Otto-von-Guericke-University Magdeburg

School of Computer Science
Department of Technical & Operational Information Systems

Thesis

Streamed verification of a data stream management benchmark

Author:
Stefan Kegel

March 21, 2009

Supervisor:
Prof. Dr. rer. nat. habil. Gunter Saake
Otto-von-Guericke-University Magdeburg
School of Computer Science
P.O. Box 4120, 39016 Magdeburg, Germany

Prof. Tore Risch
Uppsala University
Department of Information Technology
P.O. Box 337, 751 05 Uppsala, Sweden

Kegel, Stefan:
Streamed verification of a data stream management benchmark
Thesis, Otto-von-Guericke-University Magdeburg, 2009.

Contents

List of Figures v

List of Tables vii

List of Abbreviations ix

1 Introduction 1
- 1.1 The Structure . . . 3

2 Background 5
- 2.1 Overview on DSMSs . . . 5
  - 2.1.1 Structure of DSMSs . . . 6
  - 2.1.2 Requirements of DSMSs . . . 7
  - 2.1.3 Categories of DSMSs . . . 8
  - 2.1.4 Examples of DSMSs . . . 8
  - 2.1.5 Stream Windows . . . 9
- 2.2 The AMOS . . . 11
  - 2.2.1 SCSQ . . . 12
- 2.3 The LR benchmark . . . 15

3 Analysis of the validator in PERL 21
- 3.1 Verification Process . . . 23
- 3.2 Validator Structure . . . 24
- 3.3 Performance . . . 26

4 Implementation of the validator in SCSQ 29
- 4.1 Development . . . 30
- 4.2 Further Work . . . 33

5 Conclusion and Future Work 35

Bibliography 37

List of Figures

2.1 Abstract visualisation of a data stream [GO03] 6
2.2 Abstract architecture of a DSMS [GO03] 7
2.3 Sliding window on a data stream [BDM02] 10
2.4 Role of AMOS in the data processing [FRS93] 11
2.5 Abstract architecture of SCSQ [ZR06] 14
2.6 Linear City's expressways [ACG+04] 19
2.7 Structure of one expressway's segment [ACG+04] 19

3.1 Verification process as flow chart 24
3.2 Abstract structure of the validator in PERL 26
3.3 Computation time 28

4.1 Parallelisation of the validation process 29
4.2 File order for the validator 32
4.3 File order for the regression tests 32

List of Tables

3.1 Performance Requirements [Tib03] 22
3.2 Accuracy Requirements [Tib03] 22

List of Abbreviations

ADT	abstract data type
AMOS	Active Mediators Object System
BG	Blue Gene
CNC	compute node coordinator
CQL	Continuous Query Language
DBMS	Database Management System
DSMS	Data Stream Management System
FC	front cluster
FSP	front stream processor
IDS	Intrusion Detection System
Lisp	List Processing Language
LR	Linear Road
Perl	Practical Extraction and Report Language
QC	query coordinator
QM	query manager
SCSQ	Super Computer Stream Query Processor
SP	stream processor
SQL	Structured Query Language
STREAM	Stanford Data Stream Management System
UU	Uppsala Universitet
UDBL	Uppsala DataBase Laboratory

1. Chapter

Introduction

In recent years the progress in computer hardware development lead to a new possibility of processing large amounts of data. Nowadays, those piles of data don't have to be stored to be accessed for further analysis and evaluation, but can be treated as they arrive in time. This brought up the topic of data stream systems which are able to research, aggregate or process over continuous streams of data [Agg06]. Commercial and academic data stream management systems like the *Active Mediators Object System* (AMOS) II [RJ01; Ris07a], Aurora [JACc+03] and the *Stanford Data Stream Management System* (STREAM) [ABB+04] have shown that the synchronous processing of continuously arriving streams of data is feasible, and also ascends a majority of new problems to solve in the context of these systems.

One major problem within data stream systems is the speed with which a data stream management system is able to provide a usable query result out of the content of the streamed data. Hereby a query would for example take a certain amount of stream data and analyse this snippet in regard to given constraints or patterns which are likely to be found within this data stream snippet. This is close to descriptive language queries in relational database management systems which query relations for certain values.

A benchmark to measure the performance of queries on data streams by data stream management systems has been proposed by Arasu, Cherniack, Galvez, et al. [ACG+04]. This *Linear Road* (LR) Benchmark [Tra07] is to be implemented in the according data stream management system and then run through to measure the processing time of this certain system. The LR Benchmark achieves to build up a common basis on which data stream management systems become comparable according to their computation duration and ability of issuing queries on data streams. This is shown by the amount of partaking universities like Brandeis University, Stanford University, The Massachusetts Institute of Technology and Uppsala University. These institutions compare the results of their research on Data Stream Management Systems (DSMSs) by the ability of the according DSMS on the amount of time it does take to simulate one run of the LR benchmark and how many runs can be gone through in parallel processing.

The setting for the LR Benchmark simulation is the flow of car traffic on a city expressway which is parted into lanes and segments which cars pass on their drive inwards or outwards of town. Statistics on the amount of traffic are kept for change on tolls. Passing cars pay those tolls for the use of the expressway according to their number of gone through expressway segments. Another aspect is the possibility of

historical queries issued by cars to gain their account's balance and spent money during past trips. A major facet after the LR Benchmark simulation run is to verify the correctness of the output data (actual result). Hereby a run involves to simulate the traffic on one expressway during a specified duration. This duration is defined by the time segment in the used input and historical data. After the run this input and historical data is utilized to compute an expected result of output data which is compared to the actual result. In the current status quo the actual and expected result is processed by a validator implemented in Perl which carries out a range of *Structured Query Language* (SQL) statements on a relational database. This validation process involves heavy load on the computing hardware as it has to take account of several hundred megabytes or even gigabytes of simulation data. This non streaming sequential task does consume a large amount of hours to process and may even exceed the duration of the original simulation process. This circumstance is a setback within the computation of the LR benchmark. To solve this problem of an extra validation process the *Uppsala DataBase Laboratory* (UDBL) at *Uppsala Universitet* (UU) stated the idea of a data stream validator which runs in parallel to the simulation process. The idea contains to calculate in parallel to the simulation process the expected values on base of the input data. The computed expected values are then compared to the actual output of the simulation run.

This led to the challenge of implementing a streamed verification for the data stream management benchmark LR in the *Super Computer Stream Query Processor* (SCSQ) system. The validator has to control four different kinds of data streams. These are the input data for the simulation, the calculated expected values, the output of the simulation and the data stream of comparison results. To reach this goal, the work involved to learn the SCSQ querying and programming language, to analyse the present implementation of the Perl validator to find out in which matter the steps of the validation process are run through an how to remodel them in a streamed manner, to develop ideas and implement the streamed variant of the validator in SCSQ by utilizing the stream window aggregate functions of Erik Zeitler of UDBL. A downloadable implementation of the LR benchmark in SCSQ can be found on the website of the UDBL [Ris07b].

The overall goal in this thesis is to give a brief view on DSMS with the detail on how to apply the sliding window model [DGI+02; DM06] for the use of a streamed verification of the actual output data of the LR benchmark. This is achieved within the AMOS II [Ris07b; Ris07a] DSMS.

1.1 The Structure

In this thesis, we reach the result of a possible implementation of a validator of the output stream of the LR benchmark which accords to the development at the UDBL at UU. This limits the solution to be implemented in the used AMOS utilizing the SCSQ language for computing expected simulation values and comparing these with the actual output data stream.

In the second chapter, we explain the background on the topic of data stream management systems, the use of sliding windows and SCSQ. We discuss the issues and models of SCSQ in a short and introductory manner and delimit it from other implementations of the LR benchmark.

In the third chapter, we line out the analysis of the Perl validator scripts. This includes the structure of the several scripts working together and a trial of a reverse engineering of the underlying validator model. The handling of the verification process is shown by issues within the scripts and running and preparing them. Also the concepts to be remodelled for the use within the data stream management system SCSQ are explained.

