IT 10 006
Examensarbete 30 hp
Mars 2010

The performance of a relational database system for a data stream management system benchmark

Fredrik Edemar

Institutionen för informationsteknologi
Department of Information Technology

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

The performance of a relational database system for a data stream management system benchmark

Fredrik Edemar

In recent years the interest in data stream management systems (DSMS) has grown. They are characterized by the processing of continuous, high-volume, and possibly time-varying data streams. Linear Road Benchmark (LRB) is an example of a data stream benchmark application that handles variable toll charges in highways in a fictional traffic system.

In this Thesis the performance of LRB implemented using the relational database MySQL is investigated. The implementation is called SCSQ-MySQL and is executed from the DSMS SCSQ. In LRB the L-rating stands for how many highways a benchmarked DSMS can handle and is used to compare the performance of DSMSs. L=0.5 (one highway in one direction) was achieved with SCSQ-MySQL executed on a single node.

Handledare: Erik Zeitler
Ämnesgranskare: Tore Risch
Examinator: Anders Jansson
IT 10 006
Tryckt av: Reprocentralen ITC

Sammanfattning

De senaste åren har intresset för dataströmhanterare (DSMS) ökat. Till skillnad från konventionella databashanterare (DBMS) som bara tillåter frågor över statiska data, så tillåter DSMS frågor över data som förändras över tiden, så kallade dataströmmar. Svaret på sådana kontinuerliga förändras i sin tur, och är därför också att betrakta som dataströmmar. Kännetecknande för DSMS är att de hanterar stora mängder kontinuerliga och tidsvarierande dataströmmar.

Linear Road Benchmark (LRB) är en provbänk som används för att undersöka skalbarheten och pålitligheten hos DSMS. En DSMS som implementerar LRB ska hanterar variabla motorvägstullar i ett fiktivt trafiksystem och besvara kontinuerliga och historiska frågor inom specificerad svarstid. Antalet parallella motorvägar som en LRB-implementation klarar kallas implementationens L-värden (L-rating).

Den här rapporten introducerar SCSQ-MySQL, en implementation av LRB i en konventionell databashanterare, MySQL. Implementationen i MySQL anropas från dataströmhanteraren SCSQ. L=0.5 (en motorväg i en riktning) uppnåddes när SCSQ-MySQL kördes på en persondator.

Contents

1 Introduction

In recent years the interest in data stream management systems has grown. Processing large volumes of continuous data, in areas such as online stock market analysis and sensors dealing with digital audio and images, involves multiple continuous, high-volume, and possibly time-varying data streams. For such applications, it is not feasible to store all the data on disk. The solution: a Data Stream Management System (DSMS), which is a database system for primarily handling data streams. STREAM [3], SCSQ [12], TelegraphCQ [5] and Aurora [1] are examples of such systems.
The purpose of this Thesis is to investigate the performance of a DSMS application, implemented using a traditional relational database management system (DBMS), MySQL. This approach of implementing data stream applications in a relational database has the advantage that a DBMS is a far more utilized database type. However, a standard DBMS may not be able to keep up when the data stream increases, and will therefore deliver results of queries too late.

The well-known Linear Road Benchmark (LRB) [2] is used for our experiments. LRB simulates a traffic system of highways. LRB has a dynamic toll based on the current traffic and accident situation. Input tuples are continuously sent to an LRB implementation, and the result must be delivered in a certain time frame, and of course be correct. During the simulation for three hours, the traffic load is continuously increased. To compare the performance between different LRB implementations, the *L-rating* shows how many highways the system is able to process.
LRB has been implemented in SCSQ previously, both on a single node[11] and parallelized [13]. The implementation in this Thesis is based on a regular relational database, MySQL, to process data, while the previous implementations stored working data in SCSQ's main memory database. The presented implementation of LRB is named SCSQ-MySQL.

Since the performance of a disk-based DBMS such as MySQL is significantly lower than a main-memory DBMS it is more challenging to achieve high L-ratings. In SCSQ-MySQL the future approach to achieve higher L-ratings is to parallelize the stream processing by starting several MySQL servers in parallel which each process a portion of the stream. If the implementation can achieve an L-rating of 0.5 it means that it should be possible to scale LRB by running several SCSQ-MySQL in parallel, one per direction.

In summary, the following results are presented:

- A conventional DBMS (MySQL) is used to implement a DSMS benchmark.
- The overhead of using the SCSQ JDBC interface was investigated experimentally.
- The implementation is investigated for performance and validated

experimentally.

This Thesis is organized as follows: Section 2 gives an overview of related technologies used, including DSMSs, the Linear Road Benchmark (LRB), and the tools used for this implementation of LRB. In Section 3, the architecture and a more detailed implementation description is presented, followed by the results of performance experiments in Section 4. Section 5 discusses related work, comparing the L-rating of this implementation with others and the conclusions of the Thesis are discussed in Section 6. Appendix A gives installation instructions for SCSQ-MySQL in Linux. Appendices B-E list the source code of the implementation in MySQL (A-D) and SCSQ (E).

2 Background

The key techniques that have been used in the Thesis are in general the DSMS SCSQ, the relational database management system MySQL and the JDBC relational database API. The section ends with an explanation of LRB.

2.1 DSMS

A Data Stream Management System (DSMS) is similar to a database management system (DBMS) like Microsoft SQL Server and MySQL, but with two differences:

1. A conventional DBMS stores finite data sets persistently, usually on disk, while a DSMS processes data continuously from e.g. sensors.
2. Queries to a relational database are passive in the sense that they are sent from applications to the DBMS, which delivers the result to each processed query. By contrast, a DSMS supports *continuous* queries over streams that, once they are activated, continuously deliver result streams until they are deactivated.

Usually DSMSs can combine stored and streaming data by handling both conventional *ad hoc* queries and continuous queries.
An input stream to a DSMS consists of continuously delivered data input *tuples*, e.g from a sensor. We also call a tuple in a stream an *event*. The data rate and size of the tuples may vary. In SCSQ-MySQL input tuples are processed as continuous queries, but all working data is stored in a conventional DBMS. The tuples in the output streams are calculated from the input tuples as soon as they are available. An input tuple in one input stream can lead to zero, one or several output tuples in one or more output streams depending on the continuous query.

It is important that the queries executing in a DSMS respond to input tuples in time. If not, the tuples are queued and the response times will accumulate. Alternatively tuples may be dropped when the DSMS cannot keep up with the input rate, so called *shedding* [1]. Depending of the application, an inaccurate or missing value, or too long response time could cause severe damage, for example losing a lot of money in the financial area.
In DSMSs, due to high load performance requirements, the data is usually stored in main memory. By contrast, SCSQ-MySQL stores all working data in a regular DBMS.

2.2 SCSQ

Super Computer Stream Query processor (SCSQ) [12] [13] is a DSMS developed at Uppsala University. It is based on the functional and object-oriented DBMS Amos II [10]. SCSQ handles complex queries over data streams of high volume by parallelizing the query execution. Queries involving filter, transform and join functions can be applied on the streams. The query language SCSQL features customizable parallelization functions, which enable the user to specify how the parallelization should be executed.

There are three primitive functions for parallelizing stream queries in SCSQ, as illustrated in Figure 1:

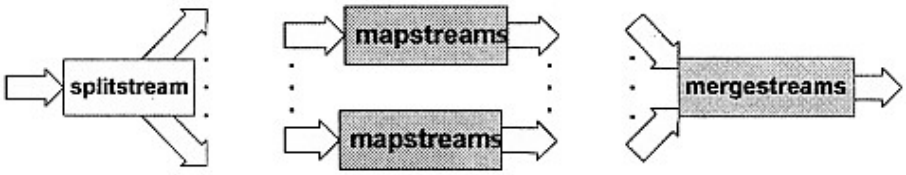

Figure 1: SCSQ functions splitstrem, mapstreams and mergestreams

- *Splitstream*. Splits one stream into two or more output streams.
- *Mapstreams*. Applies a stream function on each stream in a collection of streams.
- *Mergestreams.* Merges a collection of streams into a single output stream.

Splitstream, *mapstreams* and *mergestreams* can be combined in any way, which makes the parallelization very flexible. The system supports parallel computations in a heterogeneous distributed environment since SCSQ runs on many software platforms, such as Windows, Linux, and IBM BlueGene.

SCSQ has facilities to add time stamps to tuples when they arrive and are emitted. Time stamps are used when the output result is dependent on the input arrival time. In the present work, time stamps are used to measure response time.

The present implementation of SCSQ-MySQL runs only on a single SCSQ node, which sends the incoming tuples to MySQL, fetches the result tuples from MySQL, and finally delivers a result tuple stream. Thus there is no parallelization.

2.3 MySQL

MySQL is a widely used DBMS. MySQL has a numerous advanced DBMS features, including stored procedures, triggers, distributed storage and transactions. Stored procedures are important for processing data streams efficiently as was concluded in [2]. Stored procedures are used in SCSQ-MySQL.

2.4 JDBC

Java DataBase Connectivity (JDBC) is a Java API for communicating with relational DBMSs. The DBMS specific part is handled by *JDBC drivers*, one for each system. An SQL query is sent to the driver, which then accesses the selected DBMS and returns the result through the interface. Figure 2 explains the JDBC structure.

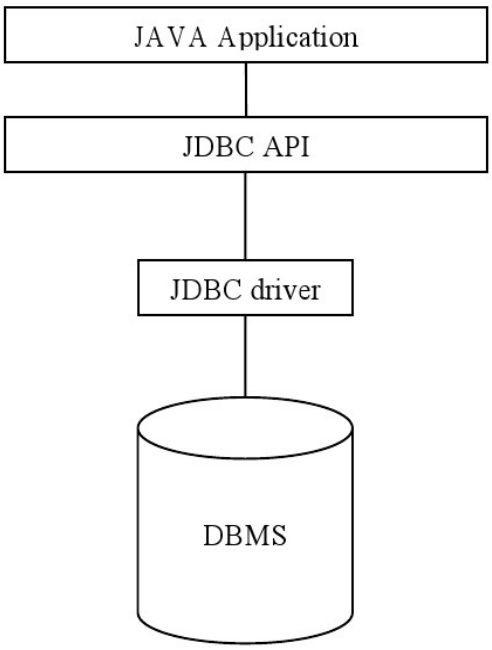

Figure 2: Structure of JDBC

SCSQ has a JDBC wrapper [9], acting as the Java Application in Figure 2. It is used in the present work for connecting to a MySQL database trough JDBC.

