Scaling out parallel data stream access to a relational database

Javad Bakhshijooybari
Abstract

Scaling out parallel data stream access to a

Javad Bakhshijooaybari

The rise of new applications requiring processing high volumes of continuous and real time data has created the demand for data stream management systems (DSMSs). Applications using DSMSs often need to access historical data saved on disk to analyze, mine, and process streaming data. The historical data is persistent and often is very voluminous and is therefore usually stored in a relational database. If this relational database is continuously accessed from the DSMS it will become a bottleneck of the system. The linear road benchmark (LRB) is a simulated data stream benchmark, which includes access to historical data stored on disk. If the query load per unit of time becomes too large the relational database access will become a bottleneck for the DSMS processing. In this thesis we scale out a relational DBMS storing the historical LRB data in order to eliminate this bottleneck. Experiments were done on LRB with access to a scaled out MySQL databases running on a single NUMA machine with 64 cores. Scaling out the relational database access is shown to make it possible to run LRB with L rating of 512 and beyond for only daily expenditure queries and L rating of 150 for all the LRB implemented in the DSMS SCSQ.
I would like to dedicate this work to my beloved and supporting wife who stood beside me throughout the long period of my studies far from home.
# Contents

1. Introduction .................................................................................. 4

2. Background .................................................................................. 6
   2.1 Data Stream Management Systems (DSMSs) ................................. 6
   2.2 The Linear Road Benchmark (LRB) .............................................. 7
       2.2.1 Query requirements ......................................................... 8
   2.3 SCSQ ....................................................................................... 9
       2.3.1 SCSQ Architecture ........................................................... 10
       2.3.2 SCSQ Parallelization Functions ......................................... 11

3. System Architecture ..................................................................... 13

4. Implementation ............................................................................ 15
   4.1 Simulated SCSQ implementation ................................................ 15
   4.2 Naïve SCSQ implementation ..................................................... 16
   4.3 Optimized SCSQ implementation .............................................. 17
       4.3.1 Optimization of the reading process ................................. 17
       4.3.2 Optimization of the writing process ................................. 18

5. Input Data Analysis ..................................................................... 21

6. Experiments .................................................................................. 23
   6.1 Experimental setup .................................................................... 23
       6.1.1 Starting multiple MySQL Instances on one machine .......... 24
       6.1.2 Experimental initializations ............................................ 25
       6.1.3 Initialization conclusion .................................................. 27
   6.2 Results for simulated execution .............................................. 27
   6.3 Results for SCSQ implementation ........................................... 28

7. Related Work ............................................................................... 30

8. Conclusions and future developments ........................................ 32

9. Acknowledgments ....................................................................... 33

References ....................................................................................... 33

Appendix ......................................................................................... 35
   A. Naïve implementation in SCSQ ............................................... 35
   B. Optimized implementation in SCSQ ......................................... 36
C. Historical data generator .......................................................... 37
D. Functions Used in SCSQ ........................................................... 38
E. Sample my.cnf file .................................................................. 39
1. Introduction

During the past few years new applications have raised where large volumes of real time, continuous and ordered data, so called streaming data, is processed. Examples are applications that monitor sensor networks, internet traffic, financial tickers, and stock market data in order to find signs of abnormal activity and processing them for further purposes. Because of the high volume and stream rate it is not feasible to store the stream in its entirety on disks; as is done in traditional Data Base Management Systems (DBMSs). Therefore Data Stream Management Systems (DSMSs) have been introduced to handle queries over high volume streaming data. DSMS queries are different from conventional database queries in SQL where data is expressed over stored tables. The result of a DSMS query can be a stream rather than a table as in conventional databases and, therefore, stream queries are continuous in that they run all the time until they are terminated. SCSQ [15], STREAM [2], Aurora [1] and TelegraphCQ [5] are some implementations of DSMSs.

SCSQ (Super Computer Stream Query processor) is an Object oriented main memory data stream manager developed at Uppsala University. The performance of SCSQ has been evaluated using the Linear Road Benchmark (LRB) [3] for DSMSs. LRB simulates an expressway system with dynamically varying toll rates producing data streams to be processed by a DSMS. The performance of a DSMS is measured by how many expressways it can handle simultaneously, called the L-rating. SCSQ allows calling external back-end relational databases, e.g. MySQL, using JDBC. For LRB this is used for accessing historical data used in daily expenditure queries of LRB input streams. Previous studies [16] indicate that scsq-plr can handle queries for historical data up to L=64 on a single MySQL database, but for L-ratings greater than 64 the access to the relational database becomes a bottleneck.

The main focus of this project is to scale out the access to MySQL instances during the execution of the daily expenditure queries in LRB. The results presented in the thesis can be used for commercial applications. For example, in a stock market monitoring application it is important to have access to the historical data of a certain company in order to make different analyses. In these kinds of applications DSMSs are used to process continuous queries and relational databases are used to store the historical data. In order to handle very high stream rates joined with the historical data, the relational database has to be scaled out. In general there is a point where the
demands from the database users will rise so that the DBMS resource consumption exceeds the capabilities of the underlying using processing node, i.e. CPU cores. At this point it is important and necessary to scale out the database so it can handle the queries with a reasonable response time.

This thesis is separated into two parts; the first part will examine the $L$ rating for only one kind of continuous queries in LRB (Daily expenditure queries) in a SCSQ-like architecture. The second part will examine the $L$-rating for the whole LRB implemented in SCSQ.
2. Background

The technologies used in this thesis are SCSQ and MySQL. The standard benchmark LRB has been used in order to evaluate the performance of the stream management system and MySQL. In this section a description of the above architectures will be described.

2.1 Data Stream Management Systems (DSMSs)

Traditional DBMSs are very powerful tools which are used in various applications that need complex querying over persistent data storage. However, in the past several years the need of systems to manage streams of data have emerged. Continuously growing ordered sequences of data in real time are the main characteristics of streams that make them different from ordinary data stored in a database. These characteristics impose special requirements on a DSMS [7]. Figure 1 shows the overall architecture of a typical DSMS.