In the fourth chapter, we detail an outline of the implementation in the SCSQ language. The methods which were used to progress development and solve upcoming problems. We tackle how the implementation dealt and interacted with the ongoing further development of the underlying LR benchmark in AMOS II. We also point out at which status development of the validator came to a halt and what problems arose and which challenges remain for future work.

In the fifth chapter, we lay out where to go from the current development status and what possibilities to reshape the existing implementation of the validator could be undergone. Furthermore we line out other ideas of streamed verification implementations and whose ideas may be taken into consideration for the following development. In the last part we will conclude the achieved work and re-evaluate the stated goals.

2. Chapter

Background

In this chapter, we lay out and explain the methods and principles used in the analysis and implementation later on. We will tackle the basic terms of definition as such data stream, DSMS and queries. In short, we describe the basic structure of DSMS and from there substantiate this on the example of the Active Mediators Object System (AMOS) II [Ris07a] by Uppsala Universitet (UU) which has been used to implement the streamed version of the validator. In the last part of this chapter, we state the basic facts and needs for the validation process of the Linear Road (LR) benchmark are detailed.

2.1 Overview on DSMSs

A *Data Stream Management System* (DSMS) utilizes data streams and does make their contents accessible. They allow the user to get a grip on the bypassing data. This is done by the ability to query over certain parts of the actual data stream. The nature of data streams is to be too large for complete coverage within the main memory, their data flow is too fast for saving to hard disk and their content is of a timely limited importance for processing. This implies the need for certain conditions to be met by the architecture and allocated methods of a DSMS to successfully utilize a data stream. After those basic terms, we speak about the different window types which are applied within DSMSs.

A *data stream* [GO03] is a continuous non-controlable ordered sequence of items which arrives in real time. The items are ordered in a timely fashion either by their arrival time or a timestamp of each item. This data stream is modelled as a list of elements (items) which can be characterized as a list of elements themselves, relational tuples or instances of objects. A visualisation of the stream and its items a_n is depicted in Figure 2.1 on Page 6. This figure shows the continuous flow of items during the elapsed time with t_c being the current time.

Stream queries [CDT+00] allow to access the data within these stream elements. These are long-running in the aspect of real time duration. They are of a persistent state within the DSMS and they produce results in a continuous manner.

As results are achieved continuously and in real time, they are not able to provide a view on the data within the whole data stream, only a limited snippet of data can be delivered. Limitations are either by the amount of stream elements processed or by certain time conditions which set data as obsolete by its timely identifier.

Stream queries can be divided into two kinds [GO03]. They are either *monotonic* or

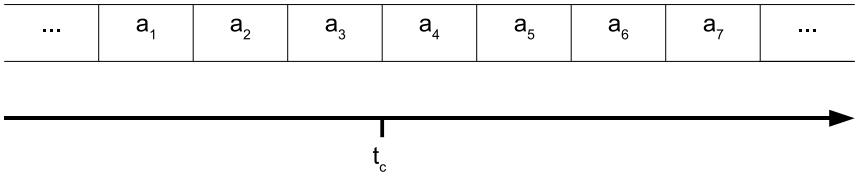

Figure 2.1: Abstract visualisation of a data stream [GO03]

non-monotonic. Monotonic stream queries re-evaluate themselves over every newly arrived stream element and compute an extended result in regard towards the former one. Non-monotonic stream queries re-process themselves from scratch during re-evaluation and thus produce a completely new result.

2.1.1 Structure of DSMSs

The basic structure of DSMSs does follow the procedure of input, processing and output. Only the circumstance that data streams cannot be stored in persistent form and as such not be processed several times after their content's first occurrence implies certain restrictions and requirements on the architecture and behaviour of DSMSs [GO03; CF02].

A layout for the abstract architecture of a DSMS can be seen in Figure 2.2. In it the continuous flow of the data stream's items is depicted by the great arrows which are labelled with *Streaming Input* and *Streaming Output*.

The *Input Monitor* does a pre analyses of the stream's data whose results are utilized by the query processor to optimize query plans and data processing. As a second task the *Input Monitor* surveys that incoming data is well formed or identify in which representation (format) data does arrive that it can be treated properly in the processing step.

In the *storage stage* streaming (dynamic) data and persistent (static) data come together. The storage is parted in two sections. The stream section is construed of the *Working Storage* and the *Summary Storage*. The *Static Storage* consists of the persistent data which is not part of the stream input and as such represents the static section. In the *Working Storage* current stream data is loaded and delivered to the *Summary Storage* whereas the stream data is aggregated according to set limits.

In the *Query Repository* the stream queries of the system's users are safed. These are loaded by the *Query Processor* to be applied on the data from the storage. The *Query Processor* merges the data from the dynamic storage section with the one from the static storage section and handles it according to the requests set by the users' stream queries. It produces query results constantly as an output of stream data which may be re-used as a new input stream.

The output data is cached in the *Output Buffer* to guarantee a continuous flow of streaming data.

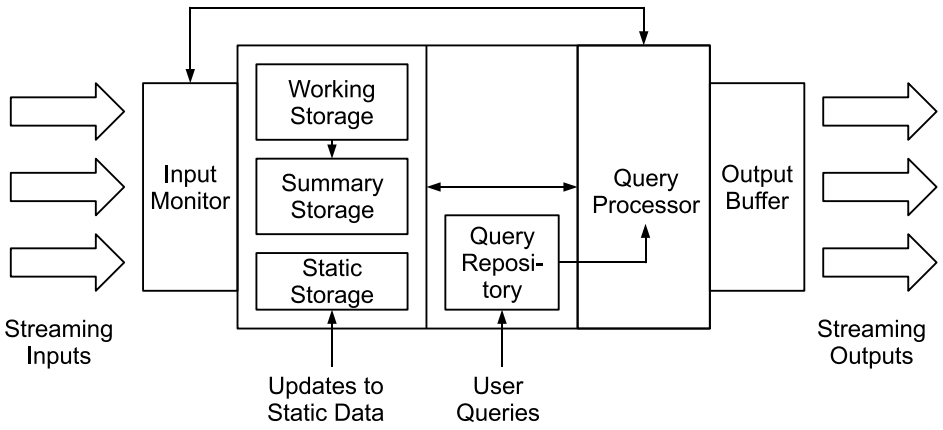

Figure 2.2: Abstract architecture of a DSMS [GO03]

2.1.2 Requirements of DSMSs

To fulfil this process chain certain requirements have to be met by DSMSs [GO03; CF02]. These regard quality of result, operational and system-inherent factors.

- *Quality of result* means that queries executed by the DSMS return a best approximated result in contrast to the real result. This exists due to the non-persistant nature of streaming queries which do not allow to process the complete stream as whole. Taking into account that queries only apply on a digest of the stream, they are not able to deliver a correct answer but only an optimal synopses of the current stream data. It disallows the placement of blocking operators within query plans as the consummation of the whole input for a query execution is not feasible.

- Another requirement is the one-pass of stream processing as it is impossible to backtrack or access past data which has exceeded a certain time limit. This shows that *operational facts* are order and time based by the arrival of stream data. It also implies that the DSMS has to react instantly on unusualities within this arrival as it may corrupt the *quality of result*. Normal changes within the stream data have to be recognized and processed by the system's queries over their life-time, i.e. re-evaluation of the query.

- *System-inherent factors* is the ability of the DSMS to scale on the amount of executed stream queries. It also needs to provide functions to share the processing of stream queries amongst several instances of the DSMS and execute stream queries in parallel on the same or different input streams.

2.1.3 Categories of DSMSs

Data Stream Management Systems (DSMSs) can be categorised by the data model they are following. By the current day three major paradigms on data models are examined.

- These are the *relational* [MWA+03; ABW03a] data model which tries to access and process streams in the same manner as known relational Database Management Systems (DBMSs). In this paradigm DSMS' query languages are similar to the SQL.

