitoring systems. Stream data management systems execute continuous and historical queries over these streams, producing query results in real-time. This benchmark provides a means of comparing the functionality and performance of stream-based data management systems relative to each other and to relational systems.

The benchmark presented is motivated by the increasing prevalence of "variable tolling" on highway systems throughout the world. Variable tolling uses dynamically determined factors such as congestion levels and accident proximity to calculate tolls. Linear Road specifies a variable tolling system for a fictional urban area, including such features as accident detection and alerts, traffic congestion measurements, toll calculations, and ad hoc requests for travel time predictions and account balances. This benchmark has already been adopted in the Aurora [ACC+03] and STREAM [MWA+03] streaming data management systems.

Thesis Supervisor: Michael Stonebraker
Title: Adjunct Professor

Acknowledgments

I would like to thank Mike Stonebraker and Hari Balakrishnan for introducing me to streaming databases and the idea for Linear Road. The Aurora implementation would not have been possible without the help of the entire Aurora team, and most notably Mitch Cherniack and Eddie Galvez. I would also like to thank Ramachandran Balakrishna of the MIT Intelligent Transport Systems Program for working with me early on, explaining traffic simulation and showing me how to use his tools. Thank you to the MIT Zephyr[DEF+88] community for help in a variety of ways, particularly `-c tibbetts` for moral support. And most importantly I would like to thank my fiancée Aletta Wallace, for encouraging me throughout this entire process.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis presents the *Linear Road* benchmark for stream-based data management systems (or *stream systems* for short). Stream data management has become a highly active research area and has inspired the development of prototype systems such as Aurora [ACC+03], STREAM [MWA+03], TelegraphCQ [CCD+03] and Niagara [CDTW00]. Up until now, there has been no way to compare the performance of stream systems with each other, or with any relational database system adapted to store and query streamed data. Linear Road attempts to remedy this issue by providing a benchmark that has the following features which together comprise a rigorous stress-test for stream data management:

- a set of *continuous queries* (to continuously monitor incoming data streams) and *historical queries* (to query previously streamed historical data),
- high-volume data requirements, both in terms of incoming stream data (between 1000 and 10,000 incoming tuples per second) and the historical data that must be maintained (roughly between 20 million and 200 million tuples at a time) and
- real-time query response and historical data accuracy requirements.

Linear Road simulates an urban highway system that uses "variable tolls": tolls that are determined according to such dynamic factors as congestion, accident proximity and travel frequency. This benchmark is intended to serve as a stress test for stream-based data management: specifying fixed input data schemas and workloads, a suite of continuous and historical queries that must be supported, and performance (i.e., query and transaction response time) requirements. The degree to which a stream-based system can scale (as measured by the number of expressways supported, L) while satisfying these requirements serves as a basis for comparison of stream systems to each other and to existing relational technology. We refer to a system that can meet the performance and correctness requirements specified here, while servicing position reports emitted from L expressways, as having an L-rating for Linear Road.

1.1 Stream-Based Data Management Systems

Continuous query processing is a relatively new field in query processing. It deals with the execution of queries over infinite streams of data, rather than over fixed collections of data. Traditional query processing systems are powerful tools for examining stores of data. Continuous query systems are similarly powerful, but focus on processing and reacting to the data as it is collected. These systems are specially designed for "stream processing" problems.

Stream processing problems involve input data that is coming into existence over time. The data rate may be very high, or come in bursts. Output is calculated as soon as the required input data is available. Output is a function of all input data available up to the present time. Stream processing problems often make explicitly use of the time domain of their input data. For example, calculating the maximum value seen in the last 5 minutes. There can also be real-time requirements on processing, where results are required within a specified amount of time after data becomes available. A continuous query system will allow stream processing problems to be specified by programmers, and executed efficiently.

There are many stream processing problems. For example, stock markets and other financial systems can be treated as data sources. There are many queries that one might want to execute continuously and in real-time based on stock market data. A simple query might be as follows: *Inform me within ten seconds if the percentage change in the five-minute rolling average of stock XYZ exceeds the percentage change in the average of the entire market.* This query makes explicit use of time, and places real-time demands on its results.

There are currently a variety of continuous query processing systems under development in various research groups. They each offer different functionality, different programming interfaces, and a different vocabulary. This paper will use the terminology developed as part of the Aurora[ACC+03] project.

The fundamental construct in nearly all continuous query processing systems is the stream[BW01]. The stream is a possibly infinite sequence of tuples. Streams often represent the output of sensors, such as the location of objects or the temperature of a room over time. Streams may also be processed data. The output of a continuous query will itself be a stream of data. In our example, the input data from the stock market would be a stream, as would the output from the query. Internally, the intermediate data, such as the 5 minute rolling average of the entire market, may also be thought of as a stream.

In Aurora, as in most systems, streams are manipulated using operators. Operators have some number of input streams and some number of output streams. The values of the output streams depend on the values on the input streams. A simple operator would be a *filter* whose output is a subset of its input selected according to some predicate. In our example, *filter* will be used to select tuples where the percent change exceeds 5%, out of a stream of tuples describing the percent change. Another operator, *aggregate*, will be used to calculate the rolling average as well as the percent change in our example. *Aggregate* calculates functions such as average over a subset of its previous input data. There are several other basic operators. More specialized operators can be created by composing these simple operators, or by introducing application specific operators.

Aurora queries are networks of operators, output streams feeding into input streams, and data sources. These queries are expressed graphically. Our example query can

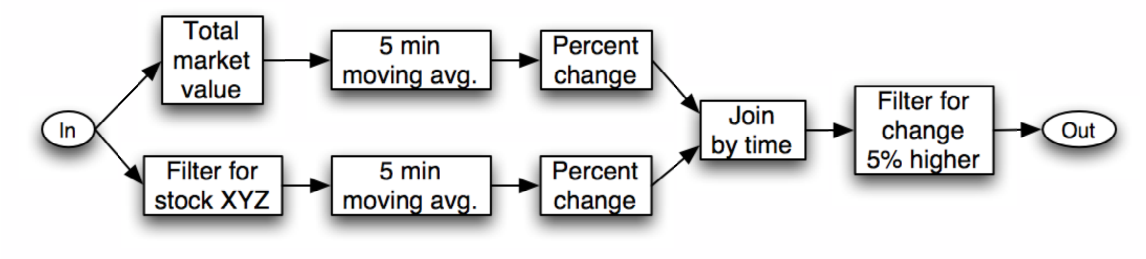

Figure 1-1: Example of an Aurora Query

be seen in Figure 1-1. Input data logically flows along the arrows, and each box adds a bit more processing. The top arc of the query calculates a total market value, then a 5 minute rolling average of that value, and then a percent change between the last 2 averages. The bottom arc filters out just stock XYZ's value, and does the same calculation. The Join box puts values from the top and bottom arcs together based on their timestamps, so that its output tuples contain the percent change in total market and in stock XYZ for a given time. Then the final filter just selects tuples where the change in XYZ is 5% higher than in the whole market.

Query processing systems support many data streams and many simultaneous queries and readers. Readers may be able to subscribe to streams of intermediate values. Such systems may eliminate common sub-queries and do other optimizations with stored data for windowed operators. They may also incorporated data from more traditional database systems.

Aurora is a single-node stream processing engine. Given a description of an operator network and a stream of input, it produces the appropriate stream of output. Aurora makes use of efficient scheduling algorithms [CCR+03] and techniques such as load shedding [TCZ+03] to achieve high performance. The load shedding functionality was not tested in this work, only the operator set and the scheduling.

Aurora has a default operator set which can be augmented with application specific operators. The default operator set at the time of this implementation is described in [ACC+03].

1.2 Benchmarking

This thesis attempts to define a useful benchmark for streaming databases. There is much history of benchmarking in the database community. For a discussion of previous work in benchmarking, see Chapter 7. Benchmarking allows users and designers of databases to compare performance between different software and hardware systems. Comparisons are important both for researchers evaluating designs and for consumers deploying systems. A good benchmark will greatly simplify a consumers choice of database system, and will help designers to build the kinds of systems that consumers desire.

According to Jim Gray, in his *Benchmark Handbook* [Gra93], a good benchmark will display four important qualities:

Relevant It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.

Portable It should be easy to implement the benchmark on many different systems and architectures.

Scalable The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems, and to parallel computer systems as computer performance and architecture evolve.

Simple The benchmark must be understandable, otherwise it will lack credibility.

As this thesis presents a new benchmark for a new kind of database system, these four criteria will be a guide.