2.5 Linear Road

LRB simulates a highway traffic toll system in the made-up city Linear City. In this town, there are L straight parallel highways, and each of them are 100 miles long. Every highway is divided into 100 segments (so that one segment is one mile). Like on a real highway, cars drive in both directions, and one lane in each direction is reserved for entering and leaving the road. For normal travelling there are three lanes. See Figure 3 for a visual explanation.

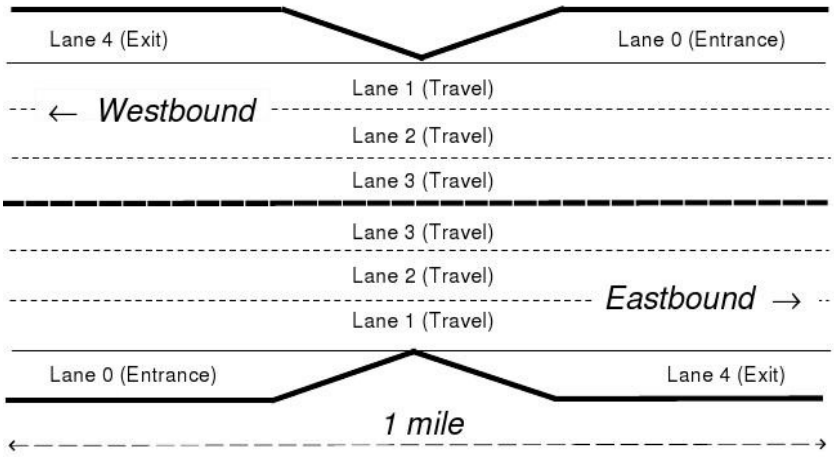

Figure 3: Geometry of a highway segment

For each car on the road in each segment, the toll charge is calculated based on the number of cars in the current segment of the car, the average speed of all cars in the current segment and if an accident has occurred in the vicinity.

The input data is generated by MITSIM, MIT's Microscopic Traffic Simulator MIT [8], which can be configured for different number of highways and cars in the system. MITSIM generates the input data for the simulation as two files, one containing the streaming data about vehicles travelling on the highways, and one with historical data about vehicles. LRB reads these two files for each of the L simulated highways in order to produce the input stream to the benchmarked DSMS. Every vehicle travels some time on the highway during the simulation. Each car emits a position report every 30 seconds. Since MITSIM ensures that the maximum speed for the vehicles is 100 mph and a highway consists of 100 segments, the cars are guaranteed to send out a position report from every segment it travels on. For entrance and exit ramps the speed limit is 40 mph to ensure at least one position report. The journey for a car begins on an entrance ramp and finishes on an exit ramp, except from the uncompleted trips at the end of the simulation period. The source location for the journeys are uniformly distributed over the entrance ramps. Note that MITSIM determines where the vehicles are going and their velocity, LRB only calculates their toll.

At random locations for every 20 minutes, the simulator generates an accident. An accident is defined as two stopped cars at the same position at the same time. The meaning of "stopped" is that a car has emitted its four latest position reports from the same position. When any of the stopped cars involved in the accident signals that it has started to move, the accident is cleared. That takes anything from 10 to 20 minutes. Position reports are still emitted from the stopped vehicles.

There is 1% probability that a position report is followed by a historical query. Half of these are account balance queries, 2/5 are travel time

prediction queries and 1/10 are daily expenditure queries.

The number of cars travelling on the highways (i.e. the flow of tuples) is continuously increased during the simulation to test how well the DSMS handles increasing load.

2.5.1 Events

LRB has four types of input events and five types of output events. The DSMS represents the events as tuples in the input and output streams. Timestamps are attached to all events, both incoming and outgoing, in order to measure the response time of the implementation.

Position report

A position report event triggers a toll notification output event (*toll alert*), containing the toll and average speed for the current road segment. The velocity is calculated by averaging the speed of all vehicles over the latest five minutes. The toll is calculated from the Number Of Vehicles (NOV) in the segment. Let us name the latest average velocity LAV, then the toll formula is $2*(\mathrm{NOV}-50)^2$. Toll is set to 0 if LAV≥40 mph, NOV≤50 or an accident has been detected within five segments downstream. Toll is also zero for vehicles on the outermost lane, used as entrance and exit ramps. The purpose of the dynamic toll calculation is that a high traffic congestion should lead to a high toll and therefore discourage more cars to the highway. When a car enters a new segment and emits a position report, the toll from the previous segment is drawn from the vehicle's toll account.

Accident detection is done when a position report event has arrived. If a vehicle is in a vicinity of five segments upstream accident, an accident notification is emitted, containing the segment where the accident occurred.

Account balance query

Account balance request queries contain a vehicle id and after arrival to LRB a result query with the sum of all assessed tolls for the selected car is emitted. If no toll has been charged, the sum is 0.

Daily Expenditure query

This query is similar to account balance requests, but instead of requiring the current total toll sum, it asks for the toll sum for a vehicle on a specific day on a specific highway. The day must be in the last ten weeks and must not include the present day or the previous day if it ended within five minutes ago. Like for account balance queries, an output query is sent when the result toll has been fetched.

Travel time estimation query

The last input query is the travel time estimation. It estimates the time to drive between two segments on a highway based on the statistics from the

previous ten weeks. Since all known implementations of LRB ignores this query due to complexity, it is henceforth ignored in this report.

2.5.2 Time and accuracy requirements

There are real time requirements for the queries in LRB. A response must be emitted in a certain time frame counted from the time its event or query arrived on the input stream. This allowed time frame is called the response time. Table 1 lists the response times for the different kinds of events.

Table 1: Maximum response times

Input queries and events	**Output queries and events**	**Max response time (sec)**
Position report	Toll alert	5
Position report	Accident notification	5
Account balance rq	Account balance	5
Daily expenditure rq	Daily expenditure	10
Time travel est rq	Time travel estimation	30

Account balance queries also have an accuracy requirement, namely 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 was issued. This means that up to three balance values are correct. The system is allowed to process an account balance query before a concurrent position report or wait until the toll has been updated.

2.5.3 Validation and L-rating

A validation tool is provided for LRB. It ensures that the output events of the implementation are correct and that they meet the time and accuracy requirements. To do this, the validator compares result of the LRB implementation with the input data.

To test how well the implementation of the DSMS perform, the L-rating is used. It stands for how many highways the system can run and still fulfil the allowed maximum response time (MRT). *L* is set to a multiple of 0.5 and a *L=0.5* means a highway in one direction, so an LRB implementation with *L=1.5* is capable of handling two highways, of which one has only one direction. The validator is apart from validating also used to check if the simulator achieves a certain L-rating. The higher *L*-rating, the better performance in comparison to other LRB implementations.

3 SCSQ-MySQL architecture

SCSQ-MySQL is controlled by SCSQ which reads an input stream and sends the input events to MySQL, where the actual simulation is done as shown in the next section. The result streams are computed by stored procedures in MySQL. The output events are temporarily stored as tables in MySQL which are explicitly polled by SCSQ. The output of MySQL is requested from SCSQ each second and is then returned as an output stream. The overall design of the MySQL and SCSQ interaction is shown in Figure 4.

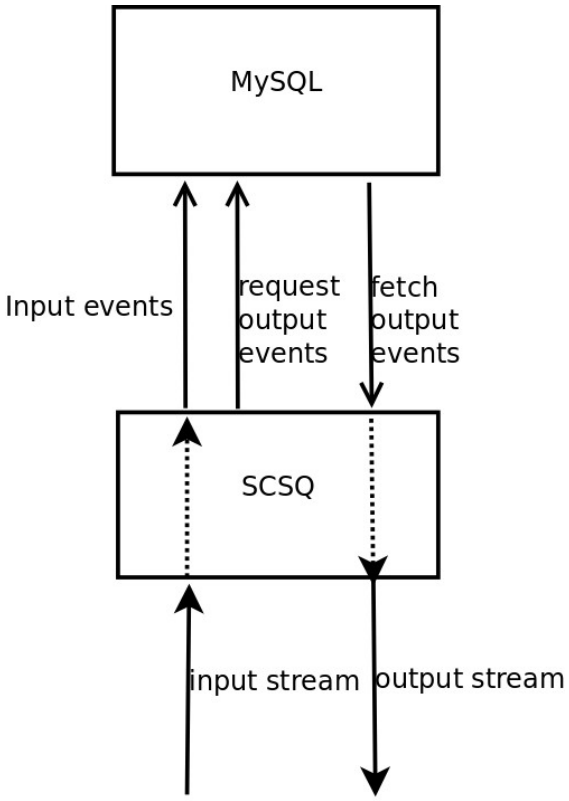

Figure 4: Overview SCSQ-MySQL

The streams in LRB are stored on the hard disk. The input stream is a file generated by MITSIM which is read by SCSQ. The output stream is stored in one file.

4 Implementation of LRB

SCSQ-MySQL is mainly implemented in MySQL. SCSQ sends the tuples of the input stream to MySQL and stores the result tuples from MySQL in an output file.

SCSQ-MySQL was implemented using test-driven development. Each MySQL stored procedure has at least one test case with minimal test data files. The test cases are written in SCSQ.

4.1 MySQL implementation

The overall design is similar to SCSQ-LR [11], but with the main difference that all tables in SCSQ-MySQL are persistent. Figure 5 illustrates the design.