  For example the Continuous Query Language (CQL) [ABW03b] which is implemented within the STREAM [ABB+04] allows to access streams and relations in the same manner and interoperate both with each other to produce query results. The CQL works on timestamp ordered windows to process data stream elements. Another example is the StreaQuel Language used in the Telegraph CQ [CCD+03b; CCD+03a; KCC+03]. In comparison to CQL it does not transform stream elements into relations to evaluate them, but does process these directly in a streamed fashion.

- The *object-oriented* data model [GKS01] regards stream elements as objects which would be treated as in an object-oriented DBMSs. Within such DSMSs the stream's elements are ordered in a type hierarchy to access them as objects. Another method is used in the COUGAR DSMS which models the stream's source as an abstract data types (ADTs). Interfaces of these ADTs describe the signature of the source's methods which are used to process the source's data stream.

- In the *procedural* paradigm, the user specifies the data flow by predefined procedures which he arranges to model his query request and get his desired output. In the Aurora DSMS [CcC+02] this is done by the user in the way that he forms a stream query plan out of the graphical placement and combination of operators, queries formed by operators and their according input streams. This plan is later optimized and executed by the DSMS and continuous results are provided.

2.1.4 Examples of DSMSs

Examples of DSMS can be found in the following fields of application:

- **Sensor Networks** In sensor networks DSMSs are used to monitor local weather by an interconnected network of independent sensors [BGS01; GM04; SG06].

  In this application area DSMSs serve as monitors for a certain environment or domain which issues continuously changing parameters. These are aggregated over streams from multiple sources to provide a comprehend picture of the current environment situation.

  Another factor is the ability to compensate for failures from single streams as the complete amount of streaming data from all sources is processed.

- **Network traffic analyser** DSMSs are used as traffic analysers for digital network traffic within corporate networks or as part of Intrusion Detection Systems (IDSs) [CGJ+02; GKM+01].

In this scenario traffic statistic are evaluated to become aware of critical situations within the communication infrastructure, i.e. heavy data load, unusual behaviour of data within the network or internal respectively external attacks on the architecture.

- **Financial tickers** The fast and instant analysis of stock exchange share price variation is the driving factor to utilize DSMSs in this area [ZS02]. They are used to discover correlations in price and stock changes which apply continuously over one day of trade.

  Another possibility is the usage for future value forecasting which allows the user to be one leap ahead of the market and fully utilize his trade opportunities.

2.1.5 Stream Windows

Stream windows allow a structured access on data streams [BBD+02; BDM02; GKS01; CF02]. They provide the ability to process a fixed and manageable amount of stream data respectively elements requested by a stream query. The method to split data streams into windows allows the DSMS to give best approximated answers on the data stream elements' content. This regards change in the monitored parameter's values, density of incoming elements or change in the structure of stream elements over a certain period or amount of elements [GO03].

Stream windows can be categorised by the factor their size is relying on which is either time-based or count-based. Another category indicator is the method for updating and evaluation of the stream window.

Size handling differs stream windows into *logical* or *physical* windows. A *logical window's* size is based on the amount of tuples processed within itself. They are count-based by the stream elements they take into account. A *physical window* is defined through the size of the time interval which it encompasses. This lets these kind of stream windows be time-based.

Both kinds of stream windows have in common that their size is formed by two endpoints, one starting and one endpoint either set on the number of stream elements (count-based) or a point in time at a certain element of the data stream.(time-based).

Stream windows are now categorised by the setting of these endpoints.

- **Fixed window** Both endpoints are fixed at their position on the stream. They are static in their size either by the number of counted elements processed or the time interval. Starting point and endpoint have a definite and absolute value in regard to the input stream and by this cannot be modified.

- **Sliding window** Endpoints in sliding windows have a constant distance between them which makes the window static in its size. The difference to the *fixed window* is the ability of both endpoints move over the stream's elements either forward or in a small amount backwards. This window is most utilized for time-based operations as it involves the ability to move through the time dimension.

  A visualisation of the sliding window model is shown in Figure 2.3 on Page 10. In this figure t_S is the time of the starting point, t_E is the endpoint of the window

within the time dimension and t_C is the current time like already depicted in Figure 2.1 on Page 6. The window size in this example is t_W which is the constant sum of t_S and t_E ($t_W = t_S + t_E$).

This window type is most used within current DSMS applications [DM06] and which has also be adopted in the implementation of the validator in SCSQ.

- **Landmark window** This window type has a fixed starting point whereas the endpoint is able to move over the stream. This allows the window to be dynamic in size and process a differing amount of stream elements. It can increase the amount of stream elements within its borders until a certain limit is reached and then decrease its size again by i.e. aggregating stream elements [GKS01].

  Another possibility is for the window to increase its time interval until a certain window size is reached, then evaluate itself and restart with a smaller initial value.

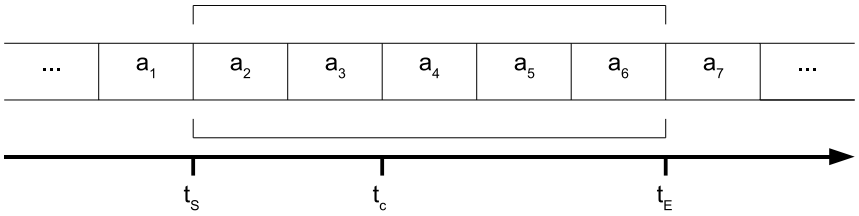

Figure 2.3: Abstract visualisation of the sliding window on a data stream [BDM02]

In the third option to categorise windows by their update behaviours three different kinds of windows are feasible. The *jumping window* and the *eager re-evaluation window* are both characterized by their behaviour on re-evaluating themselves after a new stream element did arrive.

The *jumping window* uses lazy re-evaluation in which after a certain time period newly arrived items are re-evaluated within the stream query's window. The *eager re-evaluation window* re-computes the query's window after every newly arrived stream element to provide an up to date result.

The third kind is the *tumbling window* [CcC+02]. In this window type the update interval is significantly greater than the actual window size. This leads to windows on the stream which are not able to overlap each other regarding processed stream elements.

At this point we have outlined the basic definitions for DSMSs. We described that a *data stream* is a continuous sequence of data items which form the elements of the data stream. We showed that different methods exist to access those elements and with the *stream window* technique it is possible to do this in an aggregated manner. In the next section, we going to line out one certain implementation of a DSMS which has been done at the UDBL.

2.2 The AMOS

In this section we are going to explain the *Active Mediators Object System* (AMOS) which has been used as basic architecture in its second revision [Ris07a] to implement the LR benchmark in the *Super Computer Stream Query Processor* (SCSQ) [Ris07b] at the Uppsala DataBase Laboratory (UDBL) and which will be the underlying system used for the implementation of the streamed validator described within this thesis. As the name AMOS states it is a DSMS which allows to access data sources by the user through its intermediate nature of being a software layer between the data source and the user's application. The AMOS [FRS93; RJ01; JKR03] is an object-oriented system with a descriptive query language (AMOSQL), a database facility interface and a modular architecture. It has the capability of distributed and parallel computing through the support of multiple database access which are located on any number of AMOS servers.

Figure 2.4: Role of AMOS in the data processing [FRS93]

The basic principle of AMOS is depicted in Figure 2.4. All three objects can be of multiple instances. This allows several applications to access a varying amount of differing data sources through a range of AMOS servers.

The basic architecture of the AMOS is built on a main memory based platform and as such tries to hold and process all data within the main memory during the existence of an AMOS server's instance. A consequence of this is that data only written to hard disk by concrete user interaction or on server shut-down.

The AMOS features object-oriented DBMS facilities which are transaction based. A main feature is the AMOS query language which is called AMOSQL. It incorporates those object-oriented features like object methods and parameters or interfaces as also descriptive syntax which is similar to SCSQ. AMOSQL offers the ability to access multiple AMOS server instances within one query and to process these data sources in parallel. It also has active-rule support which denotes that a query is not executed the moment it is requested to the system, but under certain conditions which are defined within the query's signature.

In the AMOS data is represented as local structures of AMOS modules which allows a comprehensive view on it without any need of translation or transformation to be accessible through the system.