1.3 Organization

The next chapter presents an overview of the Linear Road benchmark and some of the high-level design choices. Chapter 3 describes the precise specification of the benchmark requirements. Chapter 4 discusses the implementation of the benchmarking test harness. Experiences implementing the queries on the Aurora streaming database are found in chapter 5. Chapter 6 summarizes performance measurements from existing implementations. Related work is discussed in Chapter 7. Conclusions are drawn in Chapter 8.

Chapter 2

Overview of Linear Road Benchmark

2.1 Application-Level Benchmarking

An ideal benchmark will precisely define the queries to be answered by the underlying system using a standard language. This should be the same language that end-users will use to implement their own applications. This is important so that the benchmark is portable between systems, and to ensure the the systems being compared offer similar functionality.

A major challenge we faced in building a benchmark for streaming databases was the lack of a common language for specifying queries. There is a wide open range of languages in this space, including SQL-derivatives such as CQL [MWA+03] and Punctuated Streams [TMS03], entirely new languages like Medusa, and fully graphical query systems such as Aurora. Each of these languages describes a slightly different underlying set of basic operations. As a result, it is not possible to specify precise queries in a portable manner.

All of these systems are classified as streaming databases because they are all trying to solve a similar class of problems in a similar way. Rather than specify the queries, our benchmark specifies the application to be implemented. This "application-level benchmark" gives greater flexibility to implementors at the cost of added complexity in the benchmarking test harness. The benefit is that systems are able to showcase their unique operator sets while retaining the ability to compare the performance of their systems.

2.2 Variable Tolling

"Variable tolling" (also known as "congestion pricing") [ITS02, USD03, Poo02] is becoming increasingly prevalent in urban settings because of its effectiveness in reducing traffic congestion and to recent advances in microsensor technology that have made it feasible. Traffic congestion in major metropolitan areas is an increasing problem as expressways cannot be built fast enough to keep traffic flowing freely at peak periods. The idea behind *variable tolling* is to issue tolls that vary according to time-dependent factors such as congestion levels and accident proximity. The motivation of charging higher tolls during peak traffic periods is to discourage vehicles from using the roads and contributing to the congestion. Illinois, California, and Finland have pilot programs utilizing this concept. Moreover, both London and Singapore charge tolls at peak periods to let vehicles enter the downtown area using similar reasoning.

Variable tolling depends on the deployment of microsensors that continuously report on the positions of vehicles on monitored expressways. Continuous monitoring of expressways can be accomplished with two different architectures:

- *Smart Road, Dumb Vehicle*: In this scenario, position and speed-detecting sensors are embedded in the roads or roadside facilities. Vehicles carry devices that reflect the signal from the roadside systems thereby reporting the vehicle's position every time it passes a predetermined position on the road.

- *Dumb Road, Smart Vehicle*: In this scenario, every vehicle is equipped with an active device with a longer range that broadcasts the vehicle's position to a data collection network positioned at the side of the road.

Both of these architectures are being followed in the current pilots. For the purpose of this benchmark, we will assume a *dumb road, smart vehicle* architecture.

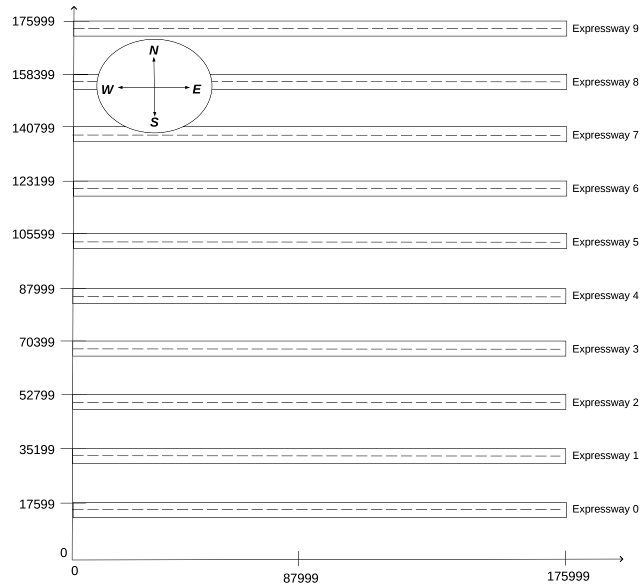

Figure 2-1: The Geometry of Linear City

2.3 Linear City

Linear Road is not intended to model a real city with all of its geographical complexities (hills, rivers etc.), nor is it intended to act as a control simulation of traffic patterns. Rather, it is intended solely as a stress test for stream data management, and therefore we assume a simple urban area consisting of some number of fixed length linear expressways (thereby inspiring the name, "Linear Road"), and excluding control mechanisms.

Linear City is a fictional metropolitan area that is 100 miles (176000 yards) wide and 100 miles long. Every position in the area can be specified with one yard granularity using (x, y) coordinates starting at (0, 0) and extending to (175999, 175999), as shown in Figure 2-1.

2.3.1 Expressways

There are 10 parallel expressways in Linear City, numbered from 0-9 and running horizontally 10 miles apart. Each expressway is 100 miles long and 96 feet (32 yards) wide, with 4 12-foot wide lanes in both (east and west) directions: 3 travel lanes and one additional 12 foot lane devoted to entrance and exit ramps. Thus, expressway #0 occupies the rectangle identified by corner points $(0, 17568)$ and $(175999, 17599)$; expressway #1 occupies the rectangle identified by corner points $(0, 35168)$ and $(175999, 35199)$; and in general, expressway $\#i$ occupies the rectangle identified by corner points $(0, (i + 1) * 17600 - 32)$ and $(175999, (i + 1) * 17600 - 1)$ for $0 \leq i \leq 9$. This is illustrated in Figure 2-1. For simplicity, there are no expressways that run vertically.

Each expressway has 100 on-ramps and 100 off-ramps in each direction, dividing it into 200 mile-long *segments* (100 eastbound and 100 westbound). The ramps are a third of a mile long, and allow vehicles to accelerate or decelerate. The on-ramp for a segment puts vehicles onto the segment just after the start of the segment. The offramp comes just before the end of the segment. Thus, each segment has a total of 8 lanes which we number from 0-7:

- lane 0 refers to the westbound entrance and exit ramps,
- lanes 1-3 refer to the westbound traffic lanes,
- lanes 4-6 refer to the eastbound traffic lanes, and
- lane 7 refers to the eastbound entrance and exit ramps.

A segment(for expressway $\#i$), and the positioning of each lane is illustrated in Figure 2-2.

2.3.2 Vehicles

There are 1,000,000 different vehicles registered in Linear City that use the expressways. Each registered vehicle is equipped with a sensor that reports its position as an (x, y) coordinate on the Linear City grid every 30 seconds while it is traveling on an entrance ramp, exit ramp or travel lane of an expressway. For the purposes of this benchmark, we will assume sensors to accurately identify vehicle coordinates.[1]

[1]With current technology, a digital GPS that uses multiple satellites can achieve position accuracy within 10 feet.[Gar02]

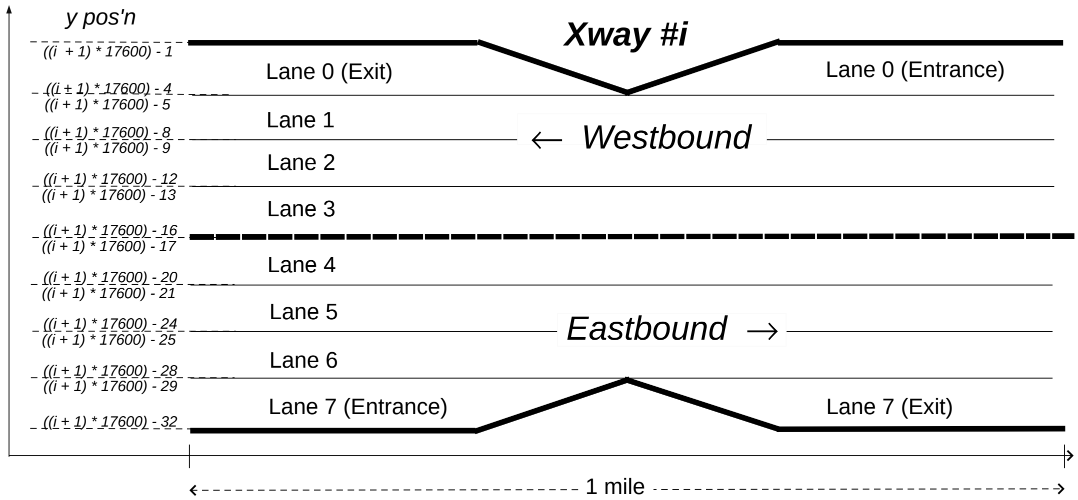

Figure 2-2: One Segment of a Linear Road Expressway

2.3.3 Tolls

The Linear Road system automatically charges each vehicle tolls for use of the expressway. Tolls are assessed to vehicles for each segment that they travel on except for those from which they exit the expressway. The toll assessed to a given vehicle for traveling on a given segment is calculated at the time the vehicle first issues a position report from the segment. By default, this toll is determined by the average speed and number of vehicles on the segment at the time the vehicle first issues a position report from the segment. Default tolls are overridden in cases where an accident has recently been detected in the given segment (or a segment in close proximity downstream), or when the driver being charged is a frequent user of the expressway system.