As shown in the figure, data arrive to the DSMS in forms of streams. These streams can differ in data rates, data types, and time between elements.
of a stream. The DSMS can process continuous queries (CQs) over incoming streams in addition to one-time regular database queries. While regular queries delivers the result on demand from the user, CQs keep delivering results as streams until they are stopped by the user or the stream is stopped.

Two main differences between a DBMS and a DSMS are [6]:
1. A conventional DBMS stores the data set on the disk persistently, while a DSMS processes data continuously over different number of input streams without necessarily storing it.
2. In regular databases querying is passive, i.e. applications send queries to DBMS and the results are returned after they are processed. In contrast, in DSMSs a query actively returns results until it is terminated by the user.

2.2 The Linear Road Benchmark (LRB)

The Linear Road Benchmark [3] is a simulation of a highway system with variable tolling. This means charging a vehicle with different toll rates based on the time of day or level of congestion of a roadway. The highways are located in a made-up city called Linear City. This city contains $L$ number of expressways, each running horizontally with four lanes in each direction. Each expressway is 100 mile long and it is partitioned into 100 segments, each with 1 mile-long, as illustrated in Figure 3.

Figure 2- An Example of an expressway segment

The input data is generated by MIT’s Microscopic Traffic Simulator (MITSIM) [10]. The generated data is stored in two separate files, one for the input data containing information of vehicles traveling on the expressway and the other one with historical data about the vehicles. A vehicle starts its journey from an entry ramp and ends its journey on an exit ramp.
except from the journeys that are uncompleted at the end of the simulation. Each vehicle emits a position report every 30 seconds. MITSIM ensures that a vehicle moves with a maximum speed of 100 Mph so that it would have at least one position report for each segment it travels on.

There is 1% probability ensured by the simulator that a report accompanied with historical query is requested. Half of these requests are queries for account balance, 40% are travel time prediction queries, and the rest of the 10% are daily expenditure queries. There are 6,215,088 daily expenditure queries in the input file for $L=512$. In this thesis we are mainly interested in calculating the maximum response time (MRT) of each daily expenditure query, which is the most time critical queries accessing historical data. For each daily expenditure query a request to the relational database is made and with the intensity of the queries at the final minutes of the simulation the implementation will not be able to keep within the limitations of LRB. The other queries are implemented in SCSQ and computed based on SCSQs built-in main memory database, whereas the daily expenditure queries execute by accessing an external relational database storing the historical data. This historical data is very large in size so it cannot be stored in the main memory; hence the historical data must be stored a relational database.

2.2.1 Query requirements

Aside from the two types of continuous queries which have to be implemented, toll processing (toll notification and toll assessment) and accident processing (accident detection and accident notification), there are three types of historical queries which have to be implemented as well, namely account balance, travel time estimation, and daily expenditure queries. Daily expenditure queries are the main focus of this thesis, since they are the most frequent among historical queries.

A daily expenditure query requests the total amount of toll spent on a certain expressway for a certain day in the last 10 weeks. The result must be delivered within 10 seconds. For the implementation to be able to respond to the daily expenditure queries it must maintain 10 weeks’ worth of toll data per vehicle and expressway. Taking into account that the data will be simulated for approximately 135,000 vehicles for 3 hours, the equivalent historical data for this amount of vehicles in 69 days for $L=1$ will result roughly around $69 \times 135,000 = 9,315,000$ of rows in the (MySQL) relational database.

As described previously our main focus is to parallelize such daily expenditure queries.
A historical query for a given vehicle is triggered by a request tuple: 
\[ d = (\text{Type}, \text{Time}, \text{VID}, \text{QID}, \text{XWay}, \text{Day}) \]

Where Type is the query type indicator, which is, e.g., 3 for daily expenditure queries, Time is the time when the query was emitted, VID is the vehicle ID, QID is the query ID, Xway is the express way number, and Day is the day when the query is issued. The implementation of L has to satisfy maximum response time requirements and the result of the continuous and historical queries must be correct. For example, the maximum response time allowed for daily expenditure queries is 10 seconds.

### 2.3 SCSQ

SCSQ is a DSMS prototype extending Amos II, a main memory and functional DBMS [13]. SCSQ has a highly flexible approach to parallelization by providing three kinds of **parallelization functions** [16], as illustrated in Figure 4. The critical parallelization function for very high volume data stream is `splitstream()` that splits a high volume stream to a possibly large number of small streams on which the `mapstream()` function is applied to do the often expensive computations over sub-streams in parallel. In the case of LRB the work to be done is the execution of the stored procedure `lr()`, which runs the CQs on each parallel node to produce one result stream per expressway. The result streams are merged through the `mergestream()` function.

SCSQ has been shown in [17] to be able to achieve an L-rating of 512, but without answering daily expenditure (DE) queries, since the relational database accesses to the historical database became a bottleneck. The present

<table>
<thead>
<tr>
<th>Trigger</th>
<th>Daily Expenditure request, ( d )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Condition</td>
<td>-</td>
</tr>
<tr>
<td>Output</td>
<td>((\text{Type}: 3, \text{Time}: t, \text{Emit}: t', \text{QID}: q, \text{Bal}: \sum p \in \text{tollset}(v), f(p))\ s.t.\ Day(p.\text{Time}) = d, p.\text{XWay} = x)\</td>
</tr>
<tr>
<td>Recipients</td>
<td>( v )</td>
</tr>
<tr>
<td>Response</td>
<td>( t' - t \leq 10 \text{ sec} )</td>
</tr>
</tbody>
</table>

*Figure 3-Daily Expenditure Requirements [3]*

\[ f(p) = \text{Toll} (\text{M}(p.\text{Time}, p.\text{XWay}, p.\text{Seg}, p.\text{Dir})) \]
work investigates the performance of a SCSQ implementation of LRB, including DE queries, based on scaling out the relational (MySQL) database storing historical data to parallel nodes. Since the focus of this thesis is to investigate the scaling out of relational database access of SCSQ which is done by executing DE queries, implementing the estimated travel time queries was not considered.