The most important role within the chain of data source access and process play the different mediators which are provided by the AMOS. These mediators may integrate or access data sources, they apply operations on data sources and forward their results or notify the external application or user of changes within the system.

The four kinds are:

- **Integrators** These mediators retrieve, translate and combine data from different data sources. They allow a comprehensive view on different data repositories. These may not only differ in location but also in schema and format.
- **Monitors** These mediators supply mechanisms to notify applications and users, if an update on a query has occurred. Their behaviour is controlled by the setting of constraints under which they should report an update.
- **Domain** Within these mediators application models and database operators are allocated. They define the general conditions on how a data source can be accessed by an application. Another feature of domain mediators is the storage of data representation schemata, i.e. query definitions are stored here.
- **Locators** They locate needed mediators within the system or data sources for a query to be executed. This retrieval process is feasible within the network of AMOS servers, not only on a single server itself.

2.2.1 SCSQ

The *Super Computer Stream Query Processor* (SCSQ) has been proposed and described in the paper of Zeitler and Risch [ZR06]. It has been detailed and evaluated later on in the paper of Zeitler and Risch [ZR07]. It is based upon the AMOS and extends it of the capability of utilizing continuous queries on data streams.

The initial idea to develop SCSQ is derived from the need to process the data of the LOFAR [Lof] project. This project is an array of 25k antenna receivers whose signals are digitalized. This massive amount of continuous data has to be analysed and processed to obtain events which occur within this stream. This led to the basic requirement of SCSQ to process data ad hoc and in parallel without the obligation to store data on hard disk as the sheer amount of data would easily exceed available disk space.

In ongoing development the SCSQ system has been adopted [Sve07] to implement the Linear Road (LR) benchmark [ACG+04]. This implementation was the base for the development of the validator described within this thesis.

The SCSQ extends the AMOSQL by its special vector treatment operators and the ability to issue continuous queries on data streams. This allows to handle large vectors as input or output streams by such as normal object types within the AMOS. In SCSQ vectors can be of any other type of the AMOS type system, even types themselves. This way large amounts of data can be computed very fast in a comprehensive manner as these vectors are also feasible as being used as join criteria on data streams. These data streams may be used within the SCSQ system for queries or sub queries by forwarding them as query parameters. This grants the possibility to process these streams massively in parallel execution.

An overview of the SCSQ architecture can be seen in figure 2.5 on page 14. This figure depicts that stream queries are executed in parallel on different nodes within the SCSQ. On all components of the architecture parts of the SCSQ system are executed. The

general flow of processing data is from the *input streams* towards the *user's application* whereas control messages go from user application by through several steps to the *preperators*.

The *front cluster* (FC) is the interface with which the user's application does interact. It consists of the single *client manager* and several front cluster nodes (*FC Node*). Within one front cluster node the *query coordinator* (QC) and the *front stream processor* (FSP) are located.

The *Blue Gene* (BG) *cluster* aggregates several Blue Gene (BG) nodes (*BG Node*). One BG node shelters the single instance of the *compute node coordinator* (CNC). On all BG nodes the *query manager* (QM) for continuous queries and their sub objects the *stream processor* (SP) are placed.

The *backend cluster* harbours any amount of stream *preperators* which transform the input streams in any needed form for the downstream processing.

The chain of task fulfilment is as follows. The user application solely interacts with the client manager in the front cluster. The client manager accepts the continuous queries from the user application, enriches them with meta data like identifiers for user application or client manager and propagates the query to a QC on a FC node for further processing.

The QC compiles incoming continuous queries and stacks these in its queue and posts instructions on query execution onto the BG nodes. Another task of the QC is to initiate the front stream processors (FSPs). These post-process the stream query results from the BG nodes that they are deliverable as a query's completion to the user's application.

The Blue Gene's CNC polls the QCs' queue to receive new queries and also the QC itself for new orders. The CNC does start new SPs, assigns them to queries and also does survey their execution process. For every query the CNC sets a SP as *query manager* (QM) which is responsible for the correct processing of this query and has the authoritative to start new SPs to compute sub queries.

The QM which is responsible for its designated query has to deliver the results of this query to a FSP which was launched by the QC arranging this particular query. This QM also monitors the complete computation of its query and communicates with the preperators of the backend cluster to fetch needed input data.

The processing of a stream query may end by successful execution and result deliverance, on explicit user signal or halt constraint within the query's definition to catch faulty behaviour. All this behaviour is controlled by the transfer of start/stop messages between the components, i.e. stream processors (SPs).

As we have until this point described how the basic assumptions and standards for DSMS are defined and outlined the used and implemented DSMS SCSQ as a possible solution which regards all these theories. In the next section we are going to explain a scenario which intends to make all the different DSMSs implementations comparable towards each other.

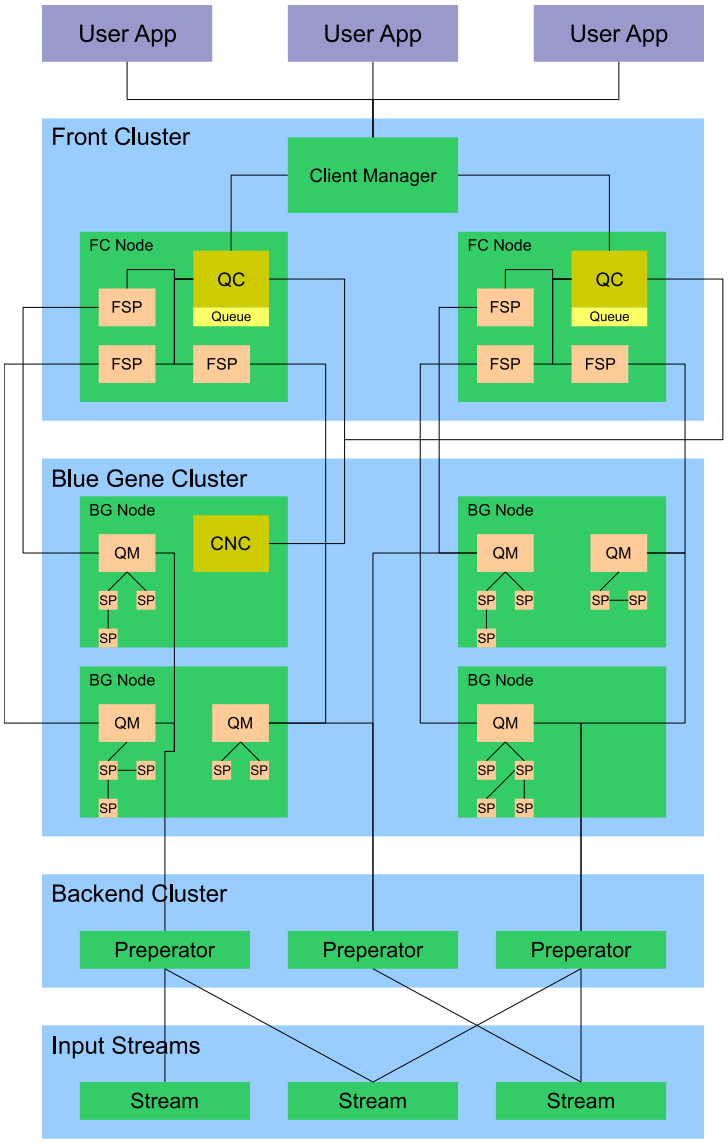

Figure 2.5: Abstract architecture of SCSQ [ZR06]

2.3 The LR benchmark

The *Linear Road* (LR) benchmark has first been proposed and described in the Master thesis of Tibbets [Tib03]. It has been redefined the following year in the conference paper of Arasu, Cherniack, Galvez, et al. [ACG+04]. For the basic definitions of the LR benchmark we are going to rely on the newer conference paper whereas for the evaluation criteria we are going to lean on the explanation in the Master thesis.

The LR benchmark standardizes this scenario for DSMSs to make implementations in this new field of data management systems comparable to each other. This need raised due to the variation in implementations of DSMS. The LR benchmark measures the performance of a DSMS in means of scalability, throughput and predictability of output. It construes a trustworthy base to make these indicators significant.