Chapter 3

Specification of Linear Road

In this chapter, we provide a detailed description of the Linear Road benchmark requirements. Running the benchmark will involve the following steps:

1. Choosing a *scale factor* (L) reflecting the number of expressways for which traffic data should be generated.

2. Using the *traffic simulator* (set with *scale factor*, L) to model the traffic patterns of Linear City, recording position data from each simulated automobile. This process generates flat files consisting of traffic data and historical query requests for a 3 hour period reflecting a load consistent with rush hour traffic.

3. Using the *stream system driver* to deliver data generated by the simulator to the implementing system in a manner consistent with the timestamps associated with the data by the simulator.

4. Generating a flat file containing all output tuples (with timestamps reflecting the times of their generation) in response to the continuous and historical queries discussed in Section 3.2.

5. Using the *validation tool* to check the latency and accuracy of generated output according to the criteria specified in Section 3.3.

The purpose of the benchmark is to determine the maximum *scale factor* at which a stream data management system can run while still meeting the latency and accuracy requirements specified herein. (This is known as the *L-rating* of the system.) Therefore, it is assumed that the benchmark will be run with increasingly larger scale factors until one is used for which the requirements cannot be met.

The benchmark specification is divided into three parts:

- *Inputs to the System:* Input data is described by its schemas in Section 3.1.1, and the simulation tool that generates the data in Section 3.1.2.

- *Outputs from the System:* Outputs are defined in terms of a suite of *continuous* (Section 3.2.1) and *historical* queries (Section 3.2.2) expected to be evaluated over system inputs, and historical data (Section 3.2.3) that must be maintained to satisfy these queries.

- *Evaluation Criteria:* Evaluation criteria are described in terms of performance (i.e., response time) requirements in Section 3.3.1, and the validation tool that verifies query output in Section 3.3.3.

3.1 Input

3.1.1 Input Schemas

The schemas for each input stream are summarized in Table 3.1 and described below. For simplicity, every field in every tuple is represented with a 32-bit signed integer.

Position report tuples are emitted by vehicles every 30 seconds while on an expressway. Position reports include the following identifying data:

- `Type` identifies these tuples as Position Reports (for all Position Report tuples, `Type` = 0).

- `Time` is a timestamp (as measured in seconds since the start of the simulation) identifying the time at which the position report was emitted. The clock(s) that

Input Tuple	*Schema*
Position Reports	`(Type, Time, VID, Speed, XWay, Mile, Offset, Lane, Dir)`
Account Queries	`(Type, Time, VID, QID)`
Expenditure Queries	`(Type, Time, VID, QID, XWay)`
Travel Queries	(Type, Time, VID, QID, XWay, $\mathtt{M}_{\mathtt{init}}$, $\mathtt{M}_{\mathtt{end}}$, DOW, TOD)

Table 3.1: Input Tuple Schemas

perform the timestamping are assumed accurate. Hence there is no issue with clock skew.

- `VID` is a vehicle identifier that uniquely identifies each vehicle registered with Linear Road.

- `Speed` is an integer number of miles per hour between 0 and 100.

In addition to the identifying data, position reports contain position data. Sensor emissions report a vehicle's position as an (x, y) coordinate on the Linear Road grid. To simplify the implementation of the benchmark, we assume that the traffic network preprocesses these coordinates to produce the following fields:

- `XWay` (*Expressway number:* $0 \ldots 9$): Equal to $\lfloor \frac{y}{17600} \rfloor$,

- `Mile` (*Mile number*: $0 \ldots 99$): Equal to $\lfloor \frac{x}{1760} \rfloor$,

- `Offset` (*Yards Since Last Mile Marker*: $0 \ldots 1759$): Equal to $x \ mod \ 1760$,

- `Lane` (*Travel lane*: $0 \ldots 7$): Equal to $\lfloor \frac{17599-(y \ mod \ 17600)}{8} \rfloor$, and

- `Dir` (*Direction*: 0 (West) or 1 (East)): Equal to $\lfloor \frac{\mathtt{Lane}}{4} \rfloor$.

Note that fields,

(`XWay`, `Mile`, `Dir`)

identify a *segment*. Fields,

(`XWay`, `Mile`, `Dir`, `Offset`, `Lane`)

Type	*Fields*	*Description*
Type	`Type`	0 = Position Report, 1 = Account Balance, 2 = Expenditure, 3 = Travel Time
Timestamp	`Time`	Seconds since start of simulation.
Vehicle ID	`VID`	(0 . . . 999,999)
Speed	`Speed`	(0 . . . 100)
Expressway #	`XWay`	(0 . . . 9)
Mile #	`Mile`, $\mathtt{M}_{\mathtt{init}}$, $\mathtt{M}_{\mathtt{end}}$	(0 . . . 99)
Offset	`Offset`	(0 . . . 1759)
Lane	`Lane`	0 = Ramp, 1 = Left, 2 = Middle, 3 = Right
Direction	`Dir`	0 = West, 1 = East
Query ID	`QID`	(0..999,999)
Day_of_week	`DOW`	0 = Sunday, 1 = Monday, . . ., 6 = Saturday
Time_of_day	`TOD`	(0:00 . . . 23:59)

Table 3.2: Input Field Types

identify a *position*: the most precise possible identifiable location for a vehicle.

The remaining three types of input tuples are issued from vehicles to trigger the invocation of historical queries, which are described in Section 3.2.2. The specific types associated with each field in input tuples are summarized in Table 3.2.

3.1.2 The Linear Road Simulation Tool

The Linear Road Simulation Tool consists of a *simulator* and a *driver*. The simulator generates input data and stores it in flat files. The driver reads the flat files and delivers the data to the benchmarked system in a manner consistent with the timestamps attached to each data tuple. In short, the simulator is concerned with the generation of traffic data while the driver ensures that this data is presented to stream systems with appropriate latencies.

In simulating a stream system, a distinction must be made between the time a tuple is generated and emitted (*generation time*) and the time it is delivered to the stream data management system (*arrival time*). Typically, the timestamp associated with a tuple is the data generation time, as this is more semantically meaningful as reflecting the time an event occurs that triggers the data emission. Our simulator/driver distinction allows us to make one simplifying assumption in this benchmark: for all tuples generated as inputs to the system, we assume that the generation time and arrival time for that tuple are the same. In other words, for the purposes of this benchmark we assume that tuples arrive to stream systems instantaneously (and we can achieve this by having a driver that delivers tuples to a stream system in a manner consistent with their timestamps.) While unrealistic in practice (this precludes tuples arriving out of order, for example), the benchmark goal of gauging the performance of stream systems is unaffected by this assumption. Throughout the remainder of this document, we will assume the timestamp associated with a data tuple is both its generation time and its arrival time to the stream system.

MITSIMLab

At the core of the simulation tool is the MITSIMLab traffic simulator [YK96]. This system generates a set of vehicles and repeatedly has each complete a *vehicle trip*: a journey beginning at an entry ramp on some segment and finishing at an exit ramp on some segment that lies on the same expressway. In making a vehicle trip, a vehicle is placed on the entrance ramp, begins reporting its position every 30 seconds and accelerating at a rate allowed by the other traffic. It then merges onto the expressway and moves towards its destination at a rate allowed by the traffic congestion. When the vehicle reaches its destination, it moves to the exit ramp and decelerates. It is assumed that a vehicle's maximum speed on a travel lane of an expressway is 100 MPH, and that its average speed on an entrance or exit ramp is no more than 40 MPH. This ensures that a vehicle emits at least one position report from every segment it travels in, and at least one position report from both an entrance ramp and an exit ramp. When a vehicle leaves the exit ramp completely, it stops reporting its position. Thus, every vehicle trip begins with a position report from an entrance ramp and ends with a position report from an exit ramp. The simulator staggers vehicle position reports so that at every second $\frac{1}{30}$ of the reports for vehicles currently on the expressway are emitted.

