UPTEC F07 105
Examensarbete 20 p
November 2007

Benchmarking the performance of a data stream management system

Mårten Svensson

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Benchmarking the performance of a data stream management system

Mårten Svensson

The area of stream processing applications has gained a lot of attention from the database research community recently. A data stream management system executes continuous and historical queries over data streams and produces result streams in real-time. The Linear Road benchmark evaluates the performance of stream database management systems. It specifies a fictional traffic system of expressways with variable tolling.

This Thesis presents an implementation of the Linear Road benchmark for the SCSQ data stream management system. The purpose is to evaluate the performance of SCSQ and compare it with other data stream management systems.

The implementation scored L=1.5 on the benchmark, which means it was able to process data from one and a half expressway and deliver results within the specified time frame. The evaluation was made on a Windows-based laptop.

Handledare: Erik Zeitler
Ämnesgranskare: Tore Risch
Examinator: Tomas Nyberg
ISSN: 1401-5757, UPTEC F07 105
Tryckt av: Ångstömslaboratoriet Uppsala

Contents

1 Introduction

The area of stream data management has gained a lot of interest from the database research community during the passed few years. As the number of digital system increases there is an emerging set of applications that involve processing large volumes of continuous data such as sensor data, stock markets, digital audio, and images. In many applications the data rate is high and it is not possible to save all the data on disk. A number of data stream management systems (DSMSs) have been developed to handle high volume data streams, including Aurora[1], STREAM[2] and TelegraphCQ[3]

In order to evaluate different DSMSs against each other the Linear Road benchmark[10] has been developed. It specifies an application to be implemented rather than specifying exact queries. The Linear Road simulates a system of expressways with variable tolling. The basic idea of the Linear Road benchmark is to evaluate the performance of a DSMS while processing large volumes of data and historical queries. The implementation of the benchmark must be able to produce accurate results as well as answering to the real time response requirements. The benchmark includes a validation program that verifies that the measured implementation works correctly.

At Uppsala University a DSMS called SCSQ[6][7][8] has been developed. This Thesis presents the SCSQ-implementation of the Linear Road benchmark, called SCSQ-LR. Even though SCSQ also supports parallel stream queries the benchmark runs on a single computer. The purpose of this Thesis is to see how well SCSQ performs and compare the results to other implementations of Linear Road Benchmark.

2 Background

2.1 Data Stream Management System

Traditional data base management systems are powerful tools for examining fixed collections of data. A data stream management system (DSMS) is different because it deals with execution of queries over infinite streams of data. A data stream consists of a continuously incoming stream of *input event* tuples, describing e.g. measurements. The incoming data rate may be very high or come in bursts. Based on the input events, streams of *output events* are computed as continuously running queries. An output event is calculated as soon as the required input events become available. The output from a DSMS will itself be one or several streams of events computed from the input events. Each incoming event is optionally associated with a time stamp. The time stamp describes the time the event arrives to the system. Each event can normally only be read once from the stream and it is not possible to change the order that events arrive. A typical stream could be output from a sensor, such as position reports from a car or data from a radio telescope.

Stream processing problems often make explicit use of the time stamp associated with the tuple. An example of a problem is to calculate the average speed of a vehicle on a freeway during the last 5 minutes. For some problems there can also be real-time requirements. That is, the result must be presented within a specified amount of time after the required data becomes available. For example, an accident has to be detected within 15 seconds from when it occurred.

2.2 SCSQ

SCSQ (Super Computer Stream Query processor) [6][7][8] is a DSMS prototype extending the AMOS II main memory database system [9]. SCSQ has been developed to deal with the challenges of queries over high volume data streams. The SCSQ prototype runs on a variety of hardware platforms, including Windows, Linux, and IBM BlueGene massively parallel computers. SCSQ enables high level specification of distributed stream queries involving advanced parallel computations in such heterogeneous communication and computation environments. SCSQ queries filter, transform, and join data from different kinds of distributed streaming data sources. In this Thesis work only non-parallel, single node, SCSQ queries are used. The design of the SCSC DSMS is illustrated in figure 1.

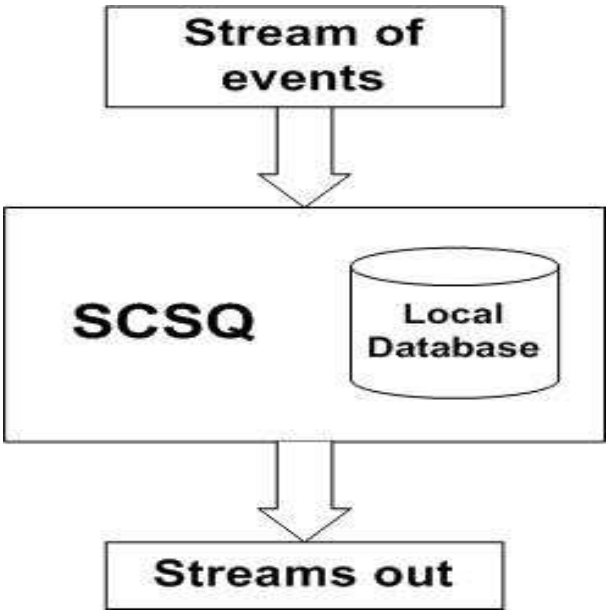

Figure 1: SCSQ

2.3 Linear Road Benchmark

This section summarizes the Linear Road benchmark. A more detailed description of the Linear Road benchmark can be found in [10]

2.3.1 Linear Road overview

The Linear Road is so far the only benchmark for DSMSs. Since there is no common language standard for specifying queries for streaming databases the benchmark specifies an application to be implemented rather than specifying queries.

Linear Road simulates a traffic system of expressways with variable tolling. Tolls are calculated based on traffic congestion and accident proximity.

Linear City is a fictional city, which is the setting for the Linear Road benchmark. The city consists of a number (L) of parallel expressways. Each expressway is 100 miles long and divided into 100 segments. Every expressway has four lanes running in each direction, three lanes are normal travel lanes and the fourth lane is both an entrance and exit ramp. Figure 2 shows the geometry of a segment on an expressway in the benchmark.