The motivation of the LR benchmark is the real world scenario of a metropolitan traffic system. These have to cover more and more traffic whilst the city's growing. Individual traffic infrastructure is not capable of being extended as fast as needed in these areas, to solve this issue *variable tolling* [Poo03] has been proposed by urban traffic research institutions.

The basic layout of the LR benchmark is the picture of the Linear City which is depicted in Figure 2.6 on Page 19. It consists of ten expressways of which each has a length of a hundred miles. Every expressway is parted into hundred equidistant mile long segments which can be travelled in two directions either westwards or eastwards. A segment, as shown in figure 2.7 on Page 19, has five lanes per direction, one lane for cars to enter the expressway on this segment, one lane for cars to exit the expressway on this segment and three lanes for cars to travel through this segment on the expressway.

The main problem in the Linear City is traffic congestion on the expressways. To control this cars which travel on the expressways are accounted for a certain amount of toll, so to force cars to leave the expressway during heavy congestion by charging a high tolling fee. To keep track of one car's paid tolls every car has its own account on which all its payments are recorded. This data ranges from the tolls during one vehicle trip, over tolls during one day and several vehicle trips till historical data on payments during the last month.

A trip of one car is defined by a car entering one expressway at one segment bound towards a certain direction, travelling on this expressway for a any amount of segments and exiting the expressway at one segment. During this trip the car issues a position report of itself for every $30sec$ of his trip. Based on the aggregated examination of all car's position reports the toll on a expressway is calculated.

The position report is a nine tuple which consists of the following parameters:

$$(\text{Type} = 0, \text{Timestamp}, \text{VID}, \text{Speed}, \text{Xway}, \text{Lane}, \text{Direction}, \text{Segment}, \text{Position})$$

- *Timestamp*: An absolute point in time which is counted upwards in seconds from zero at the start of the simulation.
- *VID*: Unique identifier of this certain vehicle.
- *Speed*: Current velocity of the vehicle at this certain moment of issuing the position report. It ranges from zero to one hundred miles per hour.

- *XWay*: The identifier of the expressway the vehicle is travelling on. (0 . . . 9)
- *Lane*: The identifier of the vehicle's lane. (0 . . . 4)
- *Direction*: The identifier of the vehicle's direction. Either zero for eastbound direction or one for westbound direction.
- *Segment*: The identifier of the vehicle's segment on the expressway. (0 . . . 99)
- *Position*: A position identifier within the LinearCity coordinate system.

During its trip the car may request statistical data regarding his past daily expenditures, its current account balance or its travel duration. These request queries are tuples of the following form:

Account Balance:
(Type = 2, Timestamp, VID, QID)

Daily Expenditure:
(Type = 3, Timestamp, VID, XWay, QID, Day)

Travel Time:
(Type = 4, Timestamp, VID, Xway, QID, Segment_{start}, Segment_{end}, DoW, ToD)

The new parameters to be considered in these request queries are:

- *QID*: Unique query identifier to relate query to the issuing car.
- *Day*: Identifies the day on which the expenditure should have occurred. Ranges from one to sixty-nine whereas one is yesterday and sixty-nine is ten weeks ago.
- $Segment_{start}$: The segment on which the car started its trip.
- $Segment_{end}$: The segment on which the car ended its trip.
- DoW: Day of the week at which the requested travel has happened.
- *ToD*: Time of the day in minutes of that day at which the requested travel has taken place.

All these single tuples can be merged into one large tuple in which the type identifier specifies which parameters are to be examined and contain a set value. This tuple would read as:
(Type, Timestamp, VID, Speed, XWay, Lane, Dir, Seg, Pos, QID, S_{start}, S_{end}, DoW, ToD)

The base for the historical query requests is the historical data provided as input data by the LR benchmark. It summarizes ten weeks of toll history on all expressways for all cars. This data also consists of tuples of the outlook:

Toll History:
(VID, Day, XWay, Tolls)

- *Day*: The day during the last ten weeks on which the toll has been paid. See Day description for historical request queries.
- *Tolls*: The amount of toll spend for this certain vehicle on that very day during all of its trips on exactly this expressway.

Segment History:
(Day, Min, XWay, Dir, Seg, LAV, CNT, Tolls)

- *Day*: For every day during the last ten weeks a tuple does exist.
- *Min*: For every minute of that day a tuple is held.
- *LAV*: The latest average velocity of all cars in this segment at this day and minute.
- *CNT*: The counted sum of all cars in this segment at this day and minute.
- *Tolls*: The toll charged for this segment at this day and minute.

All these tuples mentioned above form the input data for the simulation of the LR benchmark in the SCSQ system. During the simulation run the SCSQ computes the output data which regards the correct processing of vehicle trips, the estimation of tolls of cars during these and fulfilling a car's historical request queries. A special fact to consider is the appearance of accidents during the simulation run. A car is viewed as having an accident if four of his consecutive position reports contain a velocity of zero and are reported from the same position, i.e. expressway, direction, segment and lane have the same values as in the former position report. This leads to the avoidance of charging a toll for this segment and every five segments up or downstream to this segment according to the direction the car was headed. Also every other car within this defined reach is notified of the accident occurrence to be able to leave the expressway and by such avoid the accident's site. An accident is regarded as cleared after no position reports from the accident's site by the involved cars do come up.

If no accident does take place normal toll calculation for this segment is executed. This calculation is based on the current latest average velocity of all cars within a segment. If the latest average velocity is greater than or equal to forty miles per hour, no toll is charged as traffic is able to stream fluently on the expressway. If the latest average velocity is below the 40 miles per hour limited it is calculated this way:

$$2 \times (numvehicles - 50)^2 \tag{2.1}$$

In this equation *numvehicles* is the total number of vehicles within this segment which was the base for the computation of the latest average velocity. After the toll

has been calculated a car is notified on its amount via a *toll notification* which is then added to the car's account balance.

During a simulation run of the LR benchmark the concerned DSMS takes all the input data (position reports, historical data, account balance requests, daily expenditure requests, travel time requests) and computes the according output data (toll notifications, accident notifications, request answers and segment statistics) which is continuously streamed and written into a flat output file. This file is the base for the LR benchmark validation tool which verifies if the output meets the evaluation criteria of the LR benchmark. We are going to describe this in detail in Chapter 3.

Linear Road (LR) has been evaluated in the Master thesis of Svensson [Sve07] and Tibbets [Tib03]. The former Master thesis used SCSQ [ZR06] upon AMOS [FRS93] to measure load and performance on a single standard personal computer laptop. This has been done to achieve comparable measurements of the performance of the SCSQ towards other DSMSs like Aurora [JACc+03] or STREAM [ABB+04]. The latter MA thesis implemented the LR benchmark in the Aurora DSMS [JACc+03] to provide a basic proof of concept and harness test of the LR benchmark.

All these simulation runs of the former described LR benchmark have to evaluated for their correctness to form a valid evidence on the effectiveness of the DSMS. This is undergone by the evaluation and validation tool proposed in the Master thesis of Tibbets [Tib03]. In the next chapter we are going to outline the basic principles of this tool and analyse an existing implementation to gain a better understanding of its principles, work mechanisms and drawbacks.