![Figure 4-SCSQ functions splitstream, mapstreams, and mergestreams](image)

2.3.1. SCSQ Architecture

![Figure 5-SCSQ architecture for L=w](image)

To achieve the desired response time, SCSQ executes \( L \) parallel instances of LRB query processing, each processing queries for one expressway. The LRB query processing is implemented by the stored procedure \( Lr() \) that processes position reports, account balance queries and DE queries. In the first part of this thesis \( Lr() \) only the maximum \( L \) rating for DE queries is investigated, since that was the bottleneck which was excluded in [17]. For example, to run LRB for \( L = 64 \), 64 parallel processes of \( Lr() \) will be executed each accessing some local database node in a scaled-out MySQL for an ex-
pressway. This architecture is shown in Figure 5. In the second architecture the \( Lr() \) is implemented with only estimated arrival time queries being excluded.

2.3.2. SCSQ Parallelization Functions

Scalable execution in applications like LRB requires parallelization. The parallelization functions introduced in SCSQ are used to split an input stream into large collections of parallel streams. Each parallel stream will produce collections of result streams and finally the parallel results are merged together [16]. The main functions that are provided by SCSQ are \( \text{Splitstream()} \), \( \text{Mapstream()} \), \( \text{Mergestream()} \), and \( \text{Zipstreams()} \). The use of these functions in this thesis is described with more detail in Section 4.

\( \text{Splitstream()} \) splits one input stream \( S \) into \( w \) parallel streams by partitioning and/or replicating input streams into a collection of streams. The basic signature of \( \text{splitstream()} \) is:

\[
\text{splitstream(Stream s, Integer w, Function rfn, Function bfn) -> Vector Of Stream sv}
\]

The first functional argument \( rfn() \) is the \textit{routing function}, having signature \( rfn(Object tpl, Integer w) -> Integer \), which returns the output stream index (between 0 and \( w - 1 \) ) in the vector of output streams \( sv \) for each tuple in the input stream \( s \) that should be routed to that output stream. The functional argument \( bfn(Object tpl) -> Boolean \) is the \textit{broadcast function}, which returns \textit{true} for tuples to be broadcasted to all output streams in \( sv \).

There is another signature for \( \text{splitstream()} \) which uses \( sfn() \) instead of \( bfn() \) and \( rfn() \). This function is a pure \textit{split function} which only indicates the output stream index (between 0 and \( w - 1 \) ). This function returns a Stream instead of returning Vector of Stream.

\[
\text{splitstream(Stream s, Integer w, Function sfn) -> Stream}
\]

\( \text{Mapstreams()} \) applies a mapping function, \( \text{mapfn()} \), over each stream in a vector of streams \( sv \), which results in a new vector of streams, \( sw \). The basic signature of \( \text{mapstream()} \) is:

\[
\text{mapstreams(Vector of Stream sv, Function mapfn) -> Vector of Stream sw}
\]
**Mergestreams()** merges a collection of streams into one output stream by doing a union-all of the tuples of its input streams. The union-all is done in a first come-first serve asynchronous manor, i.e. the union will not wait for all the participants to emit a tuple. The signature of *mergestreams()* is:

```
mergestreams(Vector of Stream vs) -> Stream
```

**Zipstreams()** operates like *mergestreams()*), but in difference to *mergestreams()* it does a synchronized union-all. All processes participating in the zip have to emit a tuple in order for the union-all to happen. The signature of *zipstreams()* is:

```
zipstreams(Vector of Stream vs) -> Stream
```
3. System Architecture

In a previous implementation [6] of SCSQ-MySQL a single MySQL instance was used that interacted with SCSQ through a JDBC connection. The whole LRB logic was implemented as a stored procedure in MySQL. By running the SCSQ-MySQL implementation on a single node with one MySQL instance, $L=0.5$ was achieved. By contrast, in this thesis MySQL is used to store only historical data to and is accessed from SCSQ nodes for executing DE queries. SCSQ will handle the CQs. The dynamic LR data for each expressway is stored in the local main-memory database inside each SCSQ node.

In the SCSQ-MySQL implementation all the results to different queries were inserted into a temporary table. After a certain amount of time the results were fetched from the table and the table was flushed. In this thesis a different approach is used. The results are not saved into temporary tables because the overhead of accessing temporary tables is not efficient, instead they are emitted immediately to the output stream to be processed by downstream DSMS nodes.

Since the amount of historical data generated is too big to store in the main memory it has to be stored on disk, which makes the access to the rela-
tional database the bottleneck of the system. In order to overcome this short-
coming the MySQL instances are scaled out to parallel nodes for groups of
expressways. Each input stream will be split into $L$ streams based on its
Xway, each having access to a corresponding MySQL database for a group
of SCSQ nodes.

As shown in Figure 6 there are two types of nodes in this architecture, the
stream processing nodes $P_Q$, and the database processing nodes $D_P$. The
former execute all the implemented queries in $Lr()$ except from the DE que-
ries. The number of LR processing nodes is equal to the number of express
ways. The second type of nodes in the architecture will receive the DE
events from the LR processing nodes, execute the DE query on the relational
database, and finally ship back the result.

The databases storing the historical data are split into the same number of
partitions as the number of database processing nodes. Each LRB event,
after being routed to the corresponding LR processing node by the split func-
tions, will be first checked if it is a DE event. If so it will be shipped to a
database processing node. It will then be executed by the relational database
and finally the response from the relational database will be shipped back to
the corresponding node. The database processing node for a DE will be de-
determined by doing a modulus operation on the Xway of each event and the
number of database processing nodes.
If the incoming event is not a DE query it will be executed on the LR
processing node and the result will be sent downstream.

As described earlier in order to access MySQL from SCSQ a JDBC con-
nection is needed. SCSQ has a Java wrapper called JavaSCSQ which can
provide the JDBC connection to MySQL. Therefore the database processing
nodes accessing MySQL are run using JavaSCSQ.

As it is demonstrated in Figure 6, each JavaSCSQ process can be ac-
cessed by more than one SCSQ processes since each database will store data
for more than one express way and each $Lr()$ processing node will handle
only one expressway.

In order to meet the requirements of LRB, the overall time from where the
events are read from the input to the point where the results are merged for
each event has to take less than 10 seconds. In this project we will examine
with different number of nodes to see how well the MySQL database can be
scaled out in respect to the scaling of SCSQ. The results for the experiments
are presented in Section 6.
4. Implementation

In this section the implementation of parallelizing DE queries will be described.