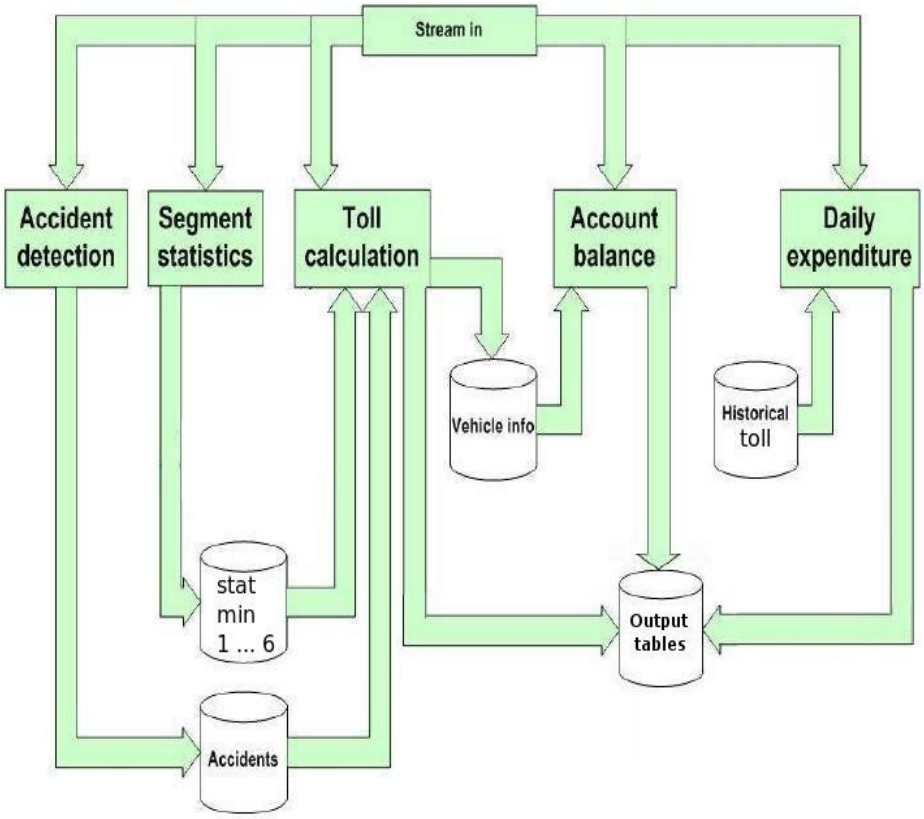

Figure 5: Overview of the MySQL implementation

4.1.1 Tables

The ER diagram of the MySQL implementation is shown in Figure 6. The entities in Figure 5 is correspond to the entities in the ER diagram.

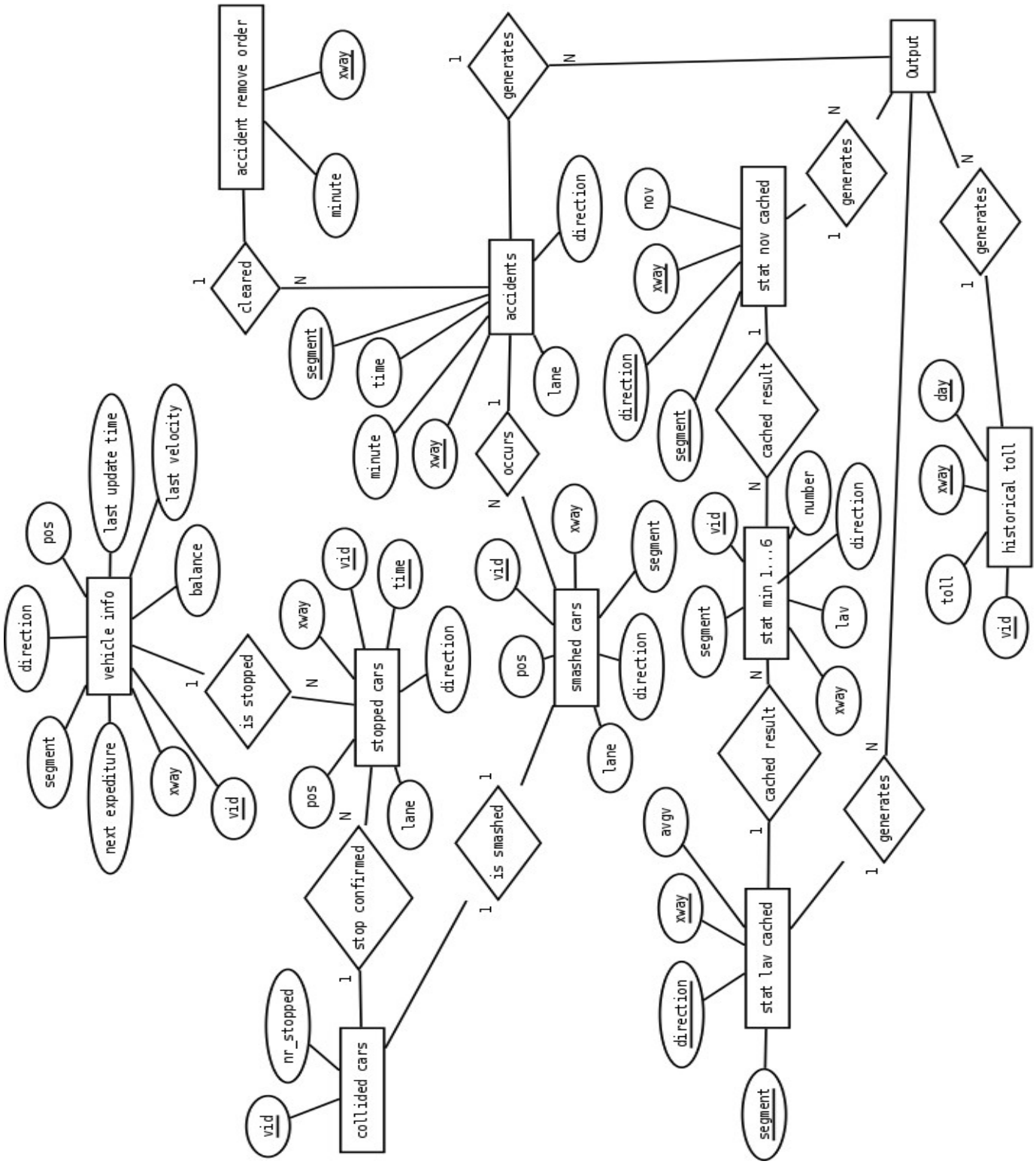

Figure 6: ER diagram of the MySQL implementation

The relationships between the entities are not represented by any tables. Each entity is represented by a table:

Vehicle_info

Primary key: *vid*
Contains information for the present and past vehicles in the traffic system. Their latest reported highway, segment, direction, position, velocity and update time are stored together with the current balance (sum of toll charges) and the toll to be charged when entering the next segment.

Stopped_cars

Primary key: *time, vid*
Index: *pos, xway, direction, lane*
Contains the present cars with velocity 0 and stores where (pos*, xway, direction, lane*) and when (*time*) the vehicle was stopped.

Collided_cars

Primary key: *vid*
Contains the present stopped cars with at least four consecutive reports and also stores the number of such reports, *nr_stopped*.

Smashed_cars

Primary key: *vid*
Contains the present smashed cars and stores where the accident has taken place (*pos, xway, segment, direction, lane*).

Accidents

Primary key: *xway, segment*
Contains the present accidents and stores the position (*xway, segment, direction*) and the minute(*min)* and second(*time)* when the accident occurred.

Accident_remove_order

Primary key: *xway*
Contains which accidents should be removed in highway *xway* after minute *min*.

Stat_min1…6

Primary key: *xway, direction, segment, vid*
Contains the vehicle statistics in one segment for this and the last 1...5 minutes. x*way, direction* and *segment* represent the segment position, *number* is the number of times the vehicle has been reported and *lav* is the vehicle's average speed in that segment.

Stat_lav_cached

Primary key: *xway, direction, segment*
Contains the cached results of LAV statistics for the segment position (*xway, direction, segment).* The LAV value is stored in *avgv*.

Stat_nov_cached

Primary key: *xway, direction, segment*
Contains the cached results of NOV statistics for the segment position (*xway, direction, segment).* The NOV value is stored in *nov*.

Historical_toll

Primary key: *vid, day, xway*
Contains the historical tolls for daily expenditure queries for a vehicle in a day on a highway (*vid, day, xway)* with the toll *toll*.

Output tables

The output events are stored in fours tables, one for each output event type:

output_accident_alert
Primary key: none
Contains the entering time of the input query, the emitting time of the result query, the accident's vehicle ID and accident segment (*time, emit_time, vid, segment).*

output_account_balance
Primary key: none
Contains the entering time of the input query, the emitting time of the result query, the query ID, the vehicle's last updated time and balance of the vehicle (*time, emit_time, quid, lasttime, balance*).

output_daily_exp
Primary key: *time, quid*
Contains the input query's entering time, the result query's emitting time, query ID and the expenditure (*time, emit_time, quid, expenditure).*

output_toll_alert
Primary key: *time, vid*
Contains the entering time of the input query, the emitting time of the result query, the vehicle's average velocity, toll for the next segment and ID (*time, emit_time, vavg, toll, vid).*

Furthermore, the table *sim_time* contains the current simulation minute and second. It consists of two fields, *sim_minute* and *sim_second*, and has one row.

The complete source code of the implementation is listed in Appendix B-D.

4.1.2 Overall dataflow

The stored SQL procedure *lr0()* is the entry point of the MySQL implementation. *lr0()* is called from SCSQ for each incoming event and takes the 11 arguments, one for each element in the input tuple:

Argument	Contains
lrtype	Type of input event: 0 = position report 2 = account balance request 3 = daily expenditure request
lrsec	time when event arrives
lrvid	vehicle identifier
lrspd	vehicle speed
lrxway	highway the vehicle is on
lrlane	lane the vehicle is in
lrdir	direction the vehicle is travelling, 0 or 1
lrseg	segment the vehicle is on
lrpos	position of vehicle
lrquid	query identifier for event type 2 and 3
lrday	desired expenditure for a day (event type 3)

The *lr0()* procedure makes all necessary computations on each incoming tuple. The result streams are stored as tables in MySQL and these tables are read from SCSQ every second. The procedure first checks if the vehicle has velocity 0, and if that is true *accident_check()* is called which is explained in the accident detection paragraph. Then the execution is branched depending on the query type.

For type 0 (position reports) events, a call to *check_accident_cleared()* is made if *is_smashed(lrvid)* is > 0, that is *lrvid* is a smashed car. *check_accident_cleared()* places the vehicle in the *accident_remove_order* table if the smashed car is reported to be in a new segment, and therefore should not be considered smashed any more. The position reports are subsequently sent to *seg_stat()*, which handles the segment statistics. There the simulation time is first updated (in the *sim_time* table) and the statistics for the segment the car is entering is then added by the *add_stat()* procedure.

After calling *seg_stat()*, the toll for the previous segment is charged and the new toll for the current segment is calculated from *toll_calc()*. This procedure first calls *add_previous_toll()* which does the charging and then *calc_toll()* calculates the toll charges according to the formula in Section 2.5.1. Once the toll has been set, the toll alert is added to the table *output_toll_alert*. *calc_toll()* also checks if the vehicle is near an accident; if so the toll is set to 0 and an insertion is done into *output_accident_alert*.

For type 2 (account balance) events, *giveme_lasttime()* and *giveme_balance()* are called to fetch the latest update time and the current

account balance of the required vid from the *vehicle_info* table. If the vid doesn't exist in the table, the update time is set to the entering time of the event and the balance is set to 0. The time and balance is finally inserted into the *output_account_balance* table.