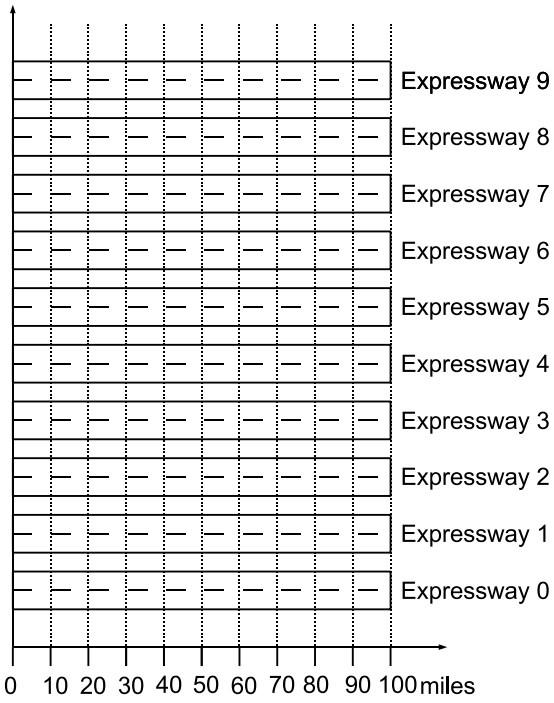

Figure 2.6: Linear City's expressways [ACG+04]

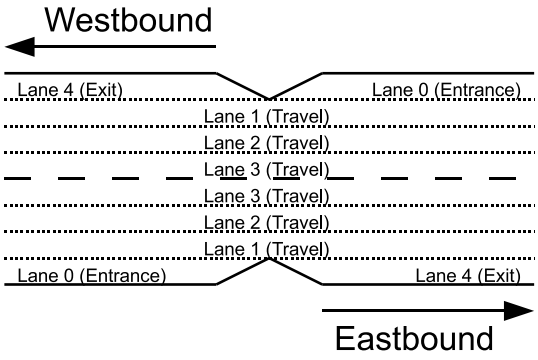

Figure 2.7: Structure of one expressway's segment [ACG+04]

3. Chapter

Analysis of the validator in PERL

The LRs validation tool (validator) has been proposed in the Master thesis of Tibbets [Tib03]. In this thesis the needed criteria and data representation specifications had been stated upon which an implementation of a validator for the LR benchmark has to be based.

We are going to explain these requirements here as they build the base for the later on analysed and evaluated validator in Perl. This analysis is undergone for the sake that the streamed validator implementation will try to remodel the methods used in this existing implementation. The evaluation we will do to proof that a streamed version of the validator is feasible in matters of saving time and computing cost. This way, simulation outputs can be regarded as trustworthy at a earlier point in time after the benchmarking.

The evaluation criteria for the output of the DSMS which did implement the LR benchmark can be divided into two grand categories [Tib03, pp.40]. *Performance requirements* encompass the timely limits for the computation of toll notifications on position reports of vehicles entering a new segment, accident notification of vehicles at an accident's site and five segments upstream or downstream of it, answering requests regarding a vehicle's account balance, its daily expenditures and its travel time. The values of these limits can be read in Table 3.1. These requirements are of a strict nature for the DSMS which implements the LR benchmark and have to be met during the validation process.

The second category are *accuracy requirements*. These are less strict than the performance requirements and may be slightly less stringent followed by the implementing DSMS. The meaning of these requirements is to allow a certain amount of "fuzziness" within the processing of input data and their accurate appearance as simulated output. An overview on the limits is given in Table 3.2. In example, if a vehicle has been reported as being at a position p at a times t by the DSMS, it is supposed to have issued its position report with the timestamp t_s at this position p in a period of $t_s \in [(t-30sec), t]$. If this assumption holds has to be verified by the validator. Another example is the calculation of segment statistics which contain the sum of vehicles and the latest average velocity within this segment. These statistics by definition have to be computed once every minute based on the issued position reports at this segment within the last five minutes. It is allowed for those statistics to have a "laziness" of sixty seconds. This means that the statistical value can be accurate at a time t sixty seconds

after the upper bound of the processed five minute period. Travel time responses do not insist on a numerical limit, but are only permitted a five percent margin of error in their return value.

The accuracy and performance requirements have to be evaluated in a defined and well formed process. On how to achieve this in general we are going to explain in the next section.

Query	Performance Requirement
Toll Notification	15 seconds after the position report announcing a vehicle as entered into a new segment.
Accident Notification	15 seconds after the last position report stating that two vehicles reside at the same position with a velocity of zero.
Account Balance	60 seconds after the point in time the request query was given.
Daily Expenditure	90 seconds after the point in time the request query was given.
Travel Time	120 seconds after the point in time the request query was given.

Table 3.1: Limits for Performance Requirements [Tib03]

Historical Data	Accuracy Requirement
Vehicle Position	30 seconds
Vehicle Account	60 seconds
Accident	30 seconds
Segment statistics	60 seconds

Table 3.2: Limits for Accuracy Requirements [Tib03]

3.1 Verification Process

In this section we explain the general process on how to validate the correctness of the simulation output. This is necessary to further understand on how the validator in Perl operates and how the underlying methods are implemented in Perl and can later be used in the streamed validator implementation in SCSQ.

The *verfication process* relies on three different kinds of data. The first is the *input data* of the LR simulation run. It is not only the base for the DSMS to simulate the three hours of expressway traffic, but also for the computation of the *expected values* by the validator. The latter constitute the ranges for the accuracy requirements, possible answers to the request queries and notifications on tolls and accidents which are supposed to have been issued during the DSMS's simulation. For this they are the base in which the third kind, the actual output of the simulation, has to fit in. Due to this the *simulation output* and the *expected values* are the two sets of values which is compared during validation. These three kinds are depicted in the shape of a parallelogram in Figure 3.1.

The process of verification can been seen in its abstract form in Figure 3.1. For the actual validation a further step is to preprocess data to bring it into a form which can be handled by the validator's scripts. This step is especially lasting for the output data of the simulation which is meshed into one big single file containing all output tuples. These have to be separated and stored in different files according to their type and striped off unnecessary elements.

The second big preprocessing step is on base of the input data to compute the expected data which should have been produced by the simulation. This data encompasses upper and lower bounds of data which constitute a range of supposed values. This data is for example expected segment statistics like the latest average velocity or the count of vehicles and further on the amount of charged toll for that segment at a certain point in time. Another part of the expected data are toll or accident notifications which should have been issued by the DSMS during the simulation.

In the *verfication process* the *expected values* and the *output* are brought together and compared in certain ways. At first it is checked, if the amount of notifications from the expected data equals this of the actual output. In case this holds, it is tested, whether the output data lays within the bounds of the expected data. In the last step the *performance* and *accuracy requirements* of the timely factor are audited. Only after passing all these measures a simulation run is assessed as successful and can be regarded as a trustworthy base for the comparison of the DSMS towards other DSMSs.

In this section we lined out the basic concept of the validation process which is in short:

1. Preprocess input and output data
2. Process expected values
3. Compare expected and output data
4. Declare correct or incorrect simulation run

In the next section we are going to explain on how this process has been implemented utilizing Perl and PostgreSQL.

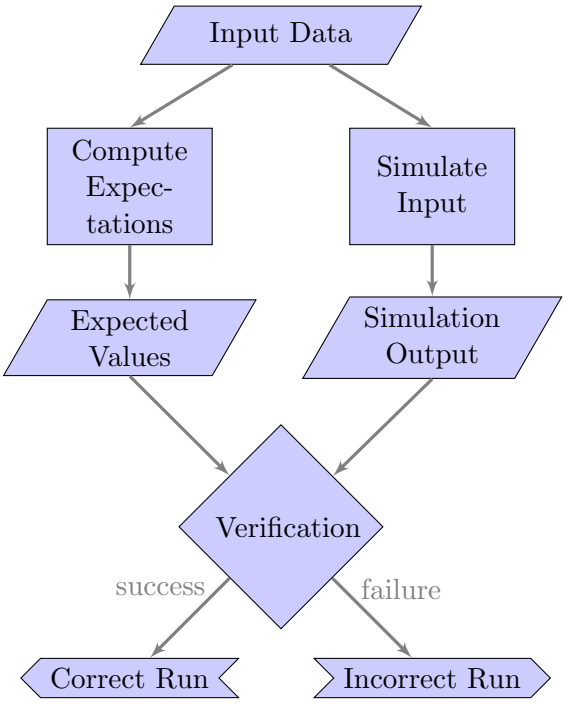

Figure 3.1: Verification process as flow chart

3.2 Validator Structure

To fully understand the operational and methodological structure of the LR benchmark validation tool we have to analyse the given scripts in Perl. This analysis is also the starting point of remodelling the validator in its streamed manner in SCSQ.