4.1 Simulated SCSQ implementation

After reading the input stream each row that issues a DE query will be looked up to the historical toll data stored in the MySQL database. The table storing the historical data in MySQL called Historical_toll is defined in Table 1.

<table>
<thead>
<tr>
<th>Field</th>
<th>vid, day, xway, toll</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary Key</td>
<td>vid, day, xway</td>
</tr>
</tbody>
</table>

Table 1-Schema of Historical_toll table

As shown above the primary key for the table Historical_toll is the combination of vid, day, and xway, which means that an index with the same combination is created on the table. As described in Section 2.3.1 the toll is calculated according to VID, Xway, and Day of the DE query. Since we have the same combination of fields as the index of the Historical_toll table, this will speed up the lookups in order to calculate the toll balance for the specified values.

For each given DE query in the input stream a look-up to the Historical_toll table will be made and for each query the toll balance matching VID, XWay, and Day in the table will be returned as a result. All the results will be merged as the output stream for DE queries.

There will be $L$ UNIX processes executed each handling one expressway. This is done using the script shown in Figure 8.

```
k=10
for i in `seq 0 255`; do
  j=$(expr $i \/ $k)
  ./Run $i $j &
done
```

Figure 7- Executing simulated SCSQ flow
This script simulates how SCSQ works. Each executed UNIX process represents an instance of \( L_x() \) executed by SCSQ and will operate as one stream processing node. Each processes will read the input of the expressway it is responsible for and using prepared SQL statements make a call to the database. The input stream will be split according to its \( X_{way} \) number; consequently each process will be routed to the database that is holding the data for the processed expressway.

The input arguments for this implementation represent \( X_{way} \) and database ID respectively. As it can be derived from the script only 10 processes access one database i.e. each database will have at most 10 connections to it. This will be described more in Section 6.

Until now the results of all processes will be \( w \) number of stream processes which each of them connect to a MySQL instance through a TCP connection that returns the result of a DE query. In this state all the processes will send their output as one output stream.

Thereafter the approach described in the previous step is implemented in SCSQ.

### 4.2 Naïve SCSQ implementation

As the first step of implementing LRB in SCSQ with DE queries, a naïve approach is chosen. The plan for this approach can be seen in Figure 9 and the code for this implementation is represented in Appendix A.

![Figure 8-Naive SCSQ implementation](image-url)
Illustrated in Figure 9 is the data flow for the naïve implementation of LRB in SCSQ for L-rating of 128. LRB is designed so that the number of inputs for each simulated minute increases where the intensity of inputs is the highest at the end of simulation. Therefore the values expressed in Figure 9 are from the execution of an LRB implementation in SCSQ for the last most intensive eleven minutes. As shown in the figure above the elapsed time for the last eleven minutes is 2116 seconds which is more than the expected time. This is because of the naïve approach where most the time is spent on reading and splitting of the input stream in Sp1 and Sp2. The processes Sp3-Sp130 are responsible for the execution of four CQs, including the DE queries. Each Sp is responsible for one expressway. The response of each query in sent to one of the four writing processes, which are responsible for the writing of each kind of query results to a file. Observations show that writing outputs for position reports and account balance takes the most time. In order to improve the results two main steps have to be made to optimize the implementation: i) optimization of reading the input stream and ii) optimization in the result writing.

4.3 Optimized SCSQ implementation

In this section the two main optimizations made in order to get improved results are described.

4.3.1 Optimization of the reading process

According to Figure 9 the CPU utilization, the percentage of CPU time over the total wall time, is at its peak in the reading and splitting SCSQ process. By studying the log files of SCSQ it is shown that most of the time, 85%, is spent in Sp1 when it is in sending data to Sp2. 36% of the time is spent in Sp2 on reading from Sp1. The majority of the remaining time, 35%, is spent on splitting the data between Sp3-Sp130. The solution for improving the performance has two parts: i) lowering the communication time and ii) decreasing the splitting time. The first part is done by packing the elements in the input stream into binary arrays of multiple tuples, called multiarrays. The na2multi() function of SCSQ handles this part as shown in the optimized code in Appendix B. With this solution instead of sending one packet for each input, a compressed packet containing many binary input tuples will be sent to Sp2 at the time.

As illustrated by Figure 9, in order to decrease the splitting time consumed the approach is to change the splitter to a window splitter [17]. The goal is to distribute incoming multiarrays to several second level splitters Sp3-Sp12 that unpack the multiarrays received from the window splitter. The unpacked tuples in the multiarrays are the sent to the processing nodes Sp13-
Based on the query type and the Xway. In Figure 10 the optimized architecture is represented for the reading and writing processes. The writing process will be further described in 4.2.2.

4.3.2 Optimization of the writing process

Based on Figure 8 the most amount of time spent for the writing is for the position reports and account balance. This is due to firstly that position reports are the most frequent queries, 99%; therefore the results generated create the most output tuples, consequently taking more time to produce. Secondly the naïve mechanism for computing the account balance creates so much delays that the system spends more time than position report queries although the frequency of account balance queries, 0.5%, are much less. In the naïve mechanism to compute the account balance for each vehicle, Sp132 in Figure 8 has to accumulate the balance of the vehicle in each expressway by polling from each process, Sp3-Sp130, and make a synchronized merge at the end with the zipstreams() function in SCSQ.

The optimized solution is to split the merge/zip for position reports and account balance processes to a two layered merge/zip. For the position reports the two layered merge is for splitting the load into more nodes so it can
be done in a faster manner. On the other hand the polling for the account balance queries is done in two steps so the delay is reduced. In the first step the number of processes responsible for the polling is equal to the value $\sqrt{Xway}$, where each will poll from the fraction of $Sp13-Sp140$. For example as illustrated in Figure-9 the polling processes $Sp150-Sp160$ each are responsible for 11 r processes to poll. In the next layer $Sp150-Sp160$ are polled by $Sp162$. As a result the query response for the vehicle will be sent to the output stream.