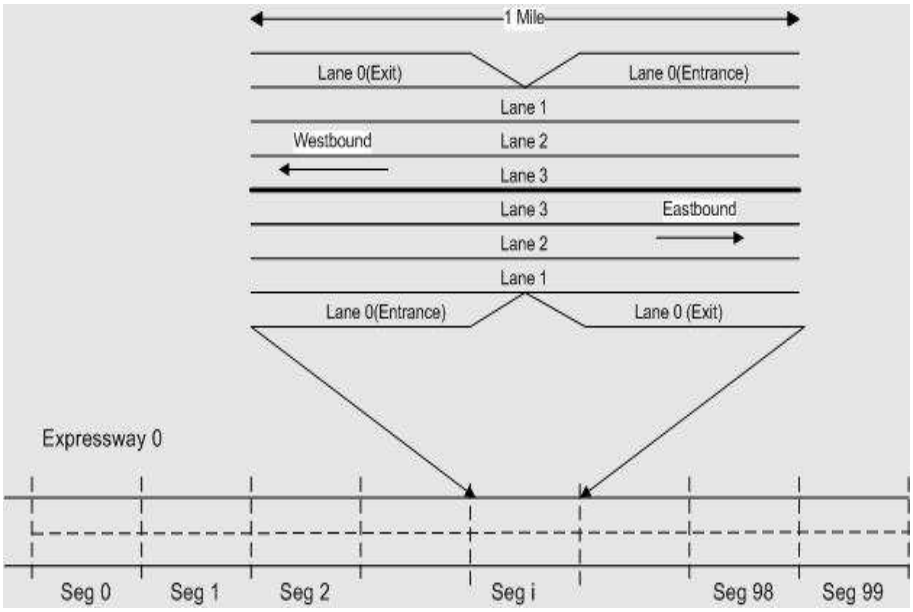

Figure 2: Geometry of an expressway in Linear Road

The input events for the benchmark is generated by the MIT Traffic Simulator (MITSIM) [5] and stored in a data file. The simulator creates a number of vehicles that undertakes at least one journey during the simulation. Each journey begins on an entry ramp and finishes on an exit ramp. Each car is equipped with a sensor that emits position reports. The simulator ensures that each car emits a position report every 30 seconds, staggering them so that every second roughly 1/30 of the events for the vehicles currently on the expressways are emitted. A vehicle never travels faster than 100mph; this ensures that every car emits at least one event from every segment it travels on. Furthermore every vehicle is guaranteed to average 40mph or less when entering or leaving an expressway. This ensures that every vehicle will leave at least one position report from an entrance ramp and one position report from an exit ramp for every trip.

The simulator creates an accident in a random location on each expressway for every 20 minutes of position reports. An accident occurs when at least two cars are stopped at the same position at the same time. A car is considered stopped if it emits four consecutive position reports from the same position. An accident takes anywhere between 10 and 20 minutes to be cleared, once detected. During the clearance the cars involved in the accident continues to emit position reports.

The simulator also ensures that with 1% probability a report is accompanied with a historical query request. About 50% of the requests are account balances, 10% are requests for daily expenditures, and about 40% are requests for travel time predictions.

The input events generated by the simulator consists of four types of tuples, *position reports*, and three types of historical query requests, *account balance, daily expenditure*, and *travel time estimation*. All generated events are stored as vectors in a file. The simulator also generates historical data to be used by the benchmark.

2.3.2 Query Requirements

The Linear Road benchmark requires the implementation to process the set of continuous and historical queries with correct result as well as produce prescribed output events with real-time response requirements. The continuous queries in Linear Road demand real-time processing of the streaming data whilst the historical queries require the implementation to keep track

over a large volume of data and present data for answering queries.

There are two types of continuous queries that have to be implemented: *toll processing* and *accident detection*. Aside from the continuous queries there are three types of historical queries: *account balance, daily expenditure*, and *travel time estimation*.

- **Toll Processing**
  A system implementing Linear Road has to calculate a toll every time an incomping event describes a position report for a car in a new segment, in order to notify the vehicle of this toll. The toll depends on the number of cars and the average speed in a given segment as well as the proximity of an accident. The toll is notified to the driver as an output event as soon the car enters the segment but the toll is not being charged to the car until it enters a new segment. If the vehicle exits on the exit ramp the toll for that segment will not be charged to the account. A system implementing Linear Road must keep track of all tolls assessed so it is possible to answer historical queries concerning account balances. The response time, i.e. the time from the input position report comes into the system to the time the output toll notification event is emitted, must not exceed five seconds.

- **Accident Detection**
  The system must detect an accident if two vehicles are stopped at the same position. A car is considered stopped if it emits four consecutive reports from the same position. Once an accident is detected an accident notification as an output event is sent to every car that comes in proximity of the accident (five segments upstream from the accident). That gives the driver a chance to leave the expressway and thereby reduce the risk for large congestions. The response time from the position report (input event) that triggers an accident to the accident notification event must not exceed five seconds.

- **Historical Queries**
  The Linear Road benchmark defines three types of historical queries, account balance, daily expenditure, and travel time estimation. These queries are specified as input events, *query events*. The account balance response must take into account all tolls which have been charged to the vehicle during the simulation. This query must be answered within five seconds from the input query event.

  The daily expenditure response must compute the sum of tolls charged

to a vehicle for a given expressway and a given day from the last ten weeks. The response must be issued within ten seconds from the input query event. The response to a travel time estimation query requires the system to calculate a prediction of the time to drive between two given segment based on statistics from the previous ten weeks.

The answer to a historical query will be an output event.

2.3.3 Validation Requirement

Besides the real time requirement to produce the result within five seconds there are also accuracy requirement. For the account balance the accuracy requirement is that the returned balance must have been accurate at some time, τ , in the 60 seconds prior to the time when the account balance request is issued. This interval gives the stream processing system some flexibility as to when to update the account balance of a vehicle. If an account balance request is concurrent with some toll charges that is yet to be charged to a vehicle´s account, the system can choose to process the query and therefore produce a result where $\tau < t$ or wait until the toll has been charged to the vehicle.

The validation tool takes the output events generated by the DSMS and compares them with the generated data. It checks the output events to see if they meet the response time and accuracy requirement. It is expected that most systems will produce accurate result but will for some scale factor not be able to meet the response time requirements.

A system achieves an *L-rating* for the benchmark, which is the maximum numbers of expressways, L, for which the system meets the response time and accuracy constraints. When reporting its L-rating, a system should also specify the hardware configuration over which it ran. The number of expressways can be increased by steps of 0.5, thus the first three possible values of L are 0.5, 1.0 and 1.5.

2.3.4 Input Distribution

A 3-hour simulation of a single expressway consists of about 12 million position reports, about 67000 account balances queries, and 14000 daily expenditure queries. As the simulation proceeds the input increases from 14 records per seconds at the start of the simulation to about 1700 records per second at the end. The input data distribution is shown in Figure 3.

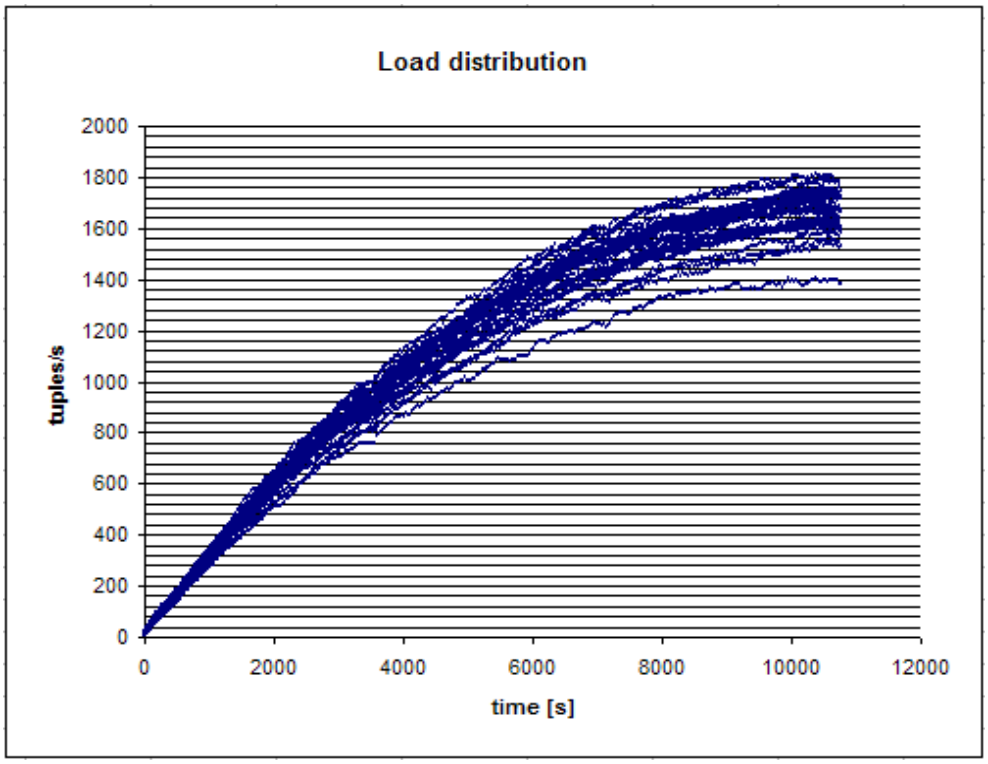

Figure 3: Load distribution for L=1.0

3 Benchmark Implementation

This section describes the SCSQ-LR. It is illustrated in Figure 4.