The implementation of the LR validation tool has been provided by Igor Pendan and Nga Tran [Tra07]. It utilizes Perl and PostgreSQL and has to be run as a batch job after the simulation run of the LR benchmark. The PostgreSQL DBMS is used for the storage of computed expected values, the simulation output and comparison results. The various Perl-Scripts control the validation process logic as they invoke the needed SQL-queries to compute the expected values and compare those to the actual output. An overview on the scripts shipped with the validator implementation is shown in Figure 3.2 on Page 26.

At this stage a major trade off in this way of validation is see able. The processing of large amounts of plain tuples within a relational DBMS is not feasible as this mechanism always invokes to load all produced input and output data into the data base whereas input and output easily exceed the Gigabyte border. Secondly this data is quite invariant in its containing values and as such does not deliver any good selectivity needed for fast relational data base operations. A third point is the grand need

of aggregation and group operations within the treated data which have to be stored to be used within further processing. This circumstance also bears the risk to store a lot of derived and thus redundant data in the data base which unnecessarily enlarges the amount of data to be written into the data base.

In Figure 3.2 it is depicted that the script *validate* is the central script which does invoke and control the complete validation process. From here all further scripts are called. First step in the validation is to drop all tables and indexes of former verifications which is done by the script *dropAllTables*. In it the needed "drop index"- and "drop table"-statements with all constraints in question are executed. The next script to be run is the *import* script. It first creates all tables for further storage and processing and then loads all needed data into these tables. The needed import has to be stated in a property file which is examined by this script. The *indexes* script creates indexes on all tables concerning the computation of toll alerts and notifications to speed up data access and processing in scripts later on. The script *XWayLoop* estimates the number of expressways to be validated and calls its derived scripts once each for every expressway of the simulation. After all following validation from the *xWayLoop* script the scripts *splitByByte* and *accountBalanceAnswer*. The former controls the compliance of the output with the *performance requirements* for the daily expenditure and account balance request queries and the latter checks whether the correct account balance has been calculated.

In the second stage of the validation process which runs in the *xWayLoop* the first step is to create temporary tables for accident and toll alerts. This is done in the *createAlertTmpTables* script. The third stage in the validation is started through the *generateAlerts*. In this stage the actual calculation of the expected values is proceeded and comparison of those to the simulation output is undergone. After the proceeding of the third stage in the scripts *addAlerts* and *renameAlertTmpTables* of the second stage the calculated values are added to the data base for the comparison and requirement matching in the first stage done in the scripts *splitByByte* and *accountBalanceAnswer*.

The first script to be run in the third stage is *runDDL*. It creates the tables used in this stage for the storage of the computed values and comparison results, in example accident statistics, toll and accident notifications and if any of those is not in the data either of the expected values or the simulation output. In the *extractAccidents* script tuples are filtered, if their position report's velocity is zero. These reports are stored and endued with an index for faster data access. Afterwards these tuples are compared on the condition, if two cars hold the same positions. On successful comparison these are saved as possible accident sites. The *insertStatistics* script inserts into the data base the segment statistics which have been computed by the *extractNumVehicles* and *extractLavs*. Further, it adds indexes on this data. In *extractNumVehicles* the amount of cars for each segment during each minute is written into a temporary table for later usage, also indexes on this data are created. The next step done in *extractLavs* is to calculate the latest average velocity from the input per segment and minute. These values are inserted into the statistics tables as an aggregated value over the last five minutes of data per segment and minute. This means for a minute t the latest

average velocity over the interval $[(t - 5min); t]$ for a certain segment is stored. In the *calculateTolls* script all segments which exceed the latest average velocity of forty miles per hour, have an amount of passing vehicles lower than fifty or are an accident site or near to one are filtered and their according expected toll is set to zero. For all other segments the base toll (Section 2.3 on Page 15) is calculated and saved. The *createAlerts* script computes the expected accident and toll notifications (alerts) for all segments and stores these for comparison which is done by updating the stored simulation output based on the expected values.

The execution of all these scripts marks one validation run. It is therefore measurable that the validation process is complex to model and implement, but also in this batched job manner highly time cost intensive. The proof of this assumption is shown in the next section in which the behaviour of the implementation of the validator in Perl is evaluated.

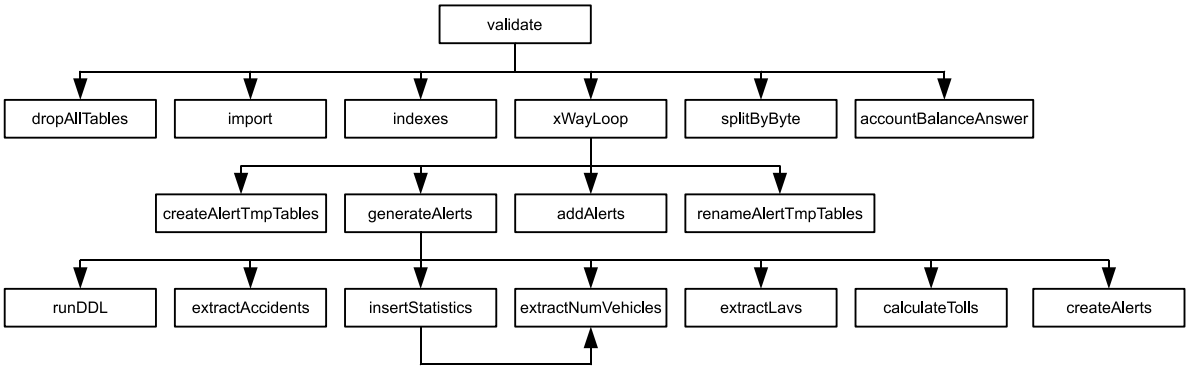

Figure 3.2: Abstract structure of the validator in PERL

3.3 Performance

To find out how much computation time it does take to validate a certain amount of simulated expressways and how the validation in Perl with PostgreSQL does scale, we have evaluated the validation process by using the output data of the SCSQ in a range from one expressway up to five expressways on a single node within a cluster of Pentium IV CPUs. Another goal of this evaluation has also been to recognise the limits which the implementation of the validator in SCSQ would have to undercut and form a base on which cost savings by parallel validation in timely measures can be calculated and compared to the batch validation of the validator using Perl and PostgreSQL.

The outcome of this evaluation can be seen in Figure 3.3 on Page 28. Initially it is readable that the minimum running duration for the validation of the output of one simulated highway is two hours and forty-five minutes. Considering that one simulation run contains input data for a period of three hours, we can state that the validation of the output would never need to significantly drop below the same duration as the simulation and hence a parallelisation of the validation process would not be a

degradation of itself. Thus it is reasonable to undertake the parallelisation step already by timely constraints.

A further look on Figure 3.3 shows that from the starting duration the regression curve rises in an exponential manner. It ends at thirteen hours for the validation of a simulation output of five expressways. If we consider that the LR benchmark defines the Linear City (Section 2.3 on Page 15) as consisting of ten expressways and we extrapolate these values on this matter, it is foreseeable that the validation of the whole Linear City would take substantially longer than one day. This circumstance can be anticipated as too long lasting and thus being not feasible for a reasonable validation.

In this section we have shown that by matters of heavy time consumption and small computing performance, but using up a large amount of computing power an implementation of the LR validation tool in a scripted manner utilizing a relational database is not sufficient in means of gaining qualified validation results in an adequate duration and cost.

We have analysed the general verfication process to build a base and explain what steps need to be undergone to transfer the validator in Perl to a streamed version in SCSQ. We detailed on the concrete scripts of the Perl validator to line out several footprints in the implementation, in example the large use of temporary tables which implies a big amount of write actions in the database or the extra need to create indexes on all important relations to speed up data access. Further, we did this to show what methods have to be reinstated in the streamed version of the validator to gain the same trustworthiness of the verfication process as the Perl implementation does. How we tried to approach and solve these problems, we are going to explain in Chapter 4.