Figure 9 illustrates that the optimization applied to the implementation has had a great impact by reducing the total elapsed time from 2116 seconds in the naïve implementation to 539 seconds for $L=128$. Also the CPU utilization value for each process shows that most time spent in the system is in the computation nodes, $Sp13-Sp140$, which is the ideal situation. The code for the optimized implementation is in Appendix B.

As described in Section 3, in order to execute DE queries an access to the relational database is needed, which is made by separate JavaSCSQ instances accessing the relational database. Figure 10 illustrates this based on the optimized architecture in Figure 9.

![Figure 10: Optimized SCSCQ implementation including DBMS access](image-url)
The processes responsible for accessing the relational databases should be started before starting to run the whole system. This is done as below.

```
{{ port = $(./../../get_nsport.sh) }}
..../bin/scsq.exe lrdexp.dmp -o "nameserverport($port);" -l
"(trace server-eval)" \
  -ns > ScsqTestLog 2>&1 &
for i in `seq 0 12`; do
  (javascsq lrdexp.dmp -O peer.osql -o "connect_jdbc($i);" -s P$i &)
done
..../bin/scsq.exe lrdexp.dmp -o "nameserverport($port); register('c1');" -O experiments.osql
```

In the script above first a name server as the coordinator will start. Then JavaSCSQ peers will start as $P_i$ and finally the client will run as a SCSQ process by registering the name server “C1” to create the federation.
5. Input Data Analysis

The input is a stream file containing an event in each line which is read by SCSQ. The historical data generated is over 2,211,840,000 rows in a MySQL table with the maximum VID of 34,560,000 for L=256. The maximum VID is roughly equal to 135K * L, consequently the amount of data in the MySQL table will be equal to Maximum VID * 69. This amount of data is equal to more than 50GB of data on disk.

This input file consists of randomly assigned xways for each event. Therefore for L=256 and considering that the xways are assigned randomly, the probability of an event occurring in the input file having a response from the database will be 1/256. Consequently the total number of responses i.e. the number of rows in the output stream will be equal to the number of input events divided by L. For L=256 we have 3,107,544 input events, so theoretically we should have 3,107,544/256 = 12,139 output events as response. By executing the implementation described in Section 4 for L=256, 11,796 responses were produced during the experiments, which are close enough to the amount expected.

The input files for the experiments in this thesis only contain events for L=2 and a multiplier function called clrmultiply() will be used in order to generate the inputs for larger L. The function clrmultiply() multiplies the existing data by a given number and creates the intended input data according to the variables given. An example of the result of an event produced by clrmultiply() can be seen in Figure 12 where the single input event for L=64 has been multiplied by eight resulting inputs for L=512. The above result is

\[
\{(3,17,1,-1,1,-1,-1,-1,5,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,1,-1,1,-1,-1,-1,5,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,140001,-1,65,-1,-1,-1,-1,125005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,280001,-1,129,-1,-1,-1,-1,250005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,420001,-1,193,-1,-1,-1,-1,375005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,560001,-1,257,-1,-1,-1,-1,500005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,700001,-1,321,-1,-1,-1,-1,625005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,840001,-1,385,-1,-1,-1,-1,750005,-1,-1,1334669273,679694,10)\}
\]

\[
\{(3,17,980001,-1,449,-1,-1,-1,-1,875005,-1,-1,1334669273,679694,10)\}
\]

*Figure 11-Input event produced by clrmultiply()*

populated based on the formula bellow where \(i\) is the coefficient value and t1-t15 are the base tuple attributes.
({ t1, t2, i*140000+t3, t4, i*64+t5, t6, t7, t8, t9, i*125000+t10, t11, t12, t13, t14, t15 })
6. Experiments

The results of different runs of the implemented code for parallelizing the daily expenditure queries over parallel nodes are presented in this section. In Section 6.1 the experimental setup is described. In Section 6.2 the results for simulated executions are presented. In Section 6.3 the results for the implemented solution in SCSQ including daily expenditure queries are shown and the performance of the system is observed.

6.1 Experimental setup

All the experiments have been run on one NUMA machine with 8 CPUs each with 8 cores. This machine has 1Terra Byte of memory. There are 10 hard drives available on this machine. More detailed description can be found in the table below:

<table>
<thead>
<tr>
<th>Processor</th>
<th>AMD Opteron(TM) Processor 6276</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU MHz</td>
<td>1400</td>
</tr>
<tr>
<td>CPU cores</td>
<td>8</td>
</tr>
<tr>
<td>Number of CPU</td>
<td>8</td>
</tr>
<tr>
<td>Total Memory</td>
<td>132124580 kB</td>
</tr>
</tbody>
</table>

*Table 2: Detail Description of Experiment Machine*

In the SCSQ-plr [16] implementation, in order to achieve an efficient parallelization the input stream is split into \( L \) parallel streams, and for each expressway one instance of \( Lr() \) is executed. Based on this architecture in this thesis the approach is to first execute \( L \) processes where each process handles only one expressway as explained in Section 4 and then compute the \( L \)-rating for running LRB in SCSQ including the DE queries.

A data generator was used for generating the historical data and the generated data was into the scaled out MySQL instances. The code for the generator can be accessed in the Appendix C.
For experimental reasons it is best to set the MySQL query cache value to zero, so every time that we run the queries we know that the results are not cached by MySQL. This is done as below:

```sql
mysql> set global query_cache_size = 0;
mysql> SHOW VARIABLES LIKE 'query_cache_size';
+------------------+
| Variable_name    | Value |
+------------------+
| query_cache_size | 0     |
+------------------+
```

An OS caching also exists that cannot be turned off; this has been described in Section 2.

The rest of this section contains some instructions on how to start multiple MySQL Instances in Section 6.1.1, and in Section 6.1.2 two experiments are described in order to reach the best configuration for running the tests. In the first experiment the number of database partitions is static and equal to one, but the number of connections differs from one to 60. In the second experiment the number of connections to the database is kept equal to one and the number of database partitions varies between one and 52.

6.1.1 Starting multiple MySQL Instances on one machine

It is necessary to explain how to start multiple MySQL Instances on one machine.