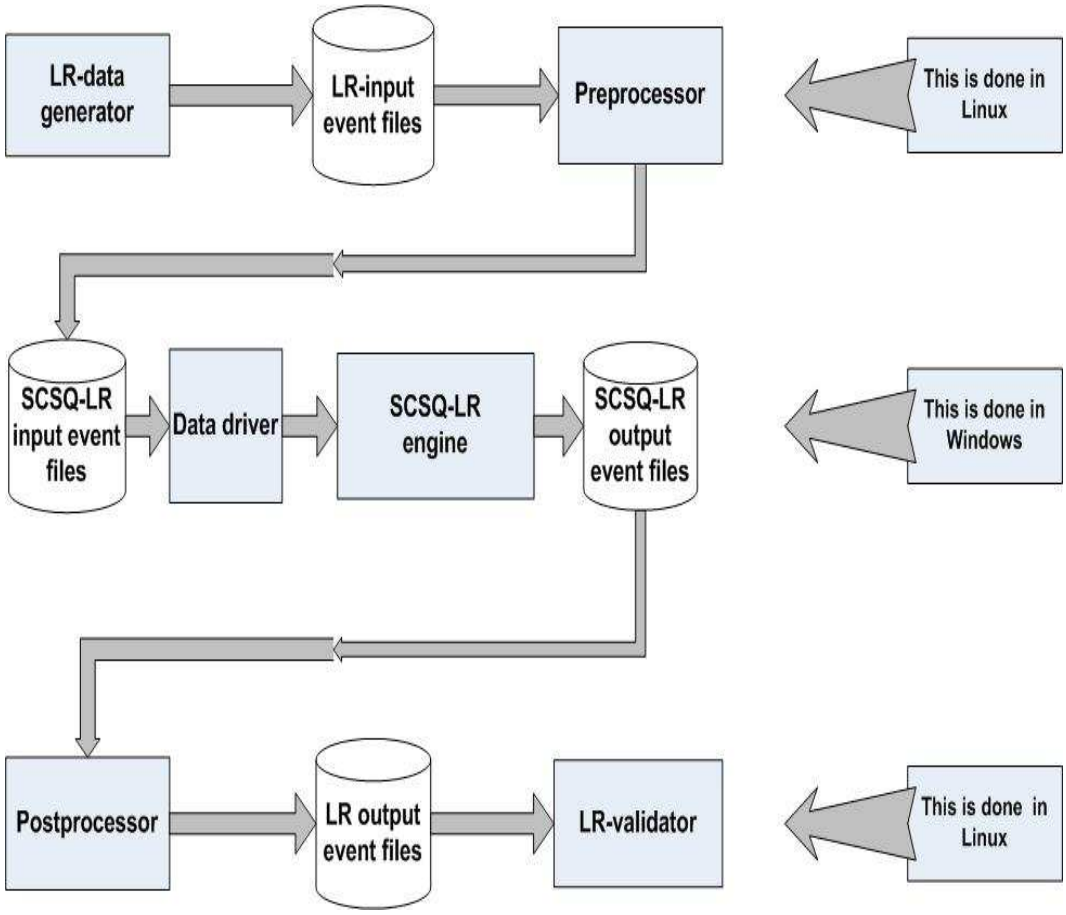

Figure 4: Overview of the SCSQ-LR

- **Data generator**
  The data generator uses MITSIM to create data input events for the benchmark. The data generator is available for download on the Linear Road homepage.

- **Preprocessor**
  The preprocessor is a two line AWK script that takes the files generated by the data generator and puts them on the form required by the SCSQ-LR engine. The script is included in the installation.

- **Data driver**
  The data driver available from the Linear Road homepage[4] gave rise to a number of problems including segmentation faults. It was also only available for the Linux platform. Therefore a new data driver, called *Lread*, had to be developed. It was implemented as a foreign SCSQL-function in Lisp [13]. The new data driver can be used on a Windows XP based systems too. The reader of input events follows the real-time requirements stated by the Linear Road benchmark specification, i.e. there are prescribed time delays to run the benchmark in real time.

  The functionality of Lread is the same as described on the Linear Road homepage. It delivers the SCSQ-LR data to the continues queries according to the timestamp associated with the input event. The code for the Lread is in the file lread.lsp, which is included in the installation.

- **SCSQ-LR engine**
  The benchmark computations are expressed as queries and stored procedures. The time critical functions to maintain statistics are implemented as foreign functions in Lisp (file visited.lsp). Furthermore there are foreign functions to write the output events to files (file lread.lsp). A more detailed description of the internal design of the SCSQ-LR engine queries and updates is presented below. All SCSQ-LR engine scripts are included in the installation.

- **Postprocessor**
  The postprocessor is a small AWK script that takes the output event files from the SCSQ-LR engine and puts them on the required form for the validator. The script is included in the installation.

- **LR-validator**
  The validation tool takes the transformed output events generated by SCSQ-LR engine and compare them with the files from the data generator. The validation tool is available for download on the Linear Road homepage.

SCSQ-LR can be downloaded from:
http://user.it.uu.se/~udbl/download/SCSQ-LR.zip

All files for the benchmark are included in the installation. The system requires a PC with Windows to run. The preprocessed data for L=1.0 and L=1.5 is provided for downloading and therefore the data generator and the preprocessor need not be run. To make a complete validation the validator

has to be installed under Linux, which is rather complicated. However, a quick preliminary test to see if the benchmark has passed the real-time requirements can be made on the PC. A more detailed description is presented in the Appendix.

3.1 SCSQ-LR engine

An overview of the design for the SCSQ-LR engine is illustrated in Figure 5.