Simulation Parameters

A vehicle may travel on more than one expressway during the course of the simulation, but it will travel on only one expressway for a given vehicle trip. For each trip, the source location of a vehicle is uniformly distributed over all of the possible entrance ramps on the chosen expressway. The exit ramp is normally distributed with a mean segment location in the middle of the expressway (i.e., mile #50) and with a standard deviation of 20 miles. Hence, vehicles have an affinity for exiting in the downtown area. Vehicles choose eastbound or westbound ramps as appropriate. Once on the expressway, each vehicle proceeds according to a standard traffic spacing model built into the traffic simulator. The simulator should be run for 3 hours to simulate the traffic data that would be generated at rush hour with segment congestion averaging 200 vehicles.

At any given point in time during the simulation, there is exactly one accident on the Linear Road expressway system. An accident is generated at a random location in each direction on each expressway. At the moment the accident is cleared (20 minutes after the accident occurs), another one is generated at a random location to replace it. Traffic proceeds by the incident at a reduced speed in the remaining travel lanes. The traffic spacing model handles this calculation. The presence of an accident on a given segment of an expressway is relevant to the toll calculation for that and nearby segments (see Section 3.2.1).

Generated Data

The data generated by the simulator consists of 4 streams. The primary stream consists of position reports issued every 30 seconds by vehicles specifying their present speed and position on the expressways. The other three event streams are historical query requests from vehicles on the network, which query for account balances (the *Account Balance* query), total expenditures for the day (the *Daily Expenditure* query), or travel time predictions (the *Travel Time* query). Each time a vehicle issues a position report, with 1% probability it also generates an historical query. Of the historical queries that can be issued, *Account Balance* queries account for 50%, *Daily Expenditure* queries account for 10%, and *Travel Time* queries account for 40%. To avoid the complication of unpredictable event delivery order, the 4 input streams are multiplexed together as a single stream. Input tuples are padded to consist of 36 bytes divided into 9 4-byte integer fields for simplicity. The `Type` field can be used to demultiplex the input into its constituent streams.

3.2 Output

Systems implementing Linear Road are expected to support two *continuous* and three *historical* queries over input data. These queries are described in detail in Sections 3.2.1 and 3.2.2. As well, systems are expected to manage a certain amount of *historical data* to support these queries. Historical data requirements are described in Section 3.2.3. Just as with input data, all output data produced by queries should be uniform in size, consisting of 5 4-byte integer fields.

3.2.1 Continuous Queries

Systems implementing the Linear Road benchmark are expected to support two continuous queries over input data. The first continuous query involves calculating tolls every time a vehicle reports a position in a new segment and alerting the driver of this toll. The second continuous query involves detecting accidents on the expressway and alerting affected drivers. These queries are described more thoroughly below.

Toll Calculation and Alerts: Every registered vehicle in the Linear Road system has an associated *account* which specifies the the total toll charges that have been issued to drivers of the vehicle. One of the two continuous queries required to support the Linear Road benchmark involves calculating tolls and maintaining toll accounts.

Toll notification and *toll charging* constitute two different activities that occur at different times within Linear Road. Every time a position report identifies a vehicle as *entering* a new segment, a toll for that segment is calculated and the vehicle is notified of that toll. Every time a position report identifies a vehicle as *exiting* a segment, the toll reported for that segment is charged to the vehicle's account. Thus, a toll calculation for one segment often is concurrent with an account being debited for the previous segment. If the vehicle exits at the offramp of a segment, the toll for that segment is not charged.[1]

A toll notification is a tuple of the form,

$$(\texttt{Type} = \texttt{0},\ \texttt{VID},\ \texttt{Toll},\ \texttt{Speed}).$$

The field, `Type = 0` identifies this tuple as a toll notification, `VID` identifies the vehicle being assessed the toll and to which the toll notification is sent, `Toll` is the toll being charged and `Speed` is the average speed, as measured over all position reports emitted in the previous 5 minutes, in the segment for which the toll is charged.

By default, a toll calculation for a segment is based on the average speed and number of vehicles in the segment at the time (t) of the first position report that identifies the vehicle as entering the segment. Specifically, if the average speed of vehicles that are in the segment between times ($t\ -\ 5\ min$) and t is greater than or equal to 40 MPH, no toll is assessed. Otherwise, the default toll is determined by the formula,

$$basetoll \times (numvehicles - 150)^2$$

where *basetoll* is a predetermined constant, and *numvehicles* is the number of vehicles that emitted at least one position report from the segment between times ($t\ -\ 1\ min$) and t. The basic intuition is to raise tolls when congestion is high so as to discourage drivers from contributing to worse congestion.

The default toll is issued for a segment unless an accident is currently being detected within 5 segments downstream. Accident detection is discussed further below. In the event of an accident, no toll is charged on the 5 upstream highway segments. To encourage drivers to exit the expressway, a credit of *basecredit* (some predetermined constant) is applied to the vehicle's account if the vehicle exits the

[1] Thus, a driver is never assessed a toll for the last segment (mile # 99) of an expressway.

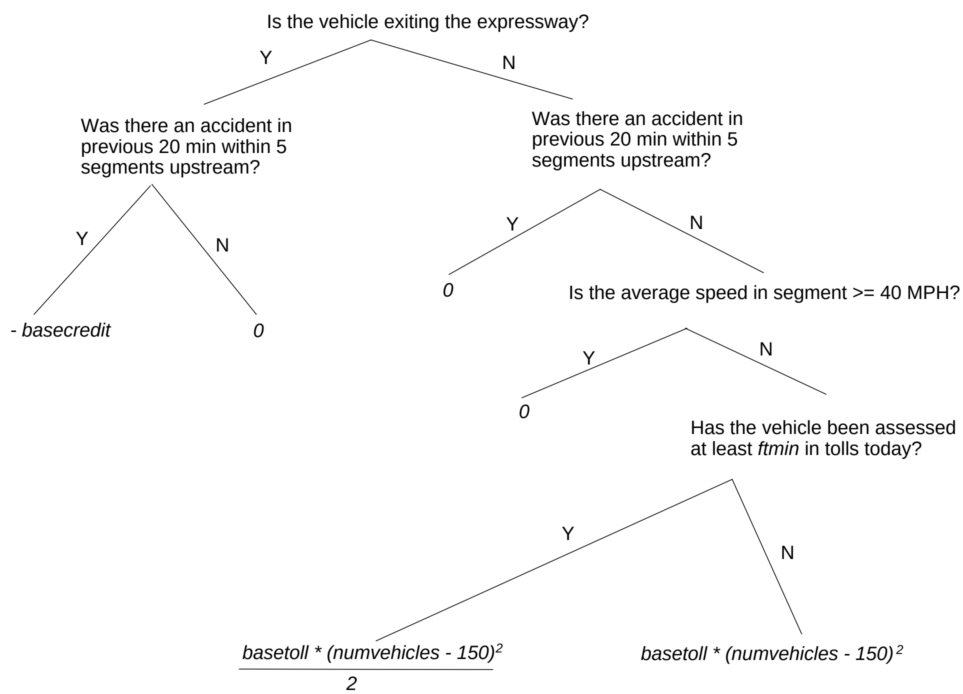

Figure 3-1: Toll Calculation for Linear Road

expressway in advance of the site of the accident.

Linear Road also assumes a special "frequent traveler" discount, whereby travelers receive a 50% discount on all tolls issued after they have accrued *ftmin* (some predetermined constant) in tolls on the same day. The accrual is from total charges and credits applied on the given day. The discount applies only to default tolls and therefore, for example, do not double the credit applied when a traveler exits just before the site of an accident. A precise summary of toll calculations is shown in the decision tree of Figure 3-1.

Toll data must be accurate to 30 seconds. In other words, every position report must be processed and reflected in the vehicle's account within 30 seconds of its arrival to the stream system.