For starting multiple MySQL instances there are four simple steps:

1. Modify the `my.cnf` file as shown in Appendix E and place it in the home root.
2. Install MySQL default tables on each data directory mentioned in `my.cnf`
   ```bash
   ./scripts/mysql_install_db -User=mysql --datadir="Path_to_Datadir"
   ```
3. Start multiple MySQL instances based on `my.cnf`:
   ```bash
   Mysqld_multi - default -file="Path_to_My.cnf" start
   ```
4. Check the status of the instances:
   ```bash
   Mysqld_multi - default -file="Path_to_My.cnf" report
   ```

It is a requirement to set the data directory of each MySQL instance in different directories as it is advised by Oracle.
6.1.2 Experimental initializations

In order to investigate if a MySQL server can handle more than one instance on each hard drive an experiment was conducted where a sample input of 200,000 queries was executed on approximately 600 million data in all the database instances. These input queries are generated based on the LRB input queries for DE queries as described in Section 2.2. These queries are samples of all the DE queries for $L=64$, hence the historical data is the complete data set for $L=64$.

Since there are only 10 hard disks available on the NUMA machine where these experiments were conducted, furthermore based on Chart 1 which is almost a straight line, the chart above indicates that having five MySQL instances running on each hard drive is favorable.
While running different experiments it was realized that the number of connections to each MySQL instance can also influence the results. Therefore an experiment was made to investigate how increasing the number of connections to a MySQL instance from one to 64 connections would influence the results. In this experiment the same sample input queries used in the previous experiment were run on one database with $1 \rightarrow 64$ connections to it with the approach shown in Figure 12.

![Figure 12a: 1 connection to database b) 4 connections to database](image)

The results of this experiment is presented in Chart 2 showing that having more than 10 connections to one database is not efficient.

![Chart 2: Number of connections to one Database](image)

The base line presented in Chart 2 is the best condition that the processes can have, meaning that the number of connections to the database wouldn’t
have any effect on the results. By contrast, when the number of connections exceeds 10 the total time for querying the inputs increases rapidly. These measurements were done using connections to MySQL through a C program using TCP connections. This experiment shows that the bottleneck of accessing a relational database from a data stream is mainly because of the number of connections to a relational database which can be solved by scaling out this access.

6.1.3 Initialization conclusion
The two experiments described in 6.1.2 conclude the setup configurations for future experiments. In order to achieve the best performance, the experiments must be designed in a manor where there are not more than 10 connections to any MySQL instance and five MySQL instances can be started on each hard drive. The connections to MySQL must be made through TCP connections.

6.2 Results for simulated execution
In this section the results to experiments for the simulated implementation described in Section 4 is presented.

Experiments have been performed on LRB for \( L = 64 \) to 512 based on the configurations described in 6.1. The results for the experiments are shown in Chart 3.

![Chart 3, MRT for L = 64 to 512 Cold Run](chart3.png)
As shown in Chart 3 the results for MRT regarding $L = 64 - 512$ is a linear trend, based on the estimation presented, the MRT will exceed the LRB limitation of 10 seconds approximately on $L = 600$.

6.3 Results for SCSQ implementation

In this section the results for the LRB implementation including the daily expenditure queries in SCSQ will be presented. The input file consists of the last eleven minutes of the LRB where the intensity of the inputs is at its highest. The goal is to finish the last eleven minutes in less than 660 seconds so that the response time for different queries meet the LRB limitations, Table 2.

<table>
<thead>
<tr>
<th>Query</th>
<th>Maximum response time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Daily expenditure</td>
<td>10 seconds</td>
</tr>
<tr>
<td>Account balance</td>
<td>5 seconds</td>
</tr>
<tr>
<td>Accident report</td>
<td>5 seconds</td>
</tr>
<tr>
<td>Position report</td>
<td>5 seconds</td>
</tr>
</tbody>
</table>

Table 3 - LRB response time limits

The implementation of SCSQ in this thesis is only focused on the last eleven minutes of the whole 3 hour simulation because the input is the most intense in the final minutes. Having passed the limitations for that period of time will confirm reaching the goal of this thesis which was to successfully scale out the access to a relational database from SCSQ.

The results for this experiment are shown in Chart 4, the response time for the position report queries are not shown in this chart since all the values for different number of express ways were less than 2 seconds.
The response times calculated by running the SCSQ implementation of LRB and the total time of execution can be tuned by the number of window splitters, \( Sp_3 \) to \( Sp_{13} \) in Figure 10.

By increasing the number of window splitters to some extent the response times go higher and the total execution time decreases. This is because by increasing the window splitters the volume of input will increase causing the total to decrease with cost of making the computing nodes more busy consequently increasing the response times. The values for different number of expressways with different number of window splitters can be seen in Table 3.

<table>
<thead>
<tr>
<th>L</th>
<th>WS</th>
<th>Total Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>3</td>
<td>312</td>
</tr>
<tr>
<td>128</td>
<td>10</td>
<td>555</td>
</tr>
<tr>
<td>150</td>
<td>10</td>
<td>695</td>
</tr>
<tr>
<td>150</td>
<td>15</td>
<td>651</td>
</tr>
<tr>
<td>154</td>
<td>15</td>
<td>691</td>
</tr>
</tbody>
</table>

Table 4—Various express ways with various window splitters

Based on the results, although the response times for \( L=154 \) is below the limits for LRB but since the total execution time is more than 660 seconds the \( L \)-rating admissible for this implementation is \( L=150 \).
7. Related Work

There have been a number of implementations of LRB, IBM Stream Processing Core [8], Aurora [1], XQuery [4], DataCell [9] and SCSQ-LR [14], SCSQ-MySQL [6] and SCSQ-PLR [17]. The results for different implementations of LRB are presented in the table below.