For type 3 (daily expenditure) events, the results are fetched by looking up the requested *vid*, *day*, and *highway* in the *historical_toll* table by the stored procedure *hist_toll()*, called from *lr0()*. The resulting expenditure value is stored in the *output_daily_exp* table.

4.1.3 Accident detection

The stored procedure *accident_check()* manages the accident detection. A car involved in an accident is placed in the s*mashed_cars* table to prevent multiple accident alerts from the same accident. If the accident detection finds that a position report event reporting has its speed equal to 0 and is not a smashed car, the vehicle is added to s*topped_cars* table. The stored procedure *accident_check()* then checks if this and another car in s*topped_cars* have reported four consecutive reports at this position. If that is true, an accident is created in the a*ccidents* table.

4.1.4 Segment statistics

LAV and NOV statistics are maintained for every segment on all highways. The LAV is calculated in *get_stat_avgv()* for a given segment with a time span of the five minutes prior to the current minute. As in [11], the LAV statistics computation part has been observed to be the bottleneck of the system.

The position reports for the latest five minutes are continuously stored in five tables, one per minute. When a position report arrives it is added to the table named *stat_min1*. Since it is possible that a car emits two position reports in one segment during the same minute, the number of reports for a segment is also stored so the average speed can be calculated.
The formula for calculating the average speed for a vehicle in one segment

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

is:
- v_{new} = New average speed
- v_{avg} = Old average speed
- v_{rep} = Last reported speed
- n_{old} = Number of reports during last minute

In the source code, the insertion to *stat_min1* is done by the SQL statement in the stored procedure *add_stat()*:

```
INSERT INTO stat_min1(segment, dir, xway,vid,lav,number)
    VALUES(lrsegment,lrdir, lrxway,lrvid, lrspeed,1)
    ON DUPLICATE KEY UPDATE
    lav=( (lav*number+lrspeed)/(number+1) ),
    number=number+1;
```

Eventually the time changes to a new minute and the content of the statistical tables are switched to match the current time:

Before minute switch		**After minute switch**	
Last 5 minute	*stat_min6*	Last 5 minute	*stat_min5*
Last 4 minute	*stat_min5*	Last 4 minute	*stat_min4*
Last 3 minute	*stat_min4*	Last 3 minute	*stat_min3*
Last 2 minute	*stat_min3*	Last 2 minute	*stat_min2*
Last 1 minute	*stat_min2*	Last 1 minute	*stat_min1*
Current minute	*stat_min1*	Current minute	Empty table

This is done in SCSQ-MySQL by first emptying *stat_min6* and then renaming the tables: *stat_min6* to *stat_min1*, *stat_min1* to *stat_min2* etc. The SQL statements TRUNCATE and RENAME is used for that in the stored procedure *switch_stat_tables()*.

When the system asks for the latest average velocity statistic of a segment for determining the toll, it is calculated from the five *stat_min2...6* tables. The LAV is calculated by taking the average velocity of the vehicles in a segment for the last five minutes, more precisely as follows:

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

j = LAV table to search
k = vehicle in segment
n = number of vehicles in segment
v_{avg} = average velocity of a vehicle in segment

The NOV is calculated in *get_stat_nov()* by counting the number of vehicles in a segment in the last minute, that is in the *stat_min2* table.
As the number of requests increases with this implementation it became clear that this solution was too slow. Many toll calculations were done for the same segment within a minute more than once, so it was natural to set up the cache tables *stat_lav_cached* and *stat_nov_cached* containing the results of the calculations, cleared at every minute.

4.2 SCSQ interface

SCSQ is used as a data driver for sending the input events to the stored procedure *lr0()* is MySQL. The source code of the SCSQ part of the LRB implementation is listed in Appendix E. The function *run_single_node()* in

SCSQL starts a simulation and returns the result. It takes six arguments: the path to the input file containing the input events, the database name, host name, user name, password for the user name, and the port number for the database.

First, *run_single_node* connects to the MySQL server with the last five function arguments. The database communication is done by the SCSQ's JDBC wrapper and JDBC. *init_lr()* is then executed which calls the stored procedure *prepare_start()* in MySQL that cleans the tables before the simulation is started.

The SCSQ function *lread()* reads the input file line by line and the function *stream_to_vector()* converts the result from *lread()* to a vector. It is then passed to the function *callLR0()* which calls the stored procedure *lr0()* in MySQL using JDBC. The MySQL procedure *lr0()* does the real simulation. The stored function *query_counter()* in SCSQ keeps track of the simulation second in the last input event to *lr0()*. At every new simulation second, *flush_mysql_result()* is called which fetches the results from the MySQL tables by SQL SELECT statements. The tables are afterwards emptied by TRUNCATE statements.

When the lines have been read in *run_single_node()*, *flush_mysql_result()* is called once more to make sure all results are fetched. All output events are also timestamped by *tslr()* in order to measure the response times.

The execution scripts *test-complete.cmd* and *test-complete.sh* execute *run_single_node()* in SCSQ and uses *writefile()* to store the results in an output file.

5 Experimental results

5.1 Performance of MySQL

Some preliminary experiments were made to determine which storage manager to use in MySQL. Two storage managers have been tested: MyISAM (persistent storage) and MEMORY (storage in main memory). To our surprise, the MyISAM showed to be 0.5% faster than MEMORY.

It was also found out that the choice of keys for the tables had a great impact on the performance. Too short composite keys increased the look-up time, too long increased the INSERT SQL statements time and choosing wrong fields as key could prolong all operations on that table. No known tools for profiling stored procedures in MySQL existed, which made index choosing even more difficult. The method that was used was to compare how long time parts of the simulation took after having changed an index.

5.2 Overhead introduced by database interfaces

To compare the overhead of SCSQ's database interface, a standalone Java program was developed. Both the SCSQ and Java program works in exactly the same way:

1. Read the input events from a file line by line
2. Send events to the stored procedure *lr0()* in MySQL
3. For each second: fetch and flush result from MySQL by SELECT and TRUNCATE SQL statements.
4. At the end of simulation: Do 3 again to fetch the remaining results.

An important difference from the SCSQ interface presented in Section 4.2 is that this slightly modified SCSQ program and Java program read the events and send them to MySQL immediately, i.e. ignoring the time the events should arrival to the system. The reason is that the performance is measured by comparing how long time it takes for MySQL to process all the events during one minute of the LRB simulation, a *simulation minute.*

A simulation minute is not an exact measurement for the time requirements, but it is a good indication. For example, if one position report takes more than 5 seconds, the probability is high that the following position report queries also takes at least 5 seconds. The reason for that is that the amount of queries increases constantly during the simulation. If a simulation minute at the end of the simulation takes less than 60 seconds, chances are high that the implementation achieves the time requirements.

The purpose of this measurement is to compare the overhead of the JDBC database interface in SCSQ with a standalone Java program. They are both executed on the same Linux machine with two Dual-Core AMD Opteron 2.8 GHz 1024 KB L2 cache processors with 8 GB main memory. The results of a 3-hours simulation with *L=0.5* are shown in Figure 6.

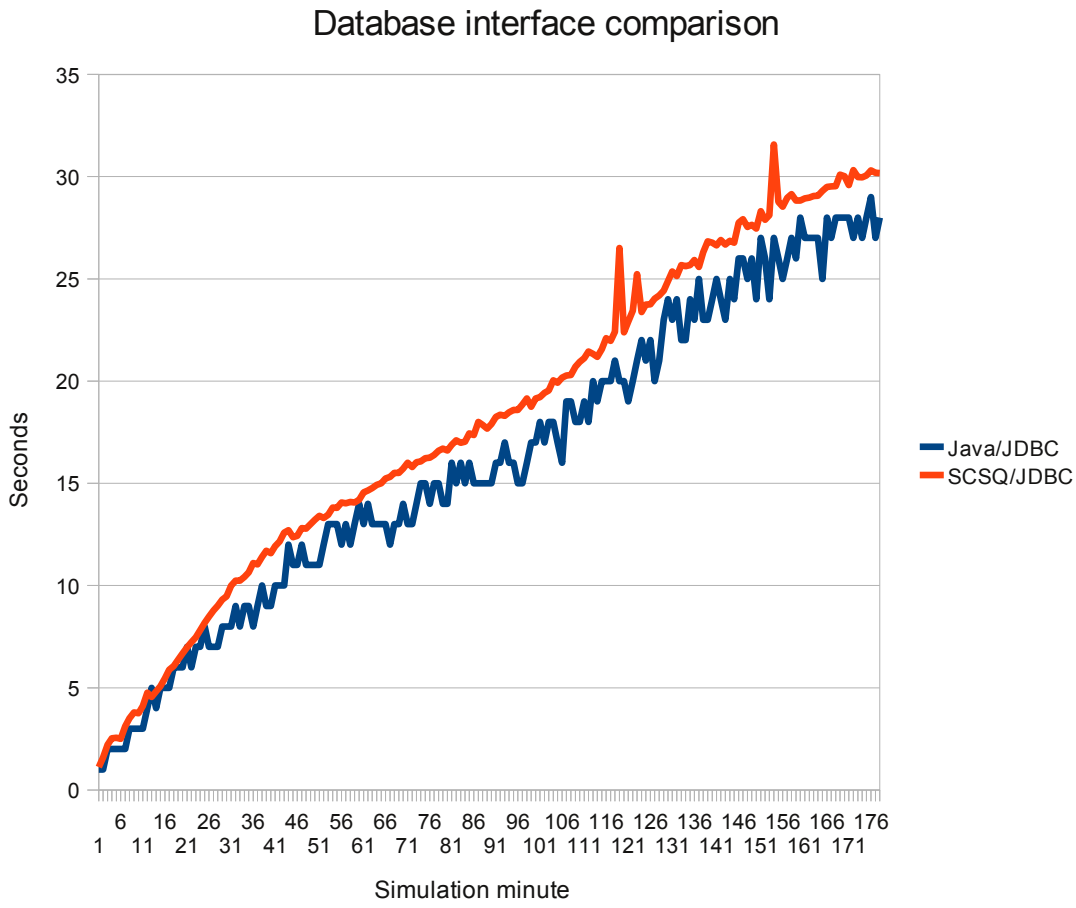

Figure 6: L=0.5 with the database interfaces SCSQ/JDBC and Java/JDBC

The total running times for the database interfaces are:

Interface	Total running time (seconds)
Java/JDBC	2 877
SCSQ/JDBC	3 257

5.3 Result of simulations in SCSQ

SCSQ-MySQL was benchmarked to determine the L-rating for a 3-hour simulation. SCSQ-MySQL was executed on a Linux machine with two Dual-Core AMD Opteron 2.8 GHz 1024 KB L2 cache processors with 8 GB main memory.