Accident Detection and Alerts: The tracking system detects an *accident* on a given segment whenever two or more vehicles are *stopped* in that segment. A vehicle is considered *stopped* if four successive 30-second readings from the same vehicle come

Query Response	*Schema*
Toll notifications	`(Type = 0, VID, Toll, Speed)`
Accident notification	`(Type = 1, VID, XWay, Mile, Dir)`

Table 3.3: Output Tuple Schemas: Continuous Queries

from the same position (i.e., the same expressway, mile #, offset, lane and direction). Once an accident is registered, every vehicle in the 5 upstream segments at the time of accident detection must be notified that there is a downstream accident so that it can seek an alternate route if a credit is desired.[2] Notification comes by way of a tuple of the form,

$$(\texttt{Type} = \texttt{1},\ \texttt{VID},\ \texttt{XWay},\ \texttt{Mile},\ \texttt{Dir})$$

such that `Type` identifies this tuple as an accident alert, `VID` identifies the vehicle to which the alert is sent, and `XWay`, `Mile`, and `Dir` identify the segment where the accident occurred. Beyond notification, accident detection also affects toll calculations as was discussed previously.

Table 3.3 summarizes the schemas for output data generated by the continuous queries for Linear Road.

3.2.2 Historical Queries

Beyond continuous queries, systems implementing Linear Road must also allow vehicles to issue three kinds of *historical* queries: a query for the balance of the account associated with the vehicle (*Account Balance*), a query requesting the sum total of expenditures for the vehicle on a given day on a given expressway (*Daily Expenditure*), and a query requesting a prediction of travel time and toll charges for a specified trip on a specified day and time (*Travel Time*). These queries are described in more detail below.

[2]Note that vehicles that enter one of these segments after the time of accident will not be notified of the accident but will still receive a credit for exiting prior to the accident site.

Account Balance: A customer traveling on the expressway can request his account balance at any time. The balance reported must be accurate through the last segment completed (i.e. the one previous to the one he is currently in). The account balance response must take into account all tolls which have been charged to the account during the simulation. This historical query is triggered by the traveler issuing a tuple of the form,

$$(\texttt{Type} = 2,\ \texttt{Time},\ \texttt{VID},\ \texttt{QID}),$$

such that `Time` is the time the query was issued, `VID` is the identifier for the vehicle issuing the query request (and for which a balance is to be reported), and `QID` is a query identifier. In response, a tuple of the form,

$$(\texttt{Type} = 3,\ \texttt{QID},\ \texttt{Balance})$$

is issued such that `QID` identifies the historical query this tuple answers and `Balance` is the account balance of the vehicle from which the query was posed.

Daily Expenditures: Aside from cumulative balances, a traveler can also request his total toll expenditures today for a given expressway. This historical query is triggered by a tuple of the form,

$$(\texttt{Type} = 3,\ \texttt{Time},\ \texttt{VID},\ \texttt{QID},\ \texttt{XWay}),$$

such that `Time` is the time the query was issued, `VID` is the identifier for the vehicle issuing the query request (and for which a day's total expenditures is to be reported), `QID` is a query identifier, and `XWay` identifies the expressway for which a daily expenditure report is desired. In response, a tuple of the form,

$$(\texttt{Type} = 3,\ \texttt{QID},\ \texttt{Expenditure})$$

is generated such that `QID` identifies the historical query this tuple answers and `Expenditure` is the day's toll expenditure for the expressway identified in the matching query tuple that triggered this response. Note that this total includes not only tolls, but credits applied due to accidents.

Travel Time Prediction: At any time, a traveler can request a prediction as to the travel time and total toll charge for a future journey between any two segments on the same expressway that begins on any day at any time. This request is issued with a tuple of the form,

$$(\texttt{Type} = 4,\ \texttt{Time},\ \texttt{VID},\ \texttt{QID},\ \texttt{XWay},\ \texttt{M}_{\text{init}},\ \texttt{M}_{\text{end}},\ \texttt{DOW},\ \texttt{TOD})$$

such that `Time` is the time the query was issued, `VID` is the identifier for the vehicle issuing the query request, `QID` is a query identifier, `XWay` is the expressway upon which the journey occurs (from mile $\texttt{M}_{\texttt{init}}$ to mile $\texttt{M}_{\texttt{end}}$), and `DOW` (day of week) and `TOD` (time of day) specify the day and time when the journey would take place.

The predicted travel time and toll charge is calculated on the basis of statistics maintained over the previous 10 weeks for all segments involved in the journey. Given a journey specified from segment k to $k+n$ on expressway x, for time t on day d, the predicted travel time is,

$$a = \sum_{i=0}^{n} a_i,$$

such that

- time a_0 is the time it takes to travel segment i as determined by averaging the `LAV` for segment i at time t on day d over the previous 10 weeks,

- time a_1 is the time it takes to travel segment $i+1$ as determined by averaging the `LAV` for segment $i+1$ at time $t + a_0$ on day d over the previous 10 weeks,

- time a_n is the time it takes to travel segment $i+n$ as determined by averaging the `LAV` for segment $i+n$ at time $t + a_0 + \ldots + a_{n-1}$ on day d over the previous 10 weeks.

The predicted tolls for each segment should be based on the predicted times of travel for each segment. That is, the total predicted toll is,

$$b = \sum_{i=0}^{n} b_i,$$

such that

- toll, b_0, is the toll calculated according to the toll calculation rules described in Section 3.2.1 based on the 10 week average speed and count of vehicles in segment i at time t on day d,

- toll, b_1, is the toll calculated according to the toll calculation rules described in Section 3.2.1 based on the 10 week average speed and count of vehicles in segment $i + 1$ at time $t + a_0$ on day d,

- toll, b_n, is the toll calculated according to the toll calculation rules described in Section 3.2.1 based on the 10 week average speed and count of vehicles in segment i at time $t + a_0 + \ldots + a_{n-1}$ on day d.

Travel and toll predictions may assume that all travel takes place on a single day and ignore the frequent traveler discount.

Upon calculating predicted travel time and toll, the system responds with a tuple of the form,

$$(\texttt{Type} = \texttt{4},\ \texttt{QID},\ \texttt{TravelTime},\ \texttt{Toll})$$

such that `QID` identifies the historical query this tuple answers, and `TravelTime` and `Toll` are the predicted travel time and toll charge for the vehicle journey calculated in the manner described above.

The schema for output tuples for historical queries issued in Linear Road are summarized in Table 3.4.

3.2.3 Historical Data

To be able to satisfy the continuous and historical queries of Linear Road, a stream data management system must maintain a certain amount of historical data. Specif-

Query Response	*Schema*
Account Balance	`(Type = 2, QID, Balance)`
Daily Expenditure	`(Type = 3, QID, Expenditure)`
Travel Time	`(Type = 4, QID, TravelTime, Toll)`

Table 3.4: Output Tuple Schemas: Historical Queries

ically, the system must maintain 10 weeks of statistical data about each segment on the Linear Road expressways, as well as account and location data for every registered vehicle. The exact historical data requirements are described below.

Segment Statistics: Systems implementing Linear Road must maintain 10 weeks worth of statistical data for each segment on the Linear Road expressways. This data is used for calculating tolls (*Toll calculation* query) and computing travel time and toll estimates (*Travel time query*). The data that must be maintained for each of the $L \times 200$ segments includes the following.

- A count (`Count`) of the number of vehicles in the segment maintained with granularity of 1 minute starting with any arbitrarily chosen time. The vehicle count should be computed at time t by identifying the most recent message from each vehicle as of t, assigning the vehicle to the segment in which it is located at that time, and counting the vehicles in each segment. The arbitrarily chosen initial time of computation need not be uniform across segments.

- The "Latest Average Velocity" (`LAV`) in each direction for each one-mile segment. This is computed as the average speeds of the vehicles in each segment, and is computed every minute by averaging the speeds of all position reports issued in the previous 5 minutes.

Note that 10 weeks of historical data at 1 minute granularity for every segment requires maintaining

$$(200 \cdot L \ \textit{segments}) \times (100,800 \ \textit{minutes}),$$

or approximately $20L$ million records (for L = number of expressways being monitored).

Vehicle Accounts: Systems implementing Linear Road must maintain for every registered vehicle, its current account balance and position, as well as 10 weeks worth of data on tolls paid per expressway. This data is used for determining current account balances (*Account Balance*), daily expressway expenditures (*Daily Expenditure*), frequent traveler discounts (*Toll Calculation*) and recipients of accident alerts (*Accident Detection*). The data that must be maintained for each of the 1 million registered vehicles includes the following:

- the current balance of the vehicle's account, accurate to the last segment in which the vehicle completed travel. This is used to respond to *Account Balance* queries.