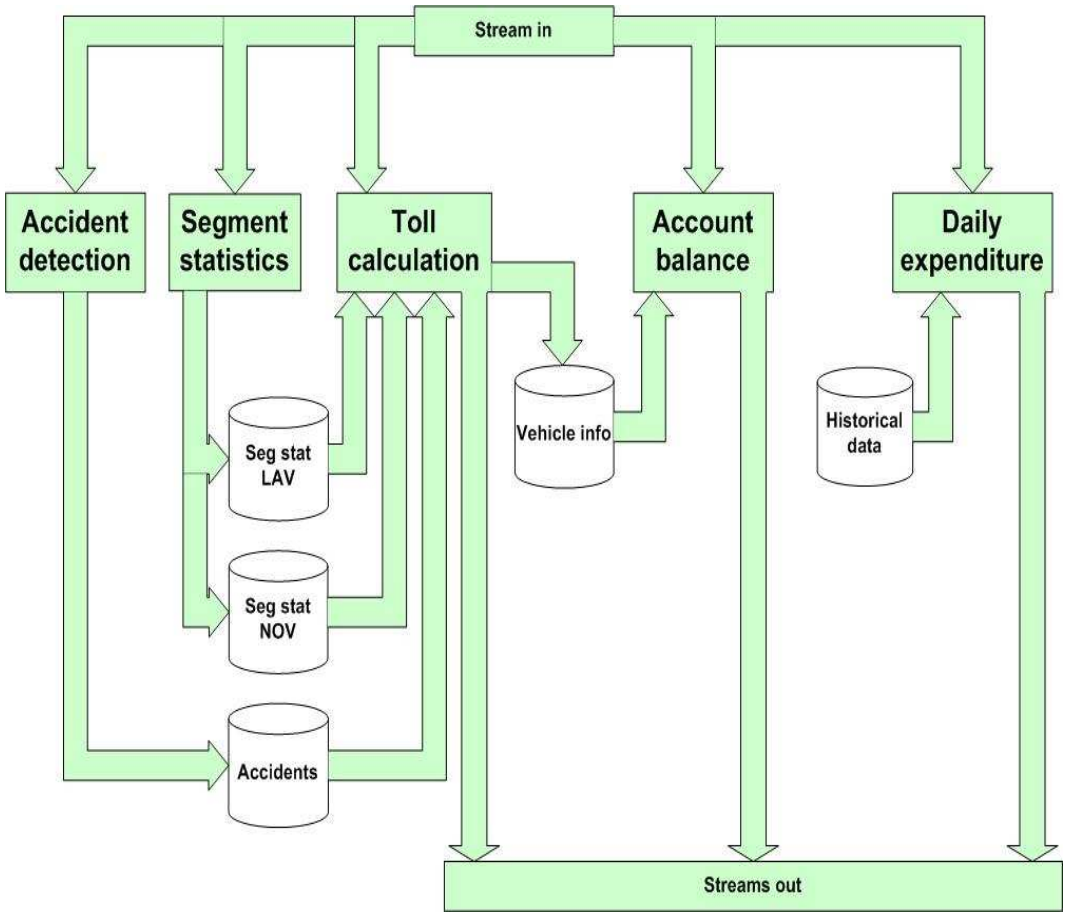

Figure 5: Overview of the SCSQ-LR engine

3.1.1 Accident Detection

An accident is to be detected if two cars report having the same position in four consecutive reports. The accident detection takes every position report

input events that reports having a speed equal to zero and checks if it is reported as a car already detected in an accident, a so called smashed car. If it is not a smashed car, a new stopped car report will be emitted as an output event. The system then checks if there are two cars that have reported four consecutive reports at this position within a time span of 120 seconds. If so, an accident is added to an *accident table* and an *accident alert* is emitted as an output event. The involved cars are added to the smashed cars table. The smashed cars table is used to prevent multiple accident alerts from the same accident. The accident detection is responsible for maintaining the *Stopped_cars, Smashed_cars*, and *Accident* tables. The design is illustrated in Figure 6.

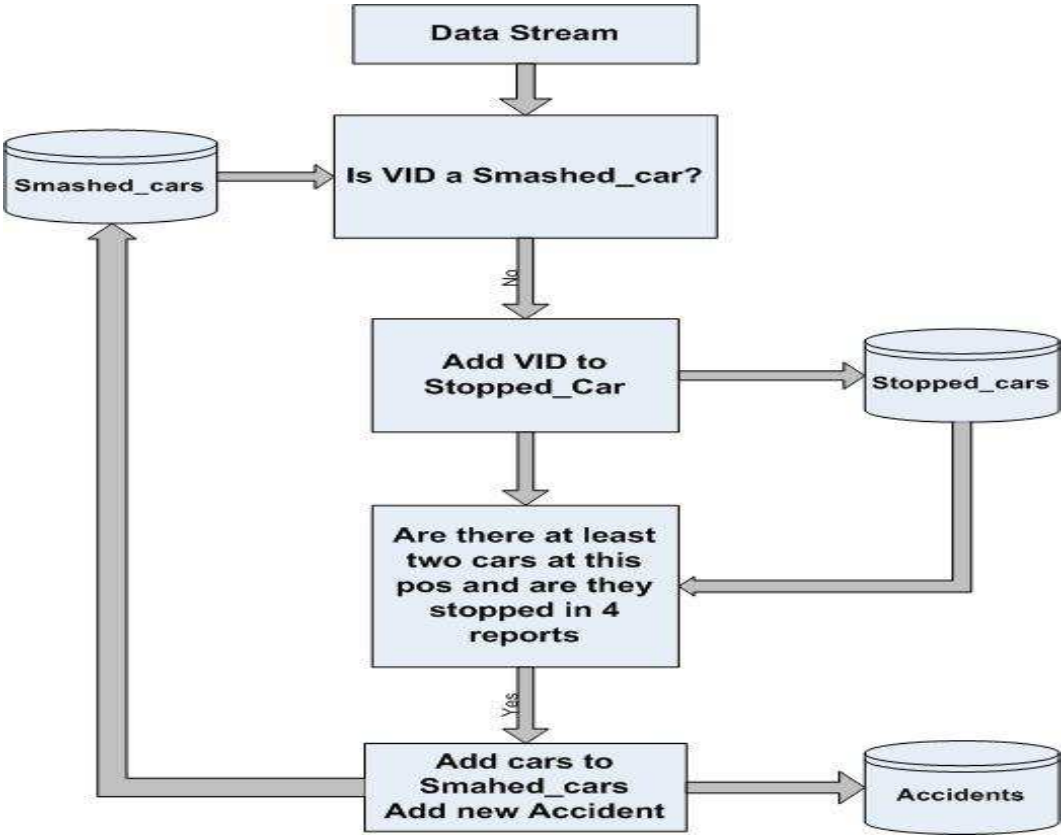

Figure 6: Accident detection

3.1.2 Segment Statistics

The *segment statistics* is responsible for maintaining the statistics for every segment on every expressway with different time span. The *Seg_stat* function

calculates two things, the *Number Of Vehicles* (NOV) present on a segment during the minute prior to the current minute, and the *Latest Average Velocity* (LAV). The LAV is calculated for a given segment with a time span of the five minutes prior to the current minute. The segment statistics was identified to be the bottleneck of the system.

The first approach was to store each position report in a table for six minutes and calculate the segment statistics for a given segment as soon as the statistics were needed for toll calculations. As the number of requests for toll calculations increased it became clear that this was not a usable solution.

The second approach was to store the reports in the same way but calculate and store the statistics for every segment when the system reported a new minute. With 200 segments (100 segments and two directions) this approach proved to be too slow.

The third approach was to fine tune the time for calculations. Instead of calculating all the statistics when the system reported a new minute an attempt to calculate every n:th second was used. Since it is possible that a vehicle emits more than one report in each segment the calculations for the number of vehicles in each segment becomes more complex. Finding unique vehicle IDs cost much time, which meant that the calculations took too much time and the system started to lag.

The final approach was to implement the position report calculations as foreign functions in Lisp called in the stream queries. To be able to answer the queries for NOV and LAV eight B-trees were used. For the NOV two B-trees were used, one for the current minute, which is filled with information from the incoming reports, and one B-tree for the preceding minute. The latter was used when answering the NOV simply by scanning the B-tree and summarizing the number of unique cars present in the segment of interest. When the time changes to a new minute the old B-tree is cleared and becomes the B-tree that will be filled with the reports from the stream.

To solve the LAV six B-trees were used. The solution is similar to the NOV. However, since it is possible that a car emits two reports in a segment during the same minute it becomes a bit more complicated. To comply with the requirements from the benchmark an average velocity has to be calculated for such a vehicle. Both the speed of the car and the number of times the car is reported in the segment are stored. The vehicle's average speed is calculated

by the formula:

$$v_{new} = \frac{v_{avg} * n_{old} + v_{rep}}{n_{old} + 1}$$

v_{avg}=Old average speed of the vehicle.
v_{new}=Average speed of the vehicle.
v_{rep}=Speed of the vehicle reported in the last report.
n_{old}=Number of sightings of the car during last minute.