<table>
<thead>
<tr>
<th>LRB Implementation</th>
<th>Maximum L Achieved</th>
<th>Number of Cores</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM SPC</td>
<td>2.5</td>
<td>170</td>
</tr>
<tr>
<td>Aurora</td>
<td>2.5</td>
<td>1</td>
</tr>
<tr>
<td>XQuery</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>SCSQ-LR</td>
<td>1.5</td>
<td>1</td>
</tr>
<tr>
<td>SCSQ-PLR</td>
<td>64 512(D type queries disabled)</td>
<td>48 560</td>
</tr>
<tr>
<td>SCSQ-MySQL</td>
<td>0.5</td>
<td>1</td>
</tr>
<tr>
<td>SCSQ-MySQL Scaled Out</td>
<td>150</td>
<td>64</td>
</tr>
</tbody>
</table>

*Table 5: Results For Different LRB Implementations*

As shown in the table above no previous implementation of LRB was able to reach SCSQ’s $L=150$ with the DE queries enabled. This thesis shows how scaling out the MySQL access makes it possible to achieve approximately $L = 600$ if only the DE queries are executed. Also it was shown that the LRB implemented in SCSQ with daily expenditure queries enabled on one NUMA machine with 64 cores can achieve $L=150$.

The experiments in [17] verified that a conventional database on a single compute node was sufficient to handle queries over historical data (event type D) for an L-rating up to 64. Because the conventional DBMS (MySQL) could not handle the very high query rates the daily expenditure queries were ignored in [17] which resulted in achieving $L=512$. The approach used in this thesis eliminates this bottleneck of accessing the historical data stored on the disk by scaling out the database and accessing each partition based on the configurations discussed earlier.
In this thesis it is shown that scaling out the relational database will meet the LRB requirements for $L = 600$ of executing DE queries and 150 for the LRB implemented in SCSQ.
8. Conclusions and future developments

While this thesis studied the access of a DSMS using the LRB benchmark it can be concluded that in order to implement the daily expenditure queries it is best to use a partitioned database with the configurations described in Section 6. This thesis also showed that in case of more limitations a DSMS access to a relational database can be successfully scaled out, which will have more improvements in the overall results.

The main experiments were made based on the SCSQ architecture as described in Section 4. The SCSQ implementation of LRB were run for the last eleven minutes of the whole 3 hour simulation due to the intensity of the inputs for that period with DE queries enabled, reached the $L$-rating of 150 on one machine with 64 cores.

A future study on this implementation could be to investigate and compare the results from this implementation to other scaling out implementations of MySQL, for instance MySQL Cluster. MySQL cluster is an alternative for scaling out DBMSs. In the new release of the MySQL cluster a 70X faster reads have been encountered for joined tables, leading to delivering more than 1Billion queries per minute [11]. MySQL Cluster can be used in the implementation of LRB for scaling out and parallelizing the DBMS access. As [17] shows, SCSQ gives a network bound maximum throughput which and the performance is not CPU bound. Therefore MySQL Cluster, in the best case, would have the same performance compared to SCSQ. This comparison can give a better insight on the performance of SCSQ. Implementing LRB with map-reduce techniques can also be studied.

Additionally implementing the remaining estimated travel time queries and also running the full 3 hour simulation for the current implementation of the LRB is open for future study.

In conclusion it has been shown how scaling out a relational database access from SCSQ can achieve $L=150$ with daily expenditure queries enabled. Although the experiments have been executed for the last eleven minutes they are regarded as acceptable since the intensity of database accesses are at their peak in this period of time and since the implementation can fulfill the LRB limitations for this period they also can fulfill the limitations for the whole 3 hours.
9. Acknowledgments

I would like to thank my supervisors Tore Risch, Sobhan Badiozamany, and Erik Zeitler for their contribution and valuable advice. Sponsored by, EU FP7 project Smart Vortex, and the Swedish Foundation for Strategic Research under contract RIT08-004.