- the balance paid on each expressway today. This is used to respond to *Daily Expenditure* queries, as well as to determine if a driver is eligible for a "frequent traveler" discount on tolls calculated by the *Toll Calculation* query.

- the current segment where the vehicle is located. This is used to determine which vehicles should receive alerts as a result of the Accident Detection query.

Note that current account balances and positions require 1 million records to be maintained (1 for each vehicle), whereas daily expenditures require L records per vehicle (one for each expressway), or L million records in all.

Accidents: Systems implementing Linear Road must keep track of all current accident locations and the times of their detection. This data is used to determine tolls (segments in proximity to an accident issue no tolls but issue credits to vehicles that leave the expressway), and issue accident alerts.

Query Identifiers: Every historical query request tuple includes the vehicle ID for the driver issuing the request and a query ID for the request itself. This data must be maintained so that a tuple result (which includes the identifier for the issued query) can be routed to the appropriate vehicle(s).

3.3 Evaluation Criteria

In this section, we describe the performance and accuracy requirements and validation procedure for systems implementing Linear Road. These requirements must be met on a single processor box running Linux, and with no more than 2.5 Ghz processing power, 1 Gigabyte of RAM and a 512K cache. A system achieves an *L-rating* if it meets these performance and correctness objectives while supporting L expressways ($0 \leq L \leq 9$).[3]

3.3.1 Query Performance Requirements

The quality of service requirements by the benchmark system are summarized for all possible system outputs in Table 3.5. A toll notification must be issued quickly so as to allow drivers the option to exit a segment prior to being charged. Thus, the notification must be sent within 15 seconds of the time of the position report that alerted the system that the vehicle had entered the segment in question. Similarly, an accident notification must be issued within 15 seconds of the position report which leads to its detection, so as to allow drivers to exit the expressway.

Historical queries have less strict real-time requirements. Account balances are valid provided that they are returned within a minute of their request, Daily Expenditure totals should be returned within a minute and a half, and a Travel Time query should be answered within 2 minutes after the query has been issued.

3.3.2 Query Accuracy Requirements

This section presents the minimum requirements for correct values that are imposed on a Linear Road implementation. A system that meets these requirements will

[3] Each expressway is simulated independently by the Simulation Tool, although a vehicle is prevented from being placed on two expressways concurrently.

Query	*Performance Requirement*
Toll Notifications	15 seconds from the time a vehicle reports its position in a new segment.
Accident Notification	15 seconds from the last report establishing two vehicles being stopped in the same position.
Account Balance	60 seconds from the time the query was issued.
Daily Expenditure	90 seconds from the time the query was issued.
Travel Time	120 seconds from the time the query was issued.

Table 3.5: Query Response Requirements

pass the validation system. The actual validation system may not check all of these requirements, or may impose less stringent requirements. But an implementation is still required to meet these performance and accuracy requirements.

Accuracy requirements for all historical data are given in Table 3.6. Accuracy requirements are given in terms of a time measure specifying an "allowable staleness". For example, the accuracy requirement for Vehicle Positions is 30 seconds. This means that if the vehicle v is reported as located at segment s as of time t, then v should have emitted a position report indicating its location as s at some time between (t $-$ *30 sec*) and t. Similarly, the account for a vehicle should be accurate to within 60 seconds. In other words, a balance response must be correct for some time between when a balance request query was issued and when the response was delivered. If an accident is reported on segment s as of time t, then it should be the case that an accident is still being cleaned up as of some time between (t $-$ *30 sec*) and t. (I.e., the accident was determined as having occurred at some time between (t $-$ *20 min, 30 sec*) and t). Segment data is required to be recomputed once per minute, based on the last 5 minutes of position reports. For a position report arriving at time t and a corresponding toll notification delivered at time $t+\delta$, there must exist some offset o between 0 and $(60+\delta)$ seconds such that the toll report is accurate given the position notifications delivered between time $(t+\delta-o)$ and $(t+\delta-o-5$ minutes). Travel time responses (not specified in Table 3.6) are allowed a 5% margin of error, due to the complexity of the calculation.

Historical Data	*Accuracy Requirement*
Vehicle Positions	30 seconds.
Vehicle Accounts	60 seconds.
Accidents	30 seconds.
Segment Statistics	60 seconds.

Table 3.6: Transaction Requirements

3.3.3 The Linear Road Validation Tool

The output of stream systems implementing Linear Road should be a flat file consisting of tuples that have been be padded as necessary to occupy 20 bytes each, with 5 4-byte integer fields of which the 1st is a type (valued from $1 \dots 5$) that is used to identify the query to which the tuple is a response.

Validation involves comparing the system's output with that generated as a reference set by the validation tool for the given input. The validation tool will read output from the flat files generated by the stream system and check the results to see if they meet the performance and accuracy requirements described in Section 3.3. It is expected that most systems will produce accurate output, but will for some scale factor, be unable to continue meeting the quality of service guidelines.

Chapter 4

Implementation of the Linear Road Test Harness

This section describes briefly the implementation of the test harness for the Linear Road benchmark. The test harness includes the software which generates the test data and queries, the program that delivers data in real-time to the implementation of Linear Road, and the system that receives output from the system being benchmarked and checks it for correctness.

4.1 Traffic Simulation

Traffic simulation is an important topic in transportation research. Testing new traffic management systems on actual travellers is expensive and difficult. While data recorded from an actual transit network may be fed to a system, this technique is limitting. Such data cannot help to predict reactions of travellers to new information. Also, it is not useful in modelling changes in methods of data collection.

There are two major classes of traffic simulators. Macroscopic traffic simulators and microscopic traffic simulators. Macroscopic traffic simulators use general models to predict the behavior of transportation networks. In contrast, microscopic simulators use simple models of individual travellers' behavior and attempt to model every vehicle and traveller in the network. Microscopic models, which are more computationally intensive, are preferred for offline simulation and evaluation of new systmes.

4.1.1 MITSIMLab

MITSIM[Yan97] is a microscopic traffic simulator developed by the MIT ITS program[ITS]. MITSIM has support for integrating arbitrary external systems[FR00] which should make it suitable for this work. It is widely used in industry and academa.

In the Linear Road test harness, MITSIMLab is driven by a wrapper script (implemented in Perl) which accepts the Linear Road parameters, and is responsible for generating many of the random parameters, such as the precise distribution of automobile origins and destinations, and the location of accidents.

4.2 Data Delivery and Output Validation

Data is delivered to the system being benchmarked over the network by the same program that receives outputs and checks correctness. The delivery of data is simple, the program checks the generated timestamp on each tuple in its input file and delivers the tuple at the appropriate time.

Currently minimal output verification is done. Toll notifications and other response are verified to have come within appropriate time-bounds. This is sufficient for basic benchmarking purposes. A stronger validation system will be neccessary if the benchmark is to be more widely used.

Chapter 5

Linear Road Implementation in Aurora

In order to validate the benchmark, and to test the test harness, I did an implementation of Linear Road in the Aurora system. Performance statistics on this implementation are presented in Chapter 6. In this chapter I describe the design of the query network and the implementation of the custom operators and functions required to implement Linear Road.

5.1 Linear Road Query Network

In Aurora, query networks are designed using a graphical tool, the Aurora GUI. This tool supports a "workflow" model of queries. Networks are drawn as boxes and arrows by the programmer and the properties of the boxes are specified. Information conceptually flows from left to right, like water through pipes. In the case of Linear Road, nonlinearities are introduced into the network via Relations, described in Section 5.2.