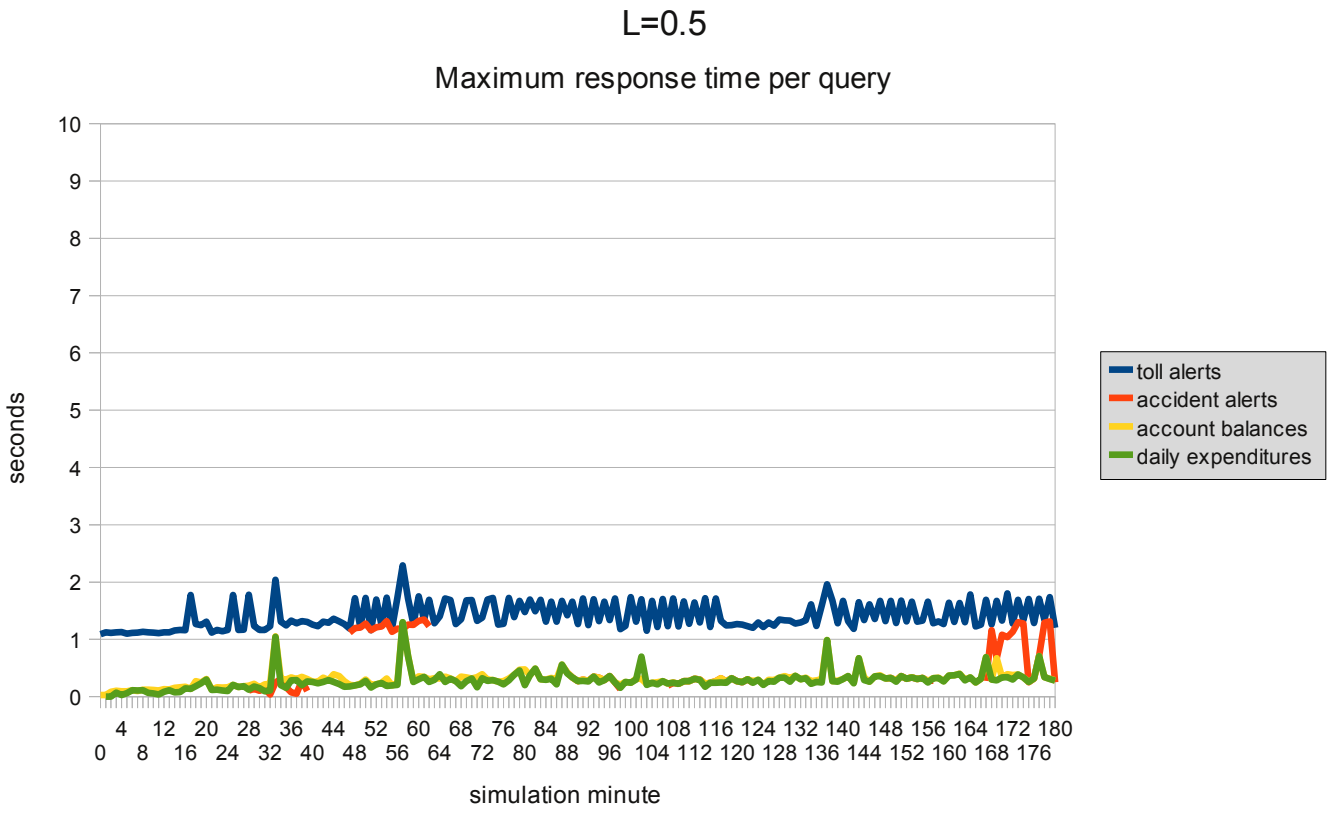

Figure 7 shows the response times for *L=0.5* grouped by the four different event types. The response time is the maximal time a query is processed in SCSQ-MySQL during one minute.

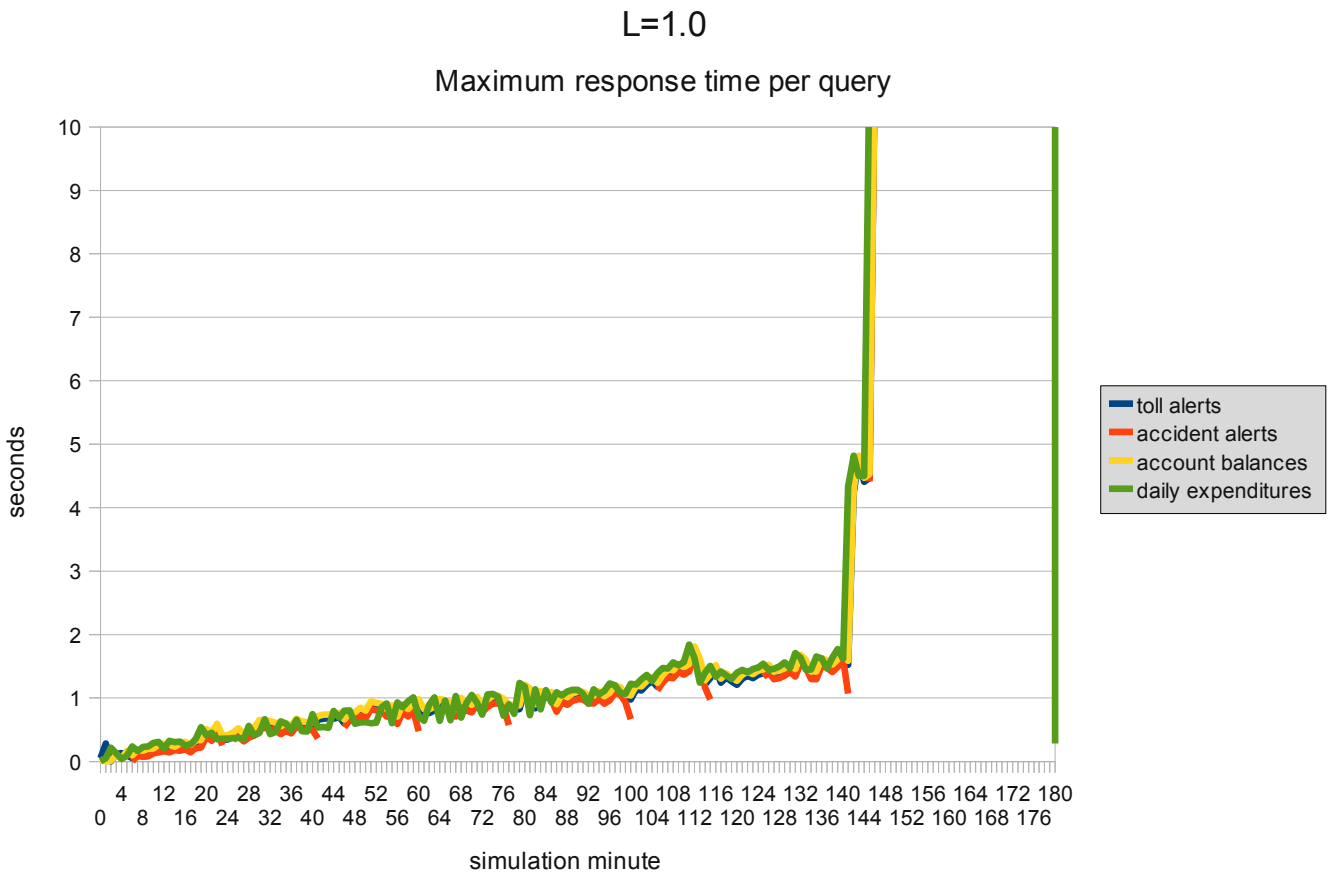

Figure 8 shows the response times for *L=1.0* grouped by the four different event types.

The maximum response times for the total simulation in Figure 7 and Figure 8 are:

Output event type	**L=0.5**	**L=1.0**
Toll alerts	2.29 seconds	493.22 seconds
Accidents alerts	1.34 seconds	457.87 seconds
Account balances	1.3 seconds	493.28 seconds
Daily expenditures	1.3 seconds	493.31 seconds

6 Related Work

There have been six published results of implementation of LRB: IBM Stream Processing Core [6], Aurora [2], XQuery [4], DataCell [7] and SCSQ-LR [11].

IBM Stream Processing Core (SPC) and Aurora both achieved *L=2.5*. They were main memory based and did not use a query language like SCSQL and SQL. The Aurora LRB implementation does not distribute the execution. The testbed of SPC was distributed over an 85-node Linux cluster each with a dual-core hyper-threaded 3 GHz Xeon processor with 2 GB RAM. The benchmark of Aurora was run on a 3 GHz Pentium box with 2 GB RAM running Linux. On the same hardware, the Aurora group obtained *L=0.5* on a LRB implementation using a unspecified commercially available DBMS.

An implementation has been done in XQuery (XML query language) with *L=1.0*, running on a Linux machine with a 2.2 GHz AMD Opteron processor and 4 GB of main memory. An open source XQuery engine was extended with continuous queries support. There was no parallelization, but the report concludes that XQuery stream processing can be implemented as efficiently as SQL stream processing.

DataCell is built on top of an open-source column-oriented main-memory DBMS kernel. The streaming functionality is implemented in *baskets*, temporary storage tables for the tuples. When a tuple arrives, continuous queries and operators queued in the system are evaluating it and then it is removed. A basket is the input to one or more query plans. The design of DataCell is made for being flexible and generic with complex queries. No parallelization was supported. *L=1.0* was achieved on a 2.4 GHz Intel Core2 Quad CPU equipped with 8 GB RAM. The response time was under 1.5 seconds.

A previously implementation of LRB in SCSQ-LR [11] achieved *L=1.5* on a 1.73 GHz machine with 1 GB RAM. The program was completely implemented in SCSQ with temporary data stored in main memory, in contrast to persistent storage in this implementation.

SCSQ-LR was also parallelized using the parallelization functions of SCSQ as published in [13]. The partitioning of the data is the same as in this Thesis with some additional hints to the scheduler. *L=64* was achieved with six machines, each of them having two quad-core Intel Xeon E5430 CPUs @ 2.66GHz and 6144 KB L2 cache.