The result is then stored in the B-trees as:

$$v_{avg} = v_{new}$$

$$n_{old} = n_{old} + 1$$

The LAV is then calculated by the formula:

$$LAV = \frac{1}{5} \sum_{j=currenminute-6}^{currentminute-1} \frac{1}{n} \sum_{k=1}^{n} (v_{avg})_k$$

j=B-tree to search.
k=car in segment
n=number of cars in segment

The design is illustrated in Figure 7.

3.1.3 Toll Calculations

The toll calculation *toll_calc* uses information from the segment statistics and the accident detection. For each position report it checks if the vehicle is entering a new segment. If the car is in a new segment the toll for the previous segment is charged. If the car is not on an exit ramp the toll for the new segment is calculated. The calculation is based on the statistics for the given segment. The LAV and NOV for the given segment is fetched and a check is made to see if the segment is in close proximity of an accident. A segment is considered to be in close proximity of an accident if there is an accident within five segments downstream. An accident alert will be sent if the car is in close proximity of an accident. If the LAV is higher than 40mph or the NOV is less than 50 or the segment is in close proximity of an accident, the toll is set to zero. In other case the toll is calculated as

$$toll = 2 * (NumberOfVehicles - 50)^2$$

The function *toll_calc* is responsible for maintaining the *Vehicle_info* table. The design is illustrated in Figure 8.

3.1.4 Historical Queries

All historical queries are handled as simple lookups into tables maintained by the implemented system. For example, the account balance query is handled by searching the *Vehicle_info* table:

```
select balance
from  integer v,integer xway, integer p, integer s,
      integer d, integer b,real l, integer va, real n
where vehicle_info(v)=<xway,p,s,d,balance,l,va,n> and v=vid;
```

A daily expenditure query is handled by searching through the history table:

```
select daily_exp
from integer v,integer d,integer x,
     integer daily_exp
where history()=<v,d,x,daily_exp>
  and v=vid and d=day and x=xway
```

The answer to a historical query will be an output event.

As in previously published implementations of the Linear Road, Aurora[10], SPC[11] and XQuery[12] the travel time estimation was not implemented. Queries of this type were therefore ignored.