References

Appendix

A. Naïve implementation in SCSQ

```plaintext
create function DexpNaive(charstring infile, integer inl, integer mult, charstring outfile) -> object as
select in( mergestreams( {t3, t2, t1, t0} ) )
from stream of iarray input, integer fo,
vector of stream xp, vector of stream proc,
stream t0, stream t1, stream t2, stream t3
where t3 = ostream(newspflex( streamof(writefiles(outfile + "-ib-3", 10000000,
in(mergestreams(vectorize(select i, proc[i*4 + 3])
from integer i where i in iota(0, dim(proc)/4))))),
make_record({"hostname", 'localhost'}), 0)
and t2 = ostream(newspflex(streamof(writefiles(outfile + "-ib-2-
"", 10000000, dtslrbin(lrf0(in(zipstreams(vectorize(select i, proc[i*4 + 2])
from integer i where i in iota(0, dim(proc)/4))))),
make_record({"hostname", 'localhost'}), 0)
and t1 = ostream(newspflex(streamof(writefiles(outfile + "-ib-1-
"", 10000000, dtslrbin(in_d(mergestreams(vectorize(select i, proc[i*4 + 1]
from integer i where i in iota(0, dim(proc)/4))))),
make_record({"hostname", 'localhost'}), 0)
and t0 = ostream(newspflex(streamof(writefiles(outfile + "-ib-0-
"", 10000000, dtslrbin(in_d(mergestreams(vectorize(select i, proc[i*4]
from integer i where i in iota(0, dim(proc)/4))))),
make_record({"hostname", 'localhost'}), 0)
```
B. Optimized implementation in SCSQ

```c
25  and proc = mapflex(xp, #'lrdepbins', make_record({'imagesize',
26                                           200000000, 'nsub', 4, 'sfn',#'dexpesplit'}))
27  and xp = splitstream(input, fo, '#xwayrfn', '#q2bfn')
28  and input = ostream(newsp(streamof(clrmultiply(
29                                           in(lreadfastbin(infile)),inl,mult))),0)
30  and fo = inl*mult;
```

```c
1  create function DexpOptimum(  
2   charstring infile, integer inl, integer mult, integer rsplitfo,  
3   integer postmsz, charstring outfile)
4  -> object as
5  select in(res) from stream of iarray src, vector of stream of  
6   iarray rsplit, integer proc, vector of stream fsplit, vector of  
7   stream of darray proc, stream of darray res, stream t0,  
8   stream t1, stream t2, vector of stream t20, vector of stream t00  
9   where res = mergestreams((t3, t2, t1, t0))
10  and t3 = ostream(newsp(streamof(writefile(outfile + '-dexp',
11                                           in d(mergestreams(vectorize(select i, proc[i*4 + 3],  
12                                           from integer i where i in iota(0, dim(proc)/4)))))),0)
13  and t2 = ostream(newsp(streamof(writefile(outfile+'-accbalance',
14                                           in d(zipmaplayer(t20, 1, 6, '#'lrfd')[0]))),0)
15  and t20 = zipmaplayerm(vectorize(select i, proc[i*4 + 2],
16                                           from integer i where i in iota(0, dim(proc)/4)),
17                                           integer(sqrt(fo)), 6, '#'lrfd', 6)
18  and t1 = ostream(newsp(streamof(writefile(outfile+'-accident',
19                                           in d(mergestreams(vectorize(select i, proc[i*4 + 1],  
20                                           from integer i where i in iota(0, dim(proc)/4)))))),0)
21  and t0 = ostream(newsp(streamof(writefiles(outfile+'-position',
22                                           100000000, in d(mergestreams(t00)))),0)
23  and t00 = mergetree(vectorize(select i, proc[i*4],
24                                           from integer i where i in iota(0, dim(proc)/4)), 6)
25  and proc = mapflex(premerge(fsplit, rsplitfo, fo), #'lrdepbins',
26                               make_record({'imagesize', 200000000, 'nsub', 4, 'sfn',
27                               #'dexpesplit', 'na2m', ('ma-size': postmsz)}))
28  and fsplit = unpackfsplit(rsplit, fo, '#xwayrfn', '#q2bfn', 15)
29  and rsplit = rsplit_force_flex(src, rsplitfo, {'hostnum':0,  
30                               'rem':#'rsplit'})
31  and src = ostream(newspflex(streamof(na2multi(clrmultiply(
32                               ts_iarray(in(lreadfastbin(infile)),12),
33                               inl, mult),15)),{'hostnum':0, 'rem':#'src'}), 0)
34  and fo = inl*mult;
```
C. Historical data generator

```java
public class History{

    public static void main(String[] args) {
        int num_threads = Integer.parseInt(args[0]);
        String Xway = args[1];
        String maxXway = args[2];
        String Direction = args[3];
        Random Generator = new Random();
        for(int i = 0; i < num_threads; i++) {
            try {
                FileWriter fout = new FileWriter("History_toll64_"+i+".txt");
                String res="";
                for (int vid=i+1; vid<=Integer.parseInt(maxXway);
                     vid=vid+num_threads){
                    for (int day = 1; day <= 69; day++) {
                        int toll = Generator.nextInt(100);
                        int Xway1 = Generator.nextInt(100);
                        res=vid+","+day+","+Xway1+","+toll;
                        fout.write(res+"\n");
                    }
                fout.close();
            } catch (Exception e) {
                    System.err.println("Exception: " + e.getMessage());
                }
            }
        }
    }
```
D. Functions Used in SCSQ

```plaintext
create function zipmmaplayer(Vector of Stream src, Integer nout, 
    Integer nasize, Function fn) -> Vector of Stream 
    as select mapstreams(strems, fn) 
        from Vector of Vector of Stream chopped, Vector of Stream streams 
            where streams = vectorize(select i, zipstreamsm(v, nasize) 
                from Vector of Stream v, Integer i 
                where v = chopped[i]) 
    and chopped = chopvector(src, nout);

create function zipmmaplayerm(Vector of Stream src, Integer nout, 
    Integer nasize, Function fn, Integer masize) -> Vector of Stream 
    as select mapflex(strems, fn, {'na2m': {'ma-size': masize}}) 
        from Vector of Vector of Stream chopped, Vector of Stream streams 
            where streams = vectorize(select i, zipstreamsm(v, nasize) 
                from Vector of Stream v, Integer i 
                where v = chopped[i]) 
    and chopped = chopvector(src, nout);

create function unpackfsplit(Vector of Stream of numarray si, 
    Integer fo, Function rfn, Function bfn, integer nadim) -> Vector of Stream 
    as vectorize(select i*fo + j, so[j]) 
        from integer i, integer j, Vector of Stream so 
            where so = splitstream(streamof(multi2na(in(si[i]), nadim)), fo, rfn, bfn) 
        and i in iota(0, dim(si) - 1) 
        and j in iota(0, fo - 1));

create function rsplit_force_flex(Stream s, Integer fanout, 
    Record opt) -> Vector of Stream 
    as vselect ostream(p, i) 
        from Sp p, Integer i 
            where p = newspflex(s, merge_records({'nsub': fanout, 'randr': 1}, opt)) 
        and i in iota(0, fanout - 1);

create function premerge(Vector of Stream fsplit, integer rrfo, integer fo) 
    -> vector of Stream 
    vectorize(select i, smjstreamsna(vectorize( 
        select j, fsplit[i + j*fo] 
            from integer j 
                where j in iota(0, rrfo - 1)), 1) 
        from integer i 
            where i in iota(0, fo - 1));
```

38
E. Sample *my.cnf* file

```
[client]
port = 13306
socket = /home/javad/mysqld/1/sock

[mysqld_safe]
socket = /home/javad/mysqld/1/sock
nice = 0
open-files-limit=102400

[mysqld_multi]
mysqld = /local/home/javad/mysql-5.1.34-linux-i686-glibc23/bin/mysqld
mysqladmin = /local/home/javad/mysql-5.1.34-linux-i686-glibc23/bin/mysqladmin
log = /local/home/javad/mysqld/multi.log
user = root

[mysqld]
skip-external-locking

[mysqld1]
port = 13306
socket = /disk1/mysqld/sock
pid-file = /disk1/mysqld/pid
datadir = /disk1/mysqld/data/
basedir = /local/home/javad/mysql-5.1.34-linux-i686-glibc23
server-id = 1
bind-address = localhost
tmpdir = /local/home/javad/mysqld/tmp/
open-files-limit=102400

[mysqld2]
port = 13307
socket = /disk2/mysqld/sock
pid-file = /disk2/mysqld/pid
datadir = /disk2/mysqld/data/
basedir = /local/home/javad/mysql-5.1.34-linux-i686-glibc23
server-id = 2
bind-address = localhost
tmpdir = /local/home/javad/mysqld/tmp/
open-files-limit=102400
```