7 Conclusions and future developments

In the experiments with SCSQ-MySQL *L=0.5* is achieved since the maximum response times never exceed the required five seconds. The experiment with *L=1.0* did not pass; after 144 simulation minutes the response times grow significantly and time requirements are therefore no longer satisfied. Before minute 144 the response times nevertheless have a linear growth. The reason for the noticeable acceleration after minute 144 is that the following input events are accumulated in the system which increases their response times. Even if *L=0.5* is worse than the LRB implementations [2], [4], [7] and [11], it shows that LRB can be implemented in a traditional DBMS. An implementation of SCSQ, [13], has previously been parallelized and work has begun to parallelize SCSQ-MySQL.

The MySQL database interface experiments show that the JDBC wrapper in SCSQ has a constant overhead of 11% compared to the JDBC interface in Java. A future work could be to investigate how to improve its performance. It is possible that *L=1.0* could be achieved with such an optimization since SCSQ-MySQL passed the response time requirements of LRB for almost the entire 3-hour simulation.

SCSQ-MySQL passes all validation and test cases.

8 Acknowledgements

I would like to thank my supervisor Erik Zeitler and professor Tore Risch for their valuable advice and contributions for completing this Thesis.

8.1 References

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

[2] A.Arasu, et al: Linear Road: A Stream Data Management Benchmark, Proc. VLDB 2004.

[3] A.Arasu et al: STREAM: The Standform Stream Data Manager, IEEE Data Engineering Bulletin, March 2003.

[4] I.Botan, et al: Extending XQuery with Window Functions., VLDB 2007

[5] S.Chandrasekaran et al: TelegraphCQ: Continuous dataflow processing for an uncertain world. Proc. CIDR 2003.

[6] N.Jain, et al: Design, Implementation, and Evaluation of the Linear Road Benchmark on the Stream Processing Core, SIGMOD 2006.

[7] L.Liarou, R. Goncalves, S. Idreos: Exploiting the Power of Relational Databases for Efficient Stream Processing, EDBT 2009.

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

[9] G.Povilavicius: A JDBC driver for an Object – Oriented Database Mediator, Uppsala Master's Thesis in Computing Science 291, ISSN 1100-1836, 2005.

[10] 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, Analysing and Integrating Heterogeneous Data, Springer, 2003.

[11] M.Svensson: Benchmarking the performance of a stream data management system, MSc Thesis UPTEC F07 105, Faculty of Science and Technology, Uppsala, November 2007.

[12] E.Zeitler and T.Risch: Processing high-volume stream queries on a supercomputer. ICDE Ph.D. Workshop 2006.

[13] E.Zeitler and T.Risch: Scalable Splitting of Massive Data Streams, in Proc. DASFAA 2010.

A. How to setup and run MySQL, SCSQ and LRB

```
Requirements
------------
SCSQ
Java
MySQL 5.0 or newer
MySQL JDBC driver

Configuration
-------------
1. Create a MySQL user and database if not done
yet:
> create user fredrik;
> create database lr;
2. Assign all available permissions to the database
for that user:
> grant all on lr.* to 'fredrik'@'localhost';
3. Modify dbhost, dbuser and dbname in install.sh
to match the database settings.
4. Execute install.sh/install.cmd in the folder
scsq/lr/mysql

Load historical input data
--------------------------
1. Set the path to the historical data file in
load_history.sql
2. Log in the MySQL ("mysql -h dbhost dname -u
"dbuser -p") and type
> source load_history.sql;

How to run the simulation
-------------------------------------
1. Edit test_single_node.osql to match the database
settings.
2. Execute "./test.sh" under Linux or “test.cmd”
```

under Windows.

3. The result is printed to the screen.

How to run the regression test

---

1. Edit test-mysql.osql to match the database settings.

2. Execute "./test-regress.sh" under Linux or "test-regress.cmd" under Windows.

B. Source code of MySQL: lr.sql

```
DROP FUNCTION IF EXISTS hist_toll;
DROP FUNCTION IF EXISTS get_second;
DROP FUNCTION IF EXISTS get_minute;
DROP FUNCTION IF EXISTS get_stat_nov;
DROP FUNCTION IF EXISTS get_stat_avgv;
DROP PROCEDURE IF EXISTS init_time;
DROP PROCEDURE IF EXISTS set_second;
DROP PROCEDURE IF EXISTS set_minute;
DROP PROCEDURE IF EXISTS switch_stat_tables;
DROP PROCEDURE IF EXISTS add_stat;
DROP PROCEDURE IF EXISTS seg_stat;
DROP PROCEDURE IF EXISTS toll_calc;
DROP PROCEDURE IF EXISTS check_accident_cleared;
DROP PROCEDURE IF EXISTS lr0;
DROP PROCEDURE IF EXISTS prepare_start;

delimiter //

# begin time functions
CREATE PROCEDURE init_time()
BEGIN
  TRUNCATE sim_time;
  INSERT INTO sim_time(sim_minute,sim_second) VALUES(1,0);
END;

CREATE PROCEDURE set_second(newsec SMALLINT)
BEGIN
  UPDATE sim_time SET sim_second=newsec LIMIT 1;
END;

CREATE FUNCTION get_second()
RETURNS SMALLINT
BEGIN
  DECLARE result SMALLINT;
  SELECT sim_second INTO result FROM sim_time LIMIT 1;
  RETURN result;
END;

CREATE PROCEDURE set_minute(newmin SMALLINT UNSIGNED)
BEGIN
  UPDATE sim_time SET sim_minute=newmin LIMIT 1;
END;

CREATE FUNCTION get_minute()
RETURNS TINYINT UNSIGNED
BEGIN
  DECLARE result TINYINT UNSIGNED;
  SELECT sim_minute INTO result FROM sim_time LIMIT 1;
  RETURN result;
END;
# end time functions

CREATE PROCEDURE switch_stat_tables()
BEGIN
  TRUNCATE TABLE stat_nov_cached;
  TRUNCATE TABLE stat_lav_cached;
  TRUNCATE TABLE stat_min6;
  RENAME TABLE  stat_min1 TO stat_tmp1,
                stat_min2 TO stat_tmp2,
                stat_min3 TO stat_tmp3,
                stat_min4 TO stat_tmp4,
                stat_min5 TO stat_tmp5,
```

```
                stat_min6 TO stat_tmp6;
  RENAME TABLE  stat_tmp1 TO stat_min2,
                stat_tmp2 TO stat_min3,
                stat_tmp3 TO stat_min4,
                stat_tmp4 TO stat_min5,
                stat_tmp5 TO stat_min6,
                stat_tmp6 TO stat_min1;
END;
CREATE FUNCTION get_stat_nov( lrsegment TINYINT, lrxway TINYINT, lrdir BOOLEAN )
RETURNS SMALLINT
BEGIN
  DECLARE result SMALLINT;
  SELECT nov INTO result FROM stat_nov_cached WHERE xway=lrxway AND dir=lrdir
    AND segment=lrsegment LIMIT 1;
--   Calculate nov if it's not in cache
  IF result IS NULL THEN
    SELECT count(*) INTO result FROM stat_min2 WHERE xway=lrxway AND dir=lrdir
    AND segment=lrsegment LIMIT 1;
    IF result IS NULL THEN
      SET result = 0;
    END IF;
    INSERT INTO stat_nov_cached(segment, dir, xway,nov)
      VALUES(lrsegment,lrdir, lrxway,result);
  END IF;
  RETURN result;
END;
CREATE FUNCTION get_stat_avgv( lrsegment TINYINT, lrxway TINYINT,
  lrdir BOOLEAN )
RETURNS TINYINT
BEGIN
  DECLARE result TINYINT;
  SELECT avgv INTO result FROM stat_lav_cached WHERE xway=lrxway AND dir=lrdir
    AND segment=lrsegment;
  IF result IS NULL THEN
  BEGIN
    SELECT floor( avg(r)) FROM (
      SELECT avg(lav) r FROM stat_min2 WHERE xway=lrxway AND dir=lrdir
        AND segment=lrsegment UNION ALL
      SELECT avg(lav) r FROM stat_min3 WHERE xway=lrxway AND dir=lrdir
        AND segment=lrsegment UNION ALL
      SELECT avg(lav) r FROM stat_min4 WHERE xway=lrxway AND dir=lrdir
        AND segment=lrsegment UNION ALL
      SELECT avg(lav) r FROM stat_min5 WHERE xway=lrxway AND dir=lrdir
        AND segment=lrsegment UNION ALL
      SELECT avg(lav) r FROM stat_min6 WHERE xway=lrxway AND dir=lrdir
        AND segment=lrsegment ) r INTO result;
    IF result IS NULL THEN
      SET result = -1;
    END IF;
    INSERT INTO stat_lav_cached( xway, dir, segment, avgv)
      VALUES(lrxway, lrdir, lrsegment, result);
    END;
  END IF;

  RETURN result;
END;
CREATE PROCEDURE add_stat(lrvid INTEGER, lrspeed TINYINT, lrxway TINYINT,
  lrdir BOOLEAN, lrsegment TINYINT)
BEGIN

  INSERT INTO stat_min1(segment, dir, xway,vid,lav,number)
    VALUES(lrsegment,lrdir, lrxway,lrvid, lrspeed,1) ON DUPLICATE KEY
    UPDATE lav=((lav*number+lrspeed)/(number+1)), number=number+1;
END;
CREATE PROCEDURE seg_stat(lrtime SMALLINT, lrvid INTEGER, lrspeed TINYINT,
```

```
  lrxway TINYINT, lrdir BOOLEAN, lrsegment TINYINT)
BEGIN
DECLARE current_second SMALLINT;
DECLARE current_minute TINYINT UNSIGNED;
SELECT get_second() INTO current_second;
  IF lrtime > current_second THEN
    CALL set_second(lrtime);
    CALL remove_stopped_cars(lrtime);

    SELECT get_minute() INTO current_minute;
    IF floor(lrtime/60)+1 > current_minute THEN
    BEGIN
        DECLARE loop_counter TINYINT DEFAULT 0;
        DECLARE new_minute TINYINT UNSIGNED DEFAULT 0;
        CALL set_minute( (floor(lrtime/60)+1) );
  --       WHILE xway<100 DO
  --     CALL check_kill_accident(xway);
  --     SET xway = xway+1;
  --       END WHILE;
        CALL check_kill_accident(lrxway);
        SELECT get_minute() INTO new_minute;
--      INSERT INTO min_test(old, new) VALUES(current_minute, new_minute);
--      WHILE loop_counter<( new_minute-current_minute ) DO
          CALL switch_stat_tables();
--        SET loop_counter=loop_counter+1;
--         END WHILE;
      END;
      END IF;
    END IF;
    CALL add_stat(lrvid, lrspeed, lrxway, lrdir, lrsegment);
END;
CREATE FUNCTION hist_toll( lrvid INTEGER, lrday TINYINT, lrxway TINYINT)
RETURNS DECIMAL(9,1)
BEGIN
  DECLARE result DECIMAL(9,1);
  SELECT toll INTO result FROM historical_toll WHERE vid=lrvid
    AND day=lrday AND xway=lrxway LIMIT 1;
  RETURN result;
END;

# begin toll calculation
CREATE PROCEDURE toll_calc( lrsec SMALLINT, lrvid INTEGER, lrxway TINYINT,
  lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrspd TINYINT, lrlane TINYINT )
BEGIN
IF lrlane <> 4 THEN
  BEGIN
  DECLARE res BOOLEAN;
  SELECT check_new_seg_report( lrvid, lrseg, lrxway) INTO res;
  IF res=0 THEN
--   add toll from previous segment
    CALL add_previous_toll( lrsec, lrvid, lrxway, lrpos, lrseg, lrdir, lrspd );
--     calculate toll for the new segment
    CALL calc_toll( lrsec, lrvid, lrxway, lrpos, lrseg, lrdir, lrspd );
  END IF;
  END;
END IF;
END;

CREATE PROCEDURE check_accident_cleared( lrsec SMALLINT, lrvid INTEGER,
  lrpos MEDIUMINT, lrseg TINYINT, lrxway TINYINT )
BEGIN
  DECLARE ifres INTEGER DEFAULT 0;
--   SELECT is_smashed(lrvid) INTO ifres;
--   IF ifres > 0 THEN
      SELECT giveme_smashed_pos(lrvid) INTO ifres;
```

```
      IF ( ifres <> lrpos)  THEN
      BEGIN
        CALL add_accident_remove_order( giveme_smashed_xway(lrvid),
          ceil((lrsec+120)/60) );
--      INSERT INTO print_accidents(xway,segment,direction,vid,time)
--        VALUES(-1,get_minute(),ceil((lrsec+120)/60),lrvid,get_second());
        END;
      END IF;
--   END IF;
END;
# end stopped calculation

CREATE PROCEDURE lr0( lrtype TINYINT, lrsec SMALLINT, lrvid INTEGER,
  lrspd TINYINT, lrxway TINYINT, lrlane TINYINT, lrdir BOOLEAN, lrseg TINYINT,
  lrpos MEDIUMINT, lrquid INTEGER, lrday TINYINT )
BEGIN
  DECLARE ifres BOOLEAN DEFAULT 0;
  IF lrspd=0 THEN
    CALL accident_check( lrsec, lrvid, lrxway, lrlane, lrdir, lrseg, lrpos );
  END IF;
  CASE lrtype
  WHEN 0 THEN
  BEGIN
    SELECT is_smashed(lrvid) INTO ifres;
    IF ifres>0 THEN
      CALL check_accident_cleared( lrsec, lrvid, lrpos, lrseg, lrxway );
    END IF;
    CALL seg_stat( lrsec, lrvid, lrspd, lrxway, lrdir, lrseg );
    CALL toll_calc( lrsec, lrvid, lrxway, lrpos, lrseg, lrdir, lrspd, lrlane );
  END;
  WHEN 3 THEN
  BEGIN
    DECLARE expenditure DECIMAL(9,1);
    SELECT hist_toll( lrvid, lrday, lrxway) INTO expenditure;
    IF expenditure IS NOT NULL THEN
      INSERT INTO output_daily_exp(time, emit_time,quid,expenditure)
        VALUES(lrsec, 0, lrquid, expenditure );
    END IF;
  END;
  WHEN 2 THEN
  BEGIN
    DECLARE lrlasttime SMALLINT;
    DECLARE lrbalance DECIMAL(9,1);
    SELECT giveme_lasttime(lrvid) INTO lrlasttime;
    SELECT giveme_balance(lrvid) INTO lrbalance;
    /* Account balance */
    IF lrbalance IS NOT NULL THEN
      INSERT INTO output_account_balance(time, emit_time,quid,lasttime,balance)
        VALUES(lrsec, 0, lrquid, lrlasttime, lrbalance );
    ELSE
      INSERT INTO output_account_balance(time, emit_time,quid,lasttime,balance)
        VALUES(lrsec, 0, lrquid, lrsec, 0.0 );
    END IF;
  END;
  WHEN 4 THEN
  BEGIN
--     Ignore travel time estimation requests
  END;
  END CASE;
END;


CREATE PROCEDURE prepare_start()
BEGIN
  TRUNCATE accident_remove_order;
```

```
  TRUNCATE accidents;
  TRUNCATE collided_cars;
  TRUNCATE output_accident_alert;
  TRUNCATE output_account_balance;
  TRUNCATE output_daily_exp;
  TRUNCATE output_toll_alert;
  TRUNCATE sim_time;
  TRUNCATE smashed_cars;
  TRUNCATE stat_min1;
  TRUNCATE stat_min2;
  TRUNCATE stat_min3;
  TRUNCATE stat_min4;
  TRUNCATE stat_min5;
  TRUNCATE stat_min6;
  TRUNCATE stat_lav_cached;
  TRUNCATE stat_nov_cached;
  TRUNCATE stopped_cars;
  TRUNCATE vehicle_info;
  CALL init_time();
END;
//
delimiter ;
```