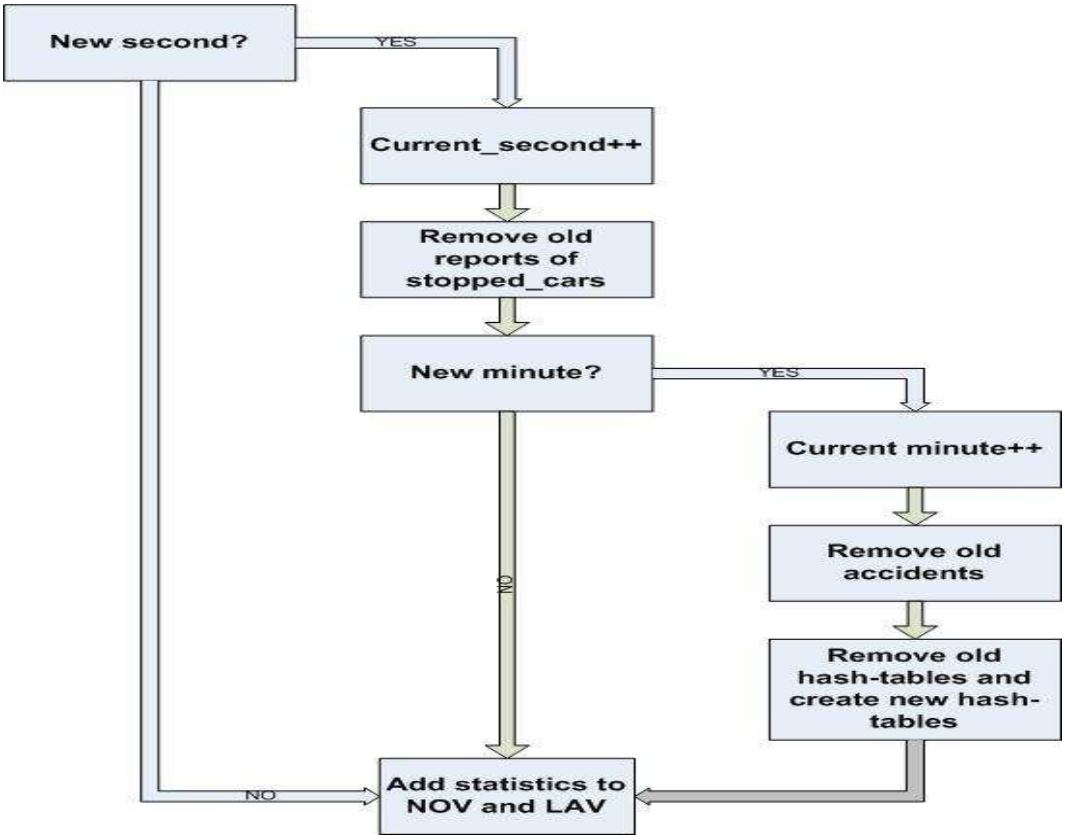

Figure 7: Computing segment statistics

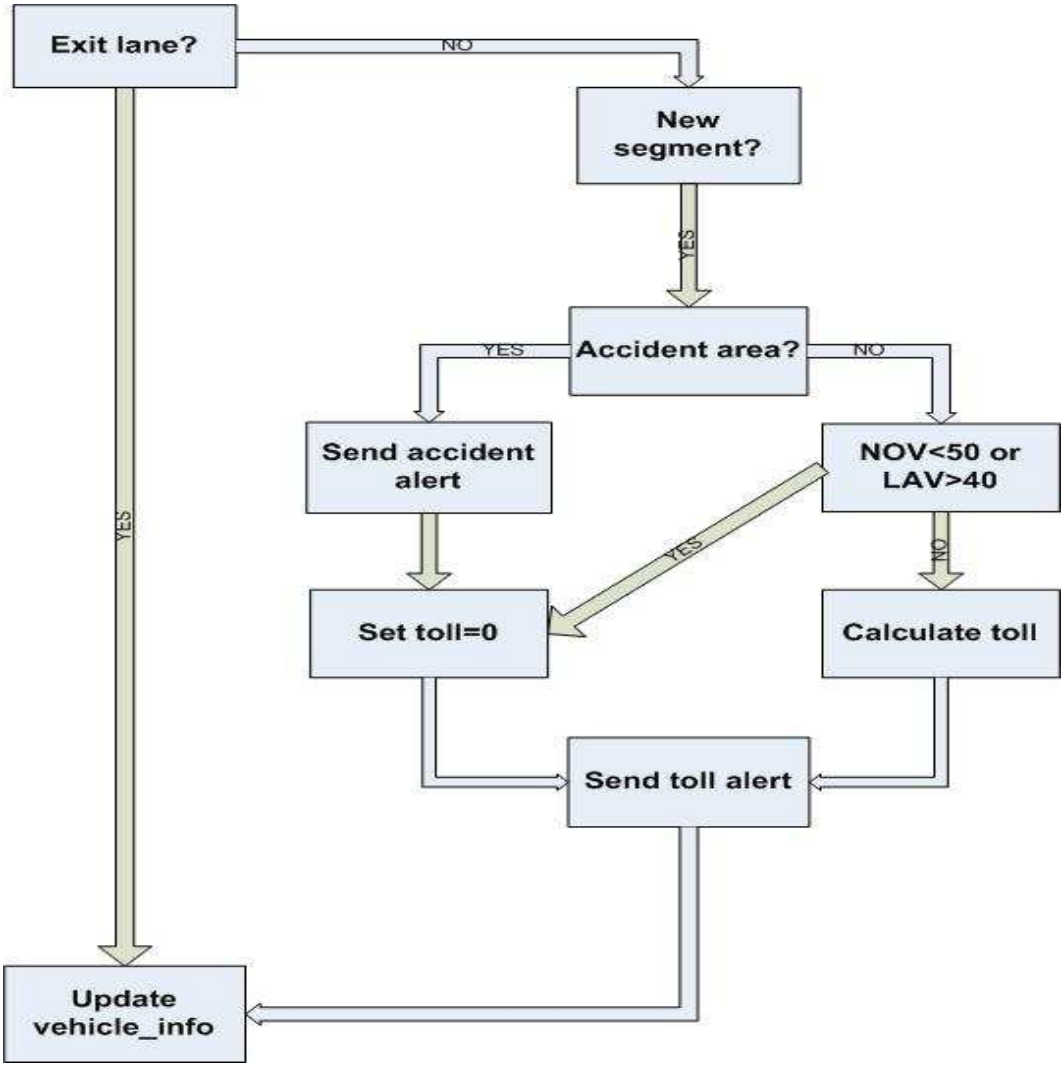

Figure 8: Toll calculation

4 Results

The SCSQ-implementation scored L=1.5 on the benchmark. That is, it was able to run one and a half expressway. The implementation of the benchmark was evaluated on a Windows XP based laptop with a 1,73GHz processor with 1GB of RAM and a 5400rpm harddrive. The results from the different tests are presented below.

4.1 L=1.0

The worst case response time(the time between the input event enters the system and the required output event is sent from the system) was 1.09 seconds, which is well within the required time frame of 5 seconds. The number of reports(output events) with the specific response time for the different queries are shown in Figure 9-12.

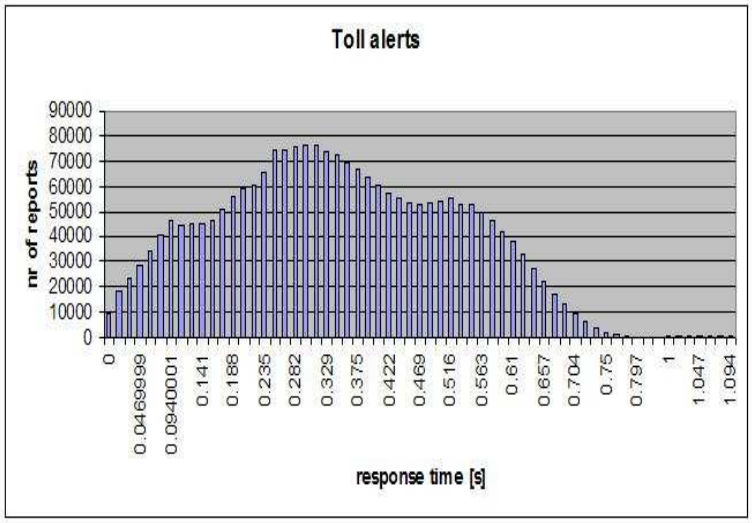

Figure 9: Response time for toll alert. L=1.0

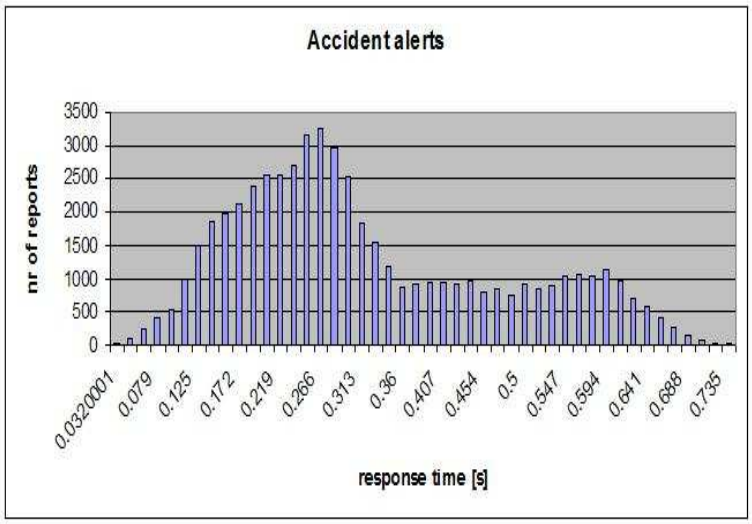

Figure 10: Response time for accident alert. L=1.0

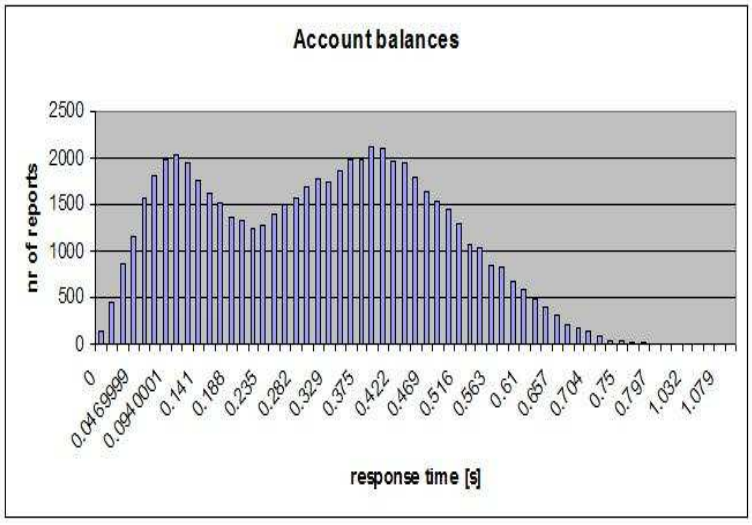

Figure 11: Response time for account balance. L=1.0

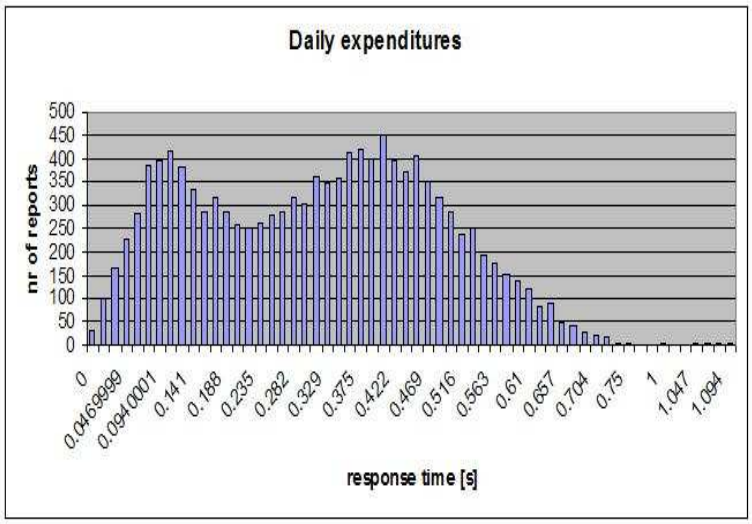

Figure 12: Response time for daily expenditure. L=1.0

4.2 L=1.5

The worst case response time was 1.28 seconds. The number of reports(output events) with the specific response time for the different queries are shown in Figure13-16.