The entire Linear Road network is shown in Figure 5-1. The historical queries are each implemented with a single box that reads data from a relation, and outputs its results directly. The continuous parts of the query can be decomposed into three subqueries. The center subquery manages accident detection. The left sub-

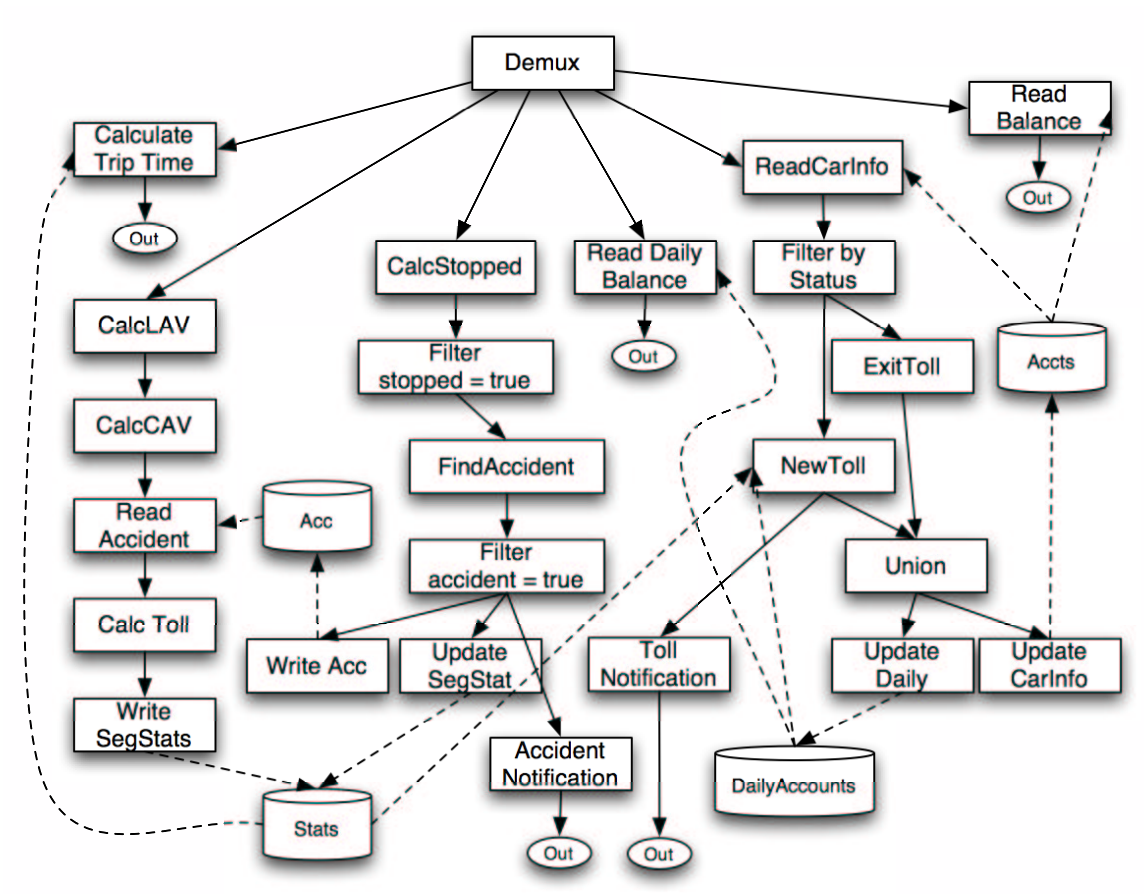

Figure 5-1: Aurora query network for Linear Road

query aggregates car position data to calculate statistics for each segment. The right subquery calculates a toll for each vehicle, and charges tolls when appropriate. These three subqueries are described in detail in following subsections.

5.1.1 Accident Detection

An accident is to be detected if two cars are in the same position for 4 consecutive position readings. To calculate this, accident detection uses a custom aggregate function to look at the last four readings for each car. If a car is stopped, a tuple is emitted, to be aggregated by the next box, which looks for two cars both not moving in the same position. when this aggregate function emits a tuple, an accident is detected. When an accident is detected, data about it is saved in one relation, the `Accidents` relation, and the tolls for cars that are on the effected segments are updated in another relation, `Accounts`.

5.1.2 Segment Statistics

The segment statistics is responsible for simple aggregate functions that calculate the rolling average over the last 20 minutes of the speed and number of cars on each street segment. Additionally, the subquery reads the `Accidents` relation to determine if there is an accident on each segment. These statistics are written out to the `SegStats` relation, which is used in toll calculation.

5.1.3 Toll Calculation

The toll calculation subquery is the most complex. It is also the critical path, because performance in the benchmark is measured by the load supported while a maximum 15 second latency between position report and toll response is maintained. This subquery is responsible for creating the toll response, and follows the decision tree from Figure 3-1.

For each car position report that comes in, this subquery looks up to see if the cars has been on the road before. If the car is entering a new segment of the road, and was last seen on the previous segment of road, then the toll for the previous segment needs to be charged. In addition, the toll for this segment needs to be calculated, and the car notified of that toll. If the car is seen on an offramp, then no toll is charged unless there is an accident, in which case the negative toll needs to be charged. Whatever toll needs to be charged is used to update the `Accounts` relation, as well as the `DailyBalance` relation.

5.1.4 Other Queries

Other queries originating with cars on the road are handled as simple lookups into relations maintained by the rest of the system. Specifically, one special operator each for looking up daily expenditures, balance, and estimated trip time.

5.2 Relations

Linear Road identified the need to store data persistently with an Aurora query network (a more appropriate term may be "Aurora application"). This storage was accomplished using the new (to streaming databases) concept of a *relation*. A relation, like a database table, is identified with a global name and has a specified schema. Relations allow persistent data storage, such as account balances. Relations can be read and updated using a special operator box. Any relation can be read or updated from anywhere in the network, and as a result they introduce nonlinearities.

At the time of the Linear Road implementation, full-fledged relations are not available in Aurora. Limitted relations, offering only the capability strictly required by Linear Road, were implemented as custom operators in Aurora.

The impact of relations on the streaming database model is still being studied.

5.3 Synchronization

Another challenge in implementing Linear Road was synchronization in the query network. Relations lead to conceptual loops in the processing, where processing in two parts of the network is interdependent because of updating and reading the same relation. Because computation does not occur instantaneously, but at the whim of the scheduler, this leads to non-determinism. Worse, there is no way to delay one part of a query network while another part of the network completes the update of a relation.

This problem is similar to the need for transactions in classic databases. However, rather than enforcing update order independence, what is needed is some way of enforcing a particular order on computation. The proposed solution to this problem, a synchronization operator that can delay a stream until a predicate is fulfilled, has not yet been implemented. Thus, it is not yet incorporated into the Aurora implementation of Linear Road. However, this operator should cover all the synchronization needs of Linear Road, in particular delaying computation while waiting for a table to update.

5.4 Custom Operators and Functions

Because Aurora is a research system, Linear Road required much functionality not present in the current system. For example, the set of functions in the predicate language, as well as the selection of built-in aggregate functions, was insufficient. Because Aurora allows the programmer to specify custom functions, aggregate functions, and operators, it was relatively easy to add this functionality. In this section, I will describe the more interesting additions.

5.4.1 Read and Update Relations

Relations, described in Section 5.2, were required to implement Linear Road. Special case relations were implemented, based on the Sleepycat DB library [Sle03]. Only the specific functionality required by Linear Road was implemented.

5.4.2 Aggregates Returning Multiple Values

In the Segment Statistics subquery (see Section 5.1.2), multiple aggregate functions need to be calculated over the same window of data. For efficiency and consistency, only one copy of this data should be maintained and only one pass should be made over the data. To accomplish this, aggregate functions returning more than one value were added to the system. They required modification of the type checking code in the Aurora GUI.

5.5 Wrapper

The Aurora system exists as a C++ library that can be linked into an application. In order to support the network interface specified in the Linear Road specification, I built a small application that connects to the test harness over the network, enqueuing received data to Aurora and sending data dequeued from Aurora over the network to the test harness. This application was based on the standard Aurora workload generator (wlgen) daemon, written by Ying Xing.

Chapter 6

Performance Measurments

At the current time we have solid performance measurements only for a preliminary version of the Aurora system. This system, with the implementation of Linear Road described in this paper, scored a 1 on the benchmark. That is, it was able to run a single expressway at rush-hour rates on the benchmark hardware platform. With a redesigned operator set, optimized support for Relations, and many other implementation improvements, a much improved level of performance should be realized in the near future.

At this time no confirmed numbers are available from the Stanford STREAM project or the MIT Medusa project.

Chapter 7

Related work

Related work in this area falls into two categories: identifying applications for stream processing and benchmarking database. These will be treated seperately.

7.1 Stream Processing Applications

There are a number of applications suitable for continuous query processing. Babu et al [BSW01] identified its use for computer network management and research. Applying continuous query processing to highway data collection is discussed in the Fjords paper by Madden and Franklin[MF02]. This work focusses on the challenges of dealing with low power (both computation and communication) sensors. The queries supported by the system are very simple, aggregating data from a small number of nearby traffic sensors.