C. Source code of MySQL: toll.sql

```
DROP PROCEDURE IF EXISTS set_vehicle_info;
DROP PROCEDURE IF EXISTS add_vehicle_info;
DROP PROCEDURE IF EXISTS add_previous_toll;
DROP PROCEDURE IF EXISTS update_vehicle_info;
DROP PROCEDURE IF EXISTS update_vehicle_toll;
DROP PROCEDURE IF EXISTS calc_toll;
DROP FUNCTION IF EXISTS check_new_seg_report;
DROP FUNCTION IF EXISTS giveme_balance;
DROP FUNCTION IF EXISTS giveme_lasttime;

delimiter //

CREATE PROCEDURE set_vehicle_info( lrvid INTEGER, lrxway TINYINT,
  lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrbal DECIMAL(9,1),
  lrlastupdate SMALLINT, lrvav TINYINT, lrnextexp DECIMAL(9,1) )
BEGIN
  INSERT INTO vehicle_info(vid, xway, segment, direction, pos, balance,
    lastupdate, vav, nextexp) VALUES(lrvid, lrxway, lrseg, lrdir, lrpos, lrbal,
    lrlastupdate, lrvav, lrnextexp) ON DUPLICATE KEY UPDATE xway=lrxway,
    segment=lrseg, direction=lrdir, pos=lrpos, balance=lrbal,
    lastupdate=lrlastupdate, vav=lrvav, nextexp=lrnextexp;
END;

CREATE PROCEDURE add_vehicle_info( lrvid INTEGER, lrxway TINYINT,
  lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrbal DECIMAL(9,1),
  lrlastupdate SMALLINT, lrvav TINYINT, lrnextexp DECIMAL(9,1) )
BEGIN
  DECLARE EXIT HANDLER FOR 1062 BEGIN END;
  INSERT INTO vehicle_info(vid, xway, segment, direction, pos, balance,
  lastupdate, vav, nextexp) VALUES(lrvid, lrxway, lrseg, lrdir, lrpos, lrbal,
  lrlastupdate, lrvav, lrnextexp);
END;

CREATE FUNCTION check_new_seg_report( lrvid INTEGER, lrseg TINYINT,
  lrxway TINYINT )
RETURNS BOOLEAN
BEGIN
  DECLARE res BOOLEAN;
  SELECT count(*)>0 INTO res FROM vehicle_info WHERE vid=lrvid
    AND segment=lrseg AND xway=lrxway;
  RETURN res;
END;

CREATE PROCEDURE add_previous_toll( lrsec SMALLINT, lrvid INTEGER,
  lrxway TINYINT, lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrspd TINYINT )
BEGIN
  DECLARE upbalance DECIMAL(9,1) DEFAULT 0;
  SELECT nextexp+balance INTO upbalance FROM vehicle_info WHERE vid=lrvid;
  CALL set_vehicle_info( lrvid, lrxway, lrpos, lrseg,
    lrdir, upbalance, lrsec, lrspd, 0.0 );
END;

CREATE FUNCTION giveme_balance( lrvid INTEGER )
RETURNS DECIMAL(9,1)
BEGIN
DECLARE res DECIMAL(9,1);
SELECT balance INTO res FROM vehicle_info WHERE vid=lrvid;
RETURN res;
END;

CREATE FUNCTION giveme_lasttime( lrvid INTEGER )
RETURNS SMALLINT
```

```
BEGIN
DECLARE res SMALLINT;
SELECT lastupdate INTO res FROM vehicle_info WHERE vid=lrvid;
RETURN res;
END;

CREATE PROCEDURE update_vehicle_info( lrsec SMALLINT, lrvid INTEGER,
  lrxway TINYINT, lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrspd TINYINT)
BEGIN
  CALL add_previous_toll( lrsec, lrvid, lrxway, lrpos, lrseg, lrdir, lrspd);
END;

CREATE PROCEDURE update_vehicle_toll( lrsec SMALLINT, lrvid INTEGER, lrxway
  TINYINT, lrpos MEDIUMINT, lrseg TINYINT,
  lrdir BOOLEAN, lrspd TINYINT, lrtoll DECIMAL(9,1))
BEGIN
  DECLARE EXIT HANDLER FOR 1048 BEGIN END;
  CALL set_vehicle_info( lrvid, lrxway, lrpos, lrseg, lrdir,
  giveme_balance(lrvid) ,lrsec, lrspd, lrtoll );
END;

CREATE PROCEDURE calc_toll( lrsec SMALLINT, lrvid INTEGER, lrxway TINYINT,
  lrpos MEDIUMINT, lrseg TINYINT, lrdir BOOLEAN, lrspd TINYINT )
BEGIN
  DECLARE action_area_res BOOLEAN DEFAULT 0;
  DECLARE lrtoll DECIMAL(9,1);
  DECLARE lravgv DECIMAL(9,1);
  SELECT is_accident_area(lrxway,lrseg, lrdir) INTO action_area_res;
  IF action_area_res=1 THEN
    INSERT INTO output_accident_alert( time, emit_time, vid, segment)
      VALUES(lrsec, 0, lrvid, get_accident_seg(lrxway) );
    SET lrtoll = 0.0;
    SELECT get_stat_avgv( lrseg, lrxway, lrdir ) INTO lravgv;
  ELSE
    BEGIN
      DECLARE nr_of_cars MEDIUMINT DEFAULT 0;
      SELECT get_stat_nov( lrseg, lrxway, lrdir ) INTO nr_of_cars;
      SELECT get_stat_avgv( lrseg, lrxway, lrdir ) INTO lravgv;
      IF (nr_of_cars<=50) OR (lravgv>=40) THEN
        SET lrtoll=0.0;
      ELSE
        SET lrtoll=2*(nr_of_cars-50)*(nr_of_cars-50);
      END IF;
--      INSERT INTO toll_debug(sec, num_cars, avgv, vid, toll)
--      VALUES(lrsec, nr_of_cars,lravgv, lrvid, lrtoll);
    END;
  END IF;
  CALL update_vehicle_toll( lrsec, lrvid, lrxway, lrpos, lrseg, lrdir, lrspd, lrtoll );
  INSERT INTO output_toll_alert( vid, time, emit_time, vavg, toll)
    VALUES(lrvid, lrsec, 0, lravgv, lrtoll );
END;
//
delimiter ;
```