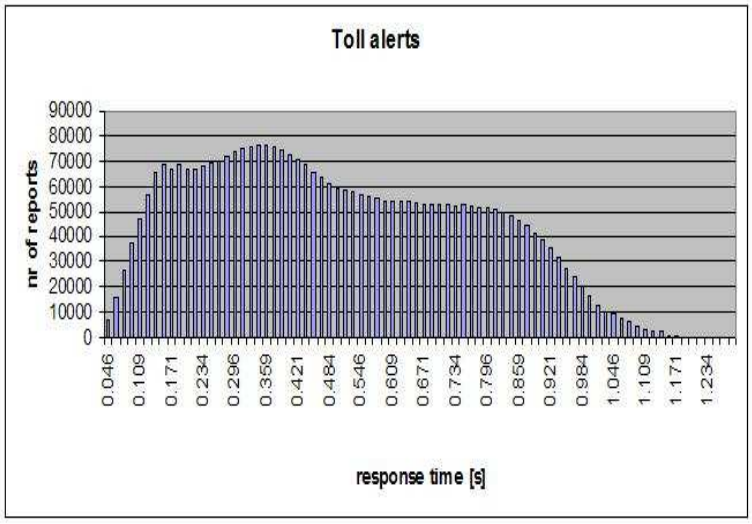

Figure 13: Response time for toll alerts. L=1.5

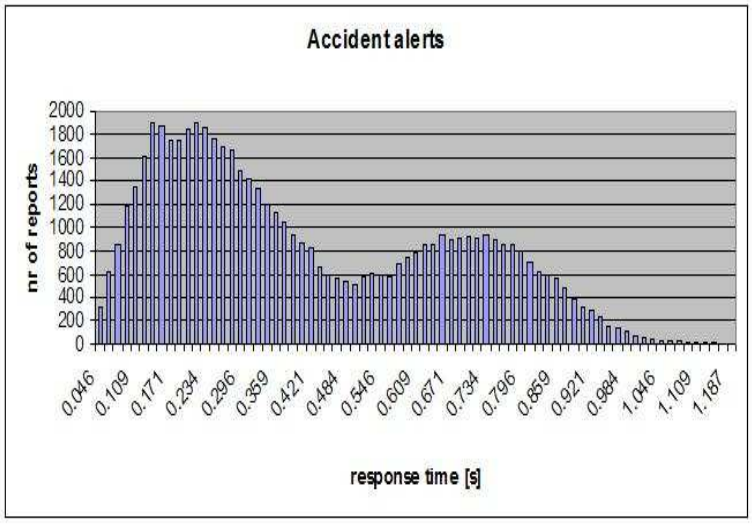

Figure 14: Response time for accident alerts. L=1.5

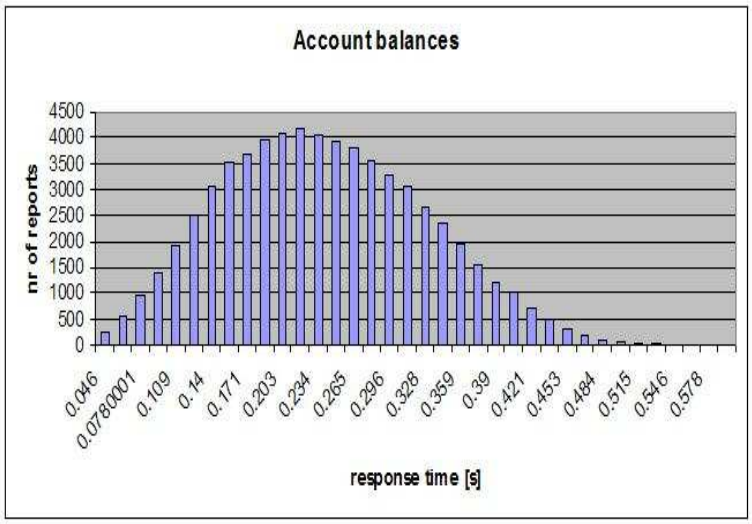

Figure 15: Response time for account balances. L=1.5

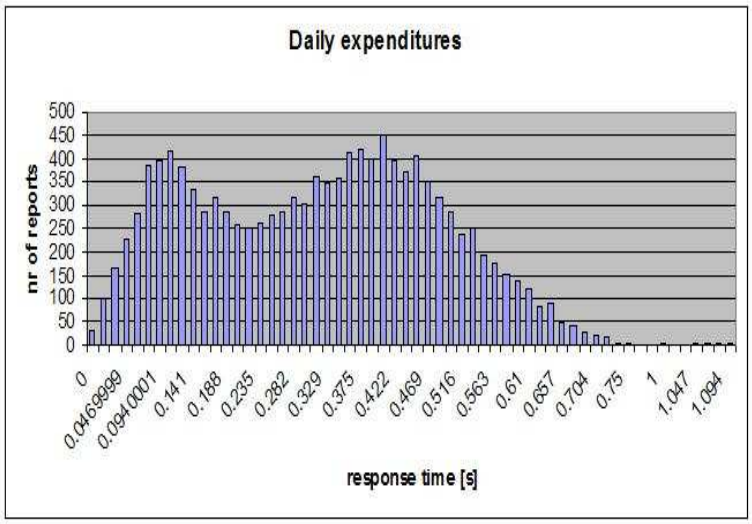

Figure 16: Response time for daily expenditures. L=1.5

5 Related Work

There are so far four published results of implementations of the Linear Road Benchmark, Aurora[10], an unknown relational database system[10], IBM Stream Processing Core[11], and XQuery[12].

In the Aurora report [10] two implementations are presented, one over a commercially available relational database and one over a pre-release commercialization of Aurora. The testbed for both implementations were a 3GHz Pentium Box with 2 GB RAM and running Linux. The implementation on the RDBMS used standard SQL and stored procedures, however no details of this implementation have been published. The L-rating of the RDBMS-implementation was L=0.5. The implementation in Aurora is in line with the approach presented in this Thesis.

In the report [11] an implementation of Linear Road Benchmark in the SPC stream processing system is presented. The testbed was an 85-node Linux cluster each with a dual-core hyper-threaded 3 GHz Xeon processor with 2GB Ram. The approach is in line with the Aurora implementation. They also evaluated the performance when using more than one node.

The highest L-ratings so far published are Aurora and SPC both with L=2.5. However both Aurora and SPC are low level C implementations and do not use a declarative query-language. The SPC furthermore used more than one node to accomplish an L-rating of L=2.5 while the SCSQ-implementation runs on a single laptop PC.

The implementation presented in [12] uses the same approach used in SPC and Aurora. The implementation is fully in XQuery, extended with FORSEQ and continuous queries. Their approach is to use window functions. The testbed was a Linux machine with 2.2 GHz AMD Opteron processor and 4GB of main memory (we use a 1.73 GHz PC with 1 GB of RAM). The L-rating for the XQuery system was L=1.5, the same as SCSQ-LR.

6 Conclusions and future development

In this Thesis the SCSQ-implementation of the Linear Road benchmark was presented. None of the previously published implementations of the Linear Road benchmark [10], [11] and [12] are available for download; therefore it has not been possible to compare the performance on the same testbed as the SCSQ-implementation was run on. However when comparing the results to the other published implementations we see that the SCSQ-implementation performs well even on a single node. The SCSQ-implementation scores the same L-rating as the XQuery-implementation. If we compare the SCSQ-implementation with the SPC-implementation, our response time is much shorter for a single node running L=1.0 even though the SCSQ-implementation ran on a laptop. The worst-case response time for SCSQ was 1.09 seconds. That should be compared to SPC that starts with a response time of 1.26 seconds and has a worst case response time of 2.23 seconds.