7.2 General Database Benchmarking

Historically databases have been the focus of much benchmarking effort. Thoughout the 1980's and up through the present time, On-Line Transaction Processing (OLTP) has been an important performance concern for corporations investing in database technology. Benchmarks in this area attempt to measure the number of transactions that can be handled per second. The original industry-wide benchmark in this area, DebitCredit[Ano85], tested very simple transactions in a simple data set. From that benchmark came a trade organization, the Transaction Processing Performance Council (TPC). The TPC created a variety of benchmarks, designating them with letters. These benchmarks initially focussed on transaction processing[TPC94a], changing to match the new capabilities of database systems[TPC94b] [TPC02a]. TPC later branched out into other performance metrics considered important for core corporate databases, such as ad hoc queries and report generation[TPC98] [TPC02b] [TPC02c] [TPC02d].

The "Wisconsin" benchmark was the first major benchmark to test the relational capability of Relational Database Management Systems (RDBMSs). The benchmark is described in a 1983 paper[DBT83]. It uses completely synthetic data, and is not based on any application. This lead to some criticisms about the realism of the benchmark[Gra93].

The next major area of database development was in object-oriented databases. Work in this area eventually resulted in several benchmarks, including OO7[CDN93] for true object-oriented databases, and the BUCKY[CDN$^+$97] benchmark for hybrid object/relational databases.

There also exist a number of benchmarks for scientific databases. Because business database users dominate the market, there is a need for customized benchmarks focussing on scientific users. One such benchmark is Sequoia 2000[SFGM93]. Most case of scientific databases involve benchmarking for storage of particular complex datatypes, or for particular complex queries. As more application areas for streaming databases are developed, such specific benchmarks may become neccessary.

Chapter 8

Conclusions

Returning to Gray's four criteria for database benchmarks, we can reflect on our success:

Relevant Linear Road is relevant, dealing with an application that is clearly a good use of streaming databases. Time, and more development of streaming database applications, will tell how generalizable the benchmark is.

Portable Because Linear Road is an application-level benchmark, it is portable across any implementation language, and even across any set of primitive database operations.

Scalable Linear Road can be scaled by simply simulating more roads, placing a heavier load on the database.

Simple Linear Road, by dealing with an easy to understand application, is easy to explain to a variety of audiences.

Based on these criteria, Linear Road should be a success as a database benchmark.

One major outcome of developing this benchmark has been the influence over the design of the Aurora engine and its operator set. During development of the Linear Road benchmark, Aurora was undergoing major design work. In particular, the set of operations for building applications was being resigned. Linear Road provided an example of a “real” application. It was important to an understanding of what functionality would be offered by Relations (see Section 5.2), as well as in arguing for high performance latch functionality. Synchronization was found to be a big problem, and is leading to the introduction of a new “Wait-for” operator (see Section 5.3). Linear Road also showed that real applications make heavy use of custom aggregate functions, and generally need more than basic aggregate functionality. The resulting operator set is more extensive, and should prove easier to use when implementing real applications.

Linear Road has also helped to bring the streaming database community together to compare actual performance numbers and discuss operator sets. This initially happened at the SIGMOD 2003 conference, and will hopefully continue.

The major conclusion to be drawn from Linear Road is that a more precisely specified benchmark would be better in many ways. However, this will have to wait until there is clear agreement on a language and model which will be implemented by streaming databases across the board.

Bibliography

[ACC+03] Daniel Abadi, Don Carney, Ugur Cetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(3), August 2003. To appear.

[Ano85] Anon et al. A measure of transaction processing power. *Datamation*, 31(7):112–118, April 1985.

[BSW01] Shivnath Babu, Laskshminarayan Subramanian, and Jennifer Widom. A data stream management system for networked traffic management. In *Proceedings of Workshop on Network-Related Data Management (NRDM 2001)*, May 2001.

[BW01] S. Babu and J. Widom. Continuous queries over data streams. *ACM SIGMOD Record*, 30(3):109–120, September 2001.

[CCD+03] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. Telegraphcq: Continuous dataflow processing for an uncertain world. In Michael Stonebraker, Jim Gray, and David DeWitt, editors, *Proceedings of the 1st Biennial Conference on Innovative Database Research (CIDR)*, Asilomar, CA, January 2003.

[CCR+03] Don Carney, Ugur Cetintemel, Alex Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream environment. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003. To Appear.

[CDN93] Michael Carey, David DeWitt, and Jeffrey Naughton. The oo7 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 2–11. ACM Press, May 1993.

[CDN+97] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes E. Gehrke, and Dhaval N. Shah. The BUCKY object-relational benchmark. In *Procedings of the ACM SIGMOD*, pages 135–146, May 1997.

[CDTW00] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In Jeff Naughton Weidong Chen and Phil Bernstein, editors, *Proceedings of the Special Interest Group on Management of Data (SIGMOD)*, Dallas, Tx, June 2000.

[DBT83] David DeWitt, Dina Bitton, and Carolyn Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 1983 Very Large Database Conference*, October 1983.

[DEF+88] C. Anthony DellaFera, Mark W. Eichin, Robert S. French, David C. Jedlinsky, John T. Kohl, and William E. Sommerfeld. The zephyr notification service. In *Proceedings of the Winter 1988 USENIX Technical Conference*, pages 213–219. USENIX, February 1988.

[FR00] Bruno Miguel Fernańdez Ruiz. Architecture for the integration of dynamic traffic management systems. Master's thesis, Massachusetts Institute of Technology, 2000.

[Gar02] Garmin Incorporated. *GPS V, Personal Navigator Owner's Manual and Reference Guide*, 2002. "URL: http://www.garmin.com/manuals/gpsv.pdf".

[Gra93] Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition).* Morgan Kaufmann, 1993.

[ITS] MIT ITS. Mit intelligent transport systems program. http://web.mit.edu/its/.

[ITS02] Conjestion Pricing: A Report From Intelligent Transportation Systems (ITS), May 2002. URL: www.path.berkeley.edu/ leap/TTM/Demand_Manage/pricing.html.

[MF02] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of the 18th International Conference on Data Engineering (ICDE 2002)*, 2002.

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olsten, Justin Rosenstein, and Rohit Varma. Query processing, resource management, and approximation in a data stream management system. In Michael Stonebraker, Jim Gray, and David DeWitt, editors, *Proceedings of the 1st Biennial Conference on Innovative Database Research (CIDR)*, Asilomar, CA, January 2003.

[Poo02] Robert W. Poole. HOT Lanes Prompted by Federal Program, November 2002. URL: http://www.rppi.org/federalhotlanes.html.

[SFGM93] Michael Stonebraker, James Frew, Kenn Gardels, and Jeff Meredith. The sequoia 2000 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993*, pages 2–11. ACM Press, May 1993.

[Sle03] Sleepycat Software. The Berkeley Database. http://www.sleepycat.com, 2003.

[TCZ+03] Nesime Tatbul, Ugur Cetintemel, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Load shedding in a data stream manager. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB)*, September 2003. To Appear.

[TMS03] Peter A. Tucker, David Maier, and Tim Sheard. Applying punctuation schemes to queries over continuous data streams. *IEEE Data Engineering Bulletin*, 26(1):33–40, March 2003.

[TPC94a] Transaction Processing Performance Council (TPC). *TPC Benchmark A: Revision 2.0*, June 1994.

[TPC94b] Transaction Processing Performance Council (TPC). *TPC Benchmark B: Revision 2.0*, June 1994.

[TPC98] Transaction Processing Performance Council (TPC). *TPC Benchmark D (Decision Support): Revision 2.12.0.1*, 1998.

[TPC02a] Transaction Processing Performance Council (TPC). *TPC Benchmark C: Revision 5.1*, December 2002.

[TPC02b] Transaction Processing Performance Council (TPC). *TPC Benchmark H (Decision Support): Revision 2.0.0*, 2002.

[TPC02c] Transaction Processing Performance Council (TPC). *TPC Benchmark R (Decision Support): Revision 2.0.0*, 2002.

[TPC02d] Transaction Processing Performance Council (TPC). *TPC Benchmark W (Web Commerce): Revision 1.8*, February 2002.

[USD03] A Guide for Hot Lane Development: A U.S. Department of Transportation Federal Highway Administration White Paper, March 2003. URL: www.itsdocs.fhwa.dot.gov/JPODOCS/REPTS_TE/13668.html.

[Yan97] Qi Yang. *A Simulation Laboratory for Evaluation of Dynamic Traffic Management Systems*. PhD thesis, Massachusetts Institute of Technology, February 1997.

[YK96] Qi Yang and Haris N. Koutsopoulos. A microscopic traffic simulator for evaluation of dynamic traffic management systems. *Transportation Research C*, 4(3):113–129, June 1996.