D. Source code of MySQL: accidents.sql

```
DROP FUNCTION IF EXISTS is_accident_area;
DROP FUNCTION IF EXISTS get_accident_seg;
DROP FUNCTION IF EXISTS is_smashed;
DROP FUNCTION IF EXISTS giveme_smashed_pos;
DROP FUNCTION IF EXISTS giveme_smashed_xway;
DROP PROCEDURE IF EXISTS add_accident;
DROP PROCEDURE IF EXISTS add_stopped_car;
DROP PROCEDURE IF EXISTS remove_stopped_cars;
DROP PROCEDURE IF EXISTS add_smashed_car;
DROP PROCEDURE IF EXISTS collided_vids_at;
DROP PROCEDURE IF EXISTS add_accident_remove_order;
DROP PROCEDURE IF EXISTS remove_accidents;
DROP PROCEDURE IF EXISTS check_kill_accident;
DROP PROCEDURE IF EXISTS accident_check;

delimiter //

CREATE PROCEDURE add_accident( lrxway TINYINT, lrseg TINYINT, lrdir BOOLEAN,
  lrtime SMALLINT, lrmin INTEGER )
BEGIN
  DECLARE EXIT HANDLER FOR 1062 BEGIN END;
  INSERT INTO accidents(xway,segment,direction,time,min)
    VALUES(lrxway, lrseg, lrdir, lrtime, lrmin);
END;

CREATE FUNCTION is_accident_area( lrxway TINYINT, lrseg TINYINT, lrdir BOOLEAN )
RETURNS BOOLEAN
BEGIN
  DECLARE res BOOLEAN DEFAULT false;
  SELECT count(*)>0 INTO res FROM accidents WHERE xway=lrxway AND
    (lrdir=1 AND direction=1 AND segment<=lrseg AND segment>lrseg-5 AND
    get_minute()>min) OR (lrdir=0 AND direction=0 AND segment>=lrseg AND
    segment<lrseg+5 AND get_minute()>min);
  RETURN res;
END;

CREATE FUNCTION get_accident_seg( lrxway TINYINT )
RETURNS TINYINT
BEGIN
  DECLARE res INTEGER DEFAULT 0;
  SELECT max(time) INTO res FROM accidents WHERE xway=lrxway;
  SELECT segment INTO res FROM accidents WHERE xway=lrxway AND time=res;
  RETURN res;
END;

CREATE PROCEDURE add_stopped_car( lrvid INTEGER, lrxway TINYINT,
  lrpos MEDIUMINT, lrdir BOOLEAN, lrtime SMALLINT, lrlane TINYINT )
BEGIN
  DECLARE EXIT HANDLER FOR 1062 BEGIN END;
  INSERT INTO stopped_cars(pos,xway,direction,time,vid,lane)
    VALUES(lrpos, lrxway, lrdir, lrtime, lrvid, lrlane);
END;

CREATE PROCEDURE remove_stopped_cars(sec SMALLINT)
BEGIN
  DELETE FROM stopped_cars WHERE time<sec-120;
END;

CREATE PROCEDURE add_smashed_car(lrvid INTEGER, lrpos MEDIUMINT, lrxway TINYINT,
  lrseg TINYINT, lrdir BOOLEAN, lrlane TINYINT )
BEGIN
  DECLARE EXIT HANDLER FOR 1062 BEGIN END;
```

```
  INSERT INTO smashed_cars(pos,xway,direction,segment,vid,lane)
    VALUES(lrpos, lrxway, lrdir, lrseg, lrvid, lrlane);
END;

CREATE FUNCTION is_smashed( lrvid INTEGER )
RETURNS BOOLEAN
BEGIN
DECLARE res BOOLEAN;
  SELECT count(*) INTO res FROM smashed_cars WHERE vid=lrvid;
RETURN res;
END;

CREATE FUNCTION giveme_smashed_pos( lrvid INTEGER )
RETURNS MEDIUMINT
BEGIN
  DECLARE res MEDIUMINT;
  SELECT pos INTO res FROM smashed_cars WHERE vid=lrvid;
  RETURN res;
END;

CREATE FUNCTION giveme_smashed_xway( lrvid INTEGER )
RETURNS TINYINT
BEGIN
  DECLARE res TINYINT;
  SELECT xway INTO res FROM smashed_cars WHERE vid=lrvid;
  RETURN res;
END;

CREATE PROCEDURE collided_vids_at( lrxway TINYINT, lrpos MEDIUMINT,
  lrdir BOOLEAN, lrlane TINYINT )
BEGIN
  TRUNCATE TABLE collided_cars;
  TRUNCATE TABLE collided_cars_temp;
  INSERT INTO collided_cars_temp(vid) SELECT vid FROM stopped_cars WHERE
    pos=lrpos AND xway=lrxway AND direction=lrdir AND lane=lrlane;
  INSERT INTO collided_cars(vid,nr_stopped) SELECT vid,count(*)
    FROM collided_cars_temp GROUP BY vid;
  DELETE FROM collided_cars WHERE nr_stopped <4;
END;

CREATE PROCEDURE add_accident_remove_order( lrxway TINYINT, lrmin INTEGER )
BEGIN
  DECLARE EXIT HANDLER FOR 1062 BEGIN END;
  INSERT INTO accident_remove_order(xway,min) VALUES(lrxway, lrmin );
END;

CREATE PROCEDURE remove_accidents(lrxway TINYINT)
BEGIN
  DELETE FROM accidents WHERE xway=lrxway;
  DELETE FROM smashed_cars WHERE xway=lrxway;
  DELETE FROM accident_remove_order WHERE xway=lrxway;
END;

CREATE PROCEDURE check_kill_accident(lrxway TINYINT)
BEGIN
  DECLARE ifres BOOLEAN;
  SELECT get_minute() >= MAX(min) FROM accident_remove_order
    WHERE xway=lrxway INTO ifres;
  IF ifres = 1 THEN
    CALL remove_accidents(lrxway);
  END IF;
END;

CREATE PROCEDURE accident_check( lrsec SMALLINT, lrvid INTEGER, lrxway TINYINT,
  lrlane TINYINT, lrdir BOOLEAN, lrseg TINYINT, lrpos MEDIUMINT )
```

```
BEGIN
  DECLARE ifres BOOLEAN DEFAULT 0;
  DECLARE stopped_count INTEGER DEFAULT 0;
  SELECT is_smashed(lrvid) INTO ifres;
  IF ifres<1 THEN
    CALL add_stopped_car( lrvid, lrxway, lrpos, lrdir, lrsec, lrlane );
    CALL collided_vids_at( lrxway, lrpos, lrdir, lrlane );
    SELECT count(*) INTO stopped_count FROM collided_cars;
    IF stopped_count>1 THEN
    BEGIN
      DECLARE done INTEGER DEFAULT 0;
      DECLARE fetched_vid INTEGER;
      DECLARE cur CURSOR FOR SELECT vid FROM collided_cars;
      DECLARE CONTINUE HANDLER FOR NOT FOUND SET done=1;
      OPEN cur;
      REPEAT
        FETCH cur INTO fetched_vid;
        CALL add_smashed_car(fetched_vid, lrpos, lrxway, lrseg, lrdir, lrlane);
      UNTIL DONE END REPEAT;
      CLOSE cur;
--       INSERT INTO print_accidents( xway, segment, direction, vid, time)
--       VALUES( lrxway, lrseg, lrdir, lrvid, lrsec );
      CALL add_accident( lrxway, lrseg, lrdir, lrsec, get_minute() );
    END;
    END IF;
  END IF;
END;
//
delimiter ;
```

E. Source code of MySQL interface in SCSQ

```
create function init_lr()->vector
  as select sql( my_jdbc(), "CALL prepare_start();" );

create function query_counter()->Integer as stored;

set query_counter()=0;

create function flush_mysql_result()->Vector of Integer as
begin
    result concat({1}, sql( my_jdbc(),
                        "SELECT * FROM output_accident_alert;"));
    result concat({2}, sql( my_jdbc(),
                         "SELECT * FROM output_account_balance;"));
    result concat({3}, sql( my_jdbc(),
                         "SELECT * FROM output_daily_exp;"));
    result concat({0}, sql( my_jdbc(),
                         "SELECT vid, time, emit_time, vavg, toll
                          FROM output_toll_alert;"));
    sql( my_jdbc(), "TRUNCATE output_accident_alert;");
    sql( my_jdbc(), "TRUNCATE output_account_balance;");
    sql( my_jdbc(), "TRUNCATE output_daily_exp;");
    sql( my_jdbc(), "TRUNCATE output_toll_alert;");
end;

create function callLR0( vector of integer x)->Vector of Integer as
begin
  select sql( my_jdbc(), "CALL lr0(?,?,?,?,?,?,?,?,?,?,?)",
              {x[0],x[1],x[2],x[3],x[4],x[5],x[6],x[7],x[8],x[9],x[14]});
  if x[1]>query_counter() then
    begin
      set query_counter()=x[1];
      result flush_mysql_result();
    end;
end;
 create function run_single_node(Charstring filename, Charstring dbname,
                                 Charstring host, Charstring dbuser,
                                 Charstring dbpass, Integer port) -> Vector of Integer as
 begin
   set my_jdbc() = jdbc("lrdb", "com.mysql.jdbc.Driver");
   connect(my_jdbc(), "jdbc:mysql://" + host + ":" + port + "/"
          + dbname, dbuser, dbpass );
   init_lr();
   result tslr(callLR0(stream_to_vector( lread(filename, 1) ) ));
   result tslr(flush_mysql_result());
 end;
```