A higher L-rating than L=1.5 might be possible to achieve for the single node implementation in SCSQ. However there was no time to to perform the tests with L=2.0 in this Thesis. A testbed with more RAM should be used for L⟩1.5.

One of the most important lessons learned in this Thesis is the importance of regression tests. An insufficient regression test was used when working to improve the performance of the SCSQ-LR. That resulted in a new system that later did not pass the validation. A more rigorous regression test has been developed to be used to further improve the performance of the SCSQ-LR-engine.

7 Acknowledgements

I would like to thank my supervisor Erik Zeitler and Professor Tore Risch for their valuable advises and contributions in completing this project.
This project has been funded by ASTRON.

References

[1] Daniel J, Abadi et al, "Aurora: a new model and architecture for data stream management", VLDB Journal 12(2),Springer August 2003

[2] A. Arasu, B. Babcock, M. Datar, K. Ito, I. Nishizawa,J. Rosenstein, and J. Widom. STREAM: The Stanford Stream Data Manager (DemonstrationDescription). In Proceedings of the 2003 ACM International Conference on Management of Data(SIGMOD 2003), San Diego, CA, June 2003.

[3] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In Proceedings of the 2003 Conference on Innovative Data Systems Research (CIDR 2003), Asilomar, CA, 2003.

[4] http://www.cs.brandeis.edu/%7Elinearroad/index.html

[5] Mitsim , http://mit.edu/its/mitsimlab.html

[6] E.Zeitler, T.Risch. Processing high-volume stream queries on a supercomputer. ICDE Ph.D. Workshop 2006, Atlanta, GA, April 2006,

[7] E.Zeitler and T.Risch: *Using stream queries to measure communication performance of a parallel computing environment.* http://user.it.uu.se/%7Eudbl/publ/DEPSA2007.pdf Proc. /First International Workshop on Distributed Event Processing, Systems and Applications (DEPSA)/ http://www.cs.uga.edu/%7Elaks/depsa/, Toronto, Canada, June 29, 2007.

[8] G.Gidofalvi, T.B. Pedersen, T.Risch, and E.Zeitler: *Highly Scalable Trip Grouping for Large Scale Collective Transportation Systems*, To be presented at /11th International Conference on Extending Database Technology, EDBT 2008 / http://edbt2008.univ-nantes.fr/, Nantes, France, March 2008.

[9] T.Risch, V.Josifovski, and T.Katchaounov: Functional Data Integration in a Distributed Mediator System, in P.Gray, L.Kerschberg, P.King, and A.Poulovassilis (eds.): Functional Approach to Data Management - Modeling, Analyzing and Integrating Heterogeneous Data, Springer, ISBN 3-540-00375-4, 2003.

[10] A.Arasu, M.Cherniack, E.Galvez, D.Maier, A.S.Maskey, E.Ryvkina, M.Stonebreaker and R.Tibbetts. Linear Road: A Stream Data Management Benchmark. In Proceedings of the 30th International Conference on Very Large Data Bases Conference(VLDB 2004), Toronto, Canada, 2004.

[11] N.Jain, L.Amini, H.Andrade, R.King, Y.Park, P.Selo, C.Venjatramani. Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In SIGMOD, Chicago, Illinois,USA, 2006

[12] I.Botan, P.M.Fischer, D.Florescu, D.Kossmann, T.Kraska, R.Tamosevicius. Extending XQuery with Window Functions, In VLDB 07, Vienna, Austria, 2007

[13] T.Risch: ALisp User's Guide, Uppsala University, 2000. http://user.it.uu.se/%7Etorer/publ/alisp.pdf

APPENDIX

A How to install and run SCSQ-LR

SCSQ-LR can be downloaded as a zip file from
`http://user.it.uu.se/~udbl/download/SCSQ-LR.zip`
The following installation instructions are in the readme file:

```
Instructions for running Linear Road under SCSQ
-----------------------------------------------

To run Linear Road under SCSQ a Windows based system is needed. The
testbed we used was a Windows-XP based laptop with a 1.73GHz Intel
dualcore processor with 1Gb of RAM.  It was able to successfully run
Linear Road for L=1.5.

A. Install SCSQ and Linear Road files
-------------------------------------

A1. Extract the files to directory, the 'LR home directory'.

A2. Install the system by in the LR home directory running the scripts:

       mkhist.cmd
    followed by
       mkdmp.cmd

A3. Test the installation by running the script

      test.cmd

    It should exit successfully after having run a small Linear Road
    event sequence.

At this point you have succesfully installed LR. The system can be run
interactively by the script

      lr.cmd

It enters a query interaction loop where you can specify SCSQL
```

queries. The language is an extension of AmosQL, documented in
http://user.it.uu.se/~udbl/amos/doc/amos_users_guide.html

B. Download data used by Linear Road benchmark
---

1. Ready made Linear Road datafiles for L=1.0 and L=1.5 can be found here:
http://udbl2.it.uu.se/LR/

2. Put all data files in <lr>/data where <lr> is the Linear Road home
directory.

3. If you want to create more data files, the Linear Road data
generator can be used on a Linux system to produce the input files:

http://www.cs.brandeis.edu/%7Elinearroad/tools.html

To get the generated files on the form requiered by the SCSQ-LR engine
use the script prprocess.sh

C. Run the Linear Road benchmark on your PC

C1. Load the historical data into the database by running one of the
scripts

        mkhist10.cmd for L=1.0 or
mkhist15.cmd for L=1.5

C2. Save the full database image in file lr.dmp by running

        mkdmp.cmd

C3. Start SCSQL query loop with the command
        lr.cmd

C4. Run Linear Road by calling the SCSQL function runLR.

    To run Linear Road for L=1.0 execute the SCSQL function call:

        runLR('data/cardatapoints10.out');

```
    To run Linear Road for L=1.5 execute the SCSQL function call:

        runLR('data/cardatapoints15.out');

    runLr is a foreign function in SCSQ-LR that runs the Linear Road
    benchmark according to the benchmark specification. This includes
    required time delays between arriving events. The very compact
    source code of runLR is in file src/historytest.osql and the data
    provider is in src/lread.lsp.

    After the run execute the SCSQL function call:

    responsetime();

    The worst case response time for the different queries will then
    be calculated.

    After the run exit SCSQ with the command:
        quit;

C5. All output events are stored in the files:
        o-acc-alert     :accident alerts
        o-t2 :accountbalances
        o-t3 :daily expenditure
        o-tollalert :toll alerts

These files log the output events produced by SCSQ while running
Linear Road. The outputs are validated by running the Linear Road
validation system, which needs a Linux installation with PostGres
installed.

The full validation process is described next.

D. Running the Linear Road validator
------------------------------------

To run the validator a Linux-based computer is needed. See
http://www.cs.brandeis.edu/%7Elinearroad/sysreqs.html for system
requirements.
```

D1. Download and install the validator from the Linear Road homepage.
http://www.cs.brandeis.edu/%7Elinearroad/tools.html

D2. Use the provided linux-script preverify.sh to transform the output files from SCSQ to the format required by the validator.

D3. In the validator replace the Pearl file extractLavs.pl with the extractLavs.pl from the LR zip-file.

D4. Follow the instructions on Linear Road homepage for the validator.