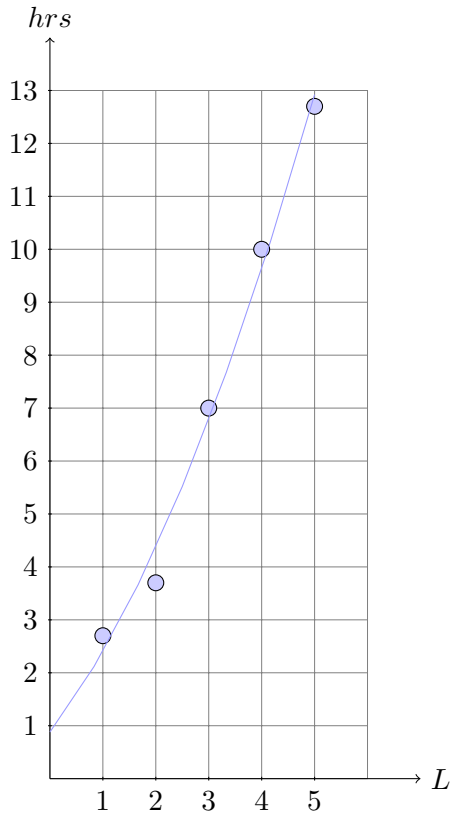

Figure 3.3: Computing time in hours of the validator on different amounts of expressways (L)

4. Chapter

Implementation of the validator in SCSQ

In this chapter we are going to line out our approach towards a streamed implementation of the LR evaluation tool. It relies on the results of the analysis of the implementation done in Perl, so the major goal is to rebuild this given tool by utilizing the features of SCSQ and the existing implementation of the LR benchmark in this DSMS. It is done to parallelise the validation process towards the simulation as shown in Figure 4.1. We undertake these measures to extinguish the drawbacks of the Perl validator implementation, in example high computing and time costs.

In our explanation, we detail that we have achieved to implement the validation of accident notification and the extraction of segment statistics by dividing the stream of position reports into timely limited windows (subwindows) by utilizing the ability of SCSQ to part streams into substreams which are supervised by their own *stream processor* (SP) and can be processed independently for every query. We then describe the work which is open to provide a complete reimplementation of the validation tool, in example toll notification, travel time and daily expenditure requests and that the ongoing development in the SCSQ complicates this goal.

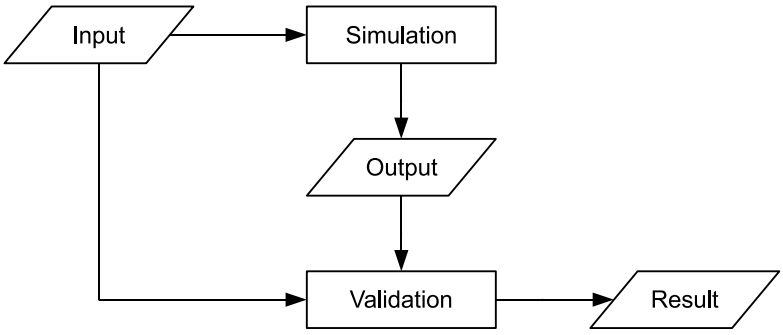

Figure 4.1: Parallelisation of the validation process

4.1 Development

The first step in the development of the streamed validator was to set which software engineering methods have to be chosen to achieve a well formed implementation. For this matter current development at the UDBL defined the guidelines which had to be followed. This is at first to provide a modular structure of the implementation to ease changes within the source code and make a certain implementation's part simply apparent. In short, the source code had to be able to speak for and explain itself.

The second chosen development method was the test-driven development by utilizing regression tests as it has been described and defined in the books on software development of Beck [Bec03] and Myers [Mye04]. This method had been selected to achieve that a change in or new insertion into the source code would not impose corrupt output values. These assumed correct output values are taken from example runs of the validator in Perl. This is the base on which the whole implementation of the validator in SCSQ trusts on to accomplish its goal of a streamed verification. These constraints lead to the following structure of the base implementation:

- **Validator Files**
  - *validator*_$(*modulename*).*osql*
  - *validator*_$(*modulename*)_$(*submodulename*).*osql*
- **Regression Tests**
  - *test*_$(*modulename*).*osql*
  - *test*_$(*modulename*)_$(*submodulename*).*osql*

The SCSQ relies on two different programming languages. These are the AmosQL mentioned in Section 2.2 and Lisp. The latter is integrated into AMOS via the definition of foreign functions [Sin04]. The Lisp has been used to implement the aggregation and conversion functions by Erik Zeitler of UDBL. This is for the matter that the LR benchmark tools produce tuples as streaming data whereas the SCSQ processes vectors.

This vector processing simplifies the handling of the streaming data of the input and the simulation output as no relational treatment is necessary which is done in the Perl validator. Due to this streaming data can be taken care of in a direct manner and aggregated easier. Aggregation of the streaming data helps to view the needed snippet out of the stream and provides a comprehensive sight to compare input and output data. This behaviour is modelled similar to the sliding window described in Section 2.1.5.

The overview on the current implementation of the validator in SCSQ is depicted in Figure 4.2. According to this structure is the one of the needed regression tests as shown in Figure 4.3. In both arrangements execution is initiated by a single start script. This is the *master* script for the validator and the *test* script for the regression tests.

In the streamed validator the script *aggops* defines the operators to aggregate the stream tuples into aggregated vectors. These vectors represent a time window of either one or five minutes of stream data which is processed. The Lisp script *lread* is from the LR implementation in SCSQ and defines the foreign functions for conversion and reading of the tuples from LR into vectors.for the SCSQ validator. The script *stat* is also in Lisp and computes the needed segment statistics like number of vehicles and latest average velocity. In *subwindow* the data vectors are aggregated to groups ranging from one to five minutes of duration. In the scripts *accident*, *actual*, *expected* and *comparison* the complete process to validate the simulation's detected accidents is detained.

Similar to the topology of the validator's scripts is the one of the regression tests. For every script within the validator there is one according regression test to ensure the correctness of the validation after a changement to the source code. At this stage of implementation, we can state that the following goals have been fulfilled:

- **Latest Average Velocity**
  - done as external functions in Lisp
  - supplies functions for counting cars and computing velocities in segments
  - used for toll calculation within segments

- **Accident Alerts**
  - done internal as functions in SCSQ
  - consists of the submodules: actual, expected & comparison
  - *actual* - extracts accident alerts out of simulation output
  - *expected* - computes all possibilities of expected alerts according to the simulation input
  - *comparison* - compares the actual accident alerts (simulation output) with the computed expected ones. Looks if the latter met the computed limits and time criteria as mentioned in the introduction of Chapter 3.

- **Subwindow**
  - providing subwindow functions for the validation on streams
  - functions encompass either window size of one or five minutes

These modules are implemented as usable scripts which can either be started seperately after the simulation to work on its output file or as SCSQ continuous stream queries fetching and analysing the current output of the running simulation in SCSQ. To be able to run all needed processing in SCSQ allows to only use the main memory as temporary storage and thus forfeit read/write cycles on hard disk as the validator in Perl does need them.

How to run a short example of this reached stage is shown in Listing 4.1. It utilizes a predefined script which runs the validation of happened accidents within the simulation run and results in the output of the non-valid accident notifications. To achieve this the script runs several parallel stream queries and splits them into subqueries whereas every subquery gets its own stream processor (SP) to deliver fast computing results.

At this development stage, we can't provide fully qualified performance or evaluation results of the streamed version of the validator. We have shown that a streamed version is feasible and achievable by remodelling the LR evaluation tool in Perl. But, certain compromises have to be undergone by changing the manner of validation from batch job after simulation to streamed verification process parallel to simulation, in example no use of static storage, fast aggregation of simulation output data and time critical achievement of comparison result.

```
1 Amos II Release 10, v4
2 [scsq] 1>
3 /* non streamed variant by loading functions out of a file */
4 [scsq] 1> < 'src/non_streamed_example_accident.osql';
5 /* output of non valid vectors */
6 [scsq] 1> nonvalid();
```

Listing 4.1: Validator console & example file (non streamed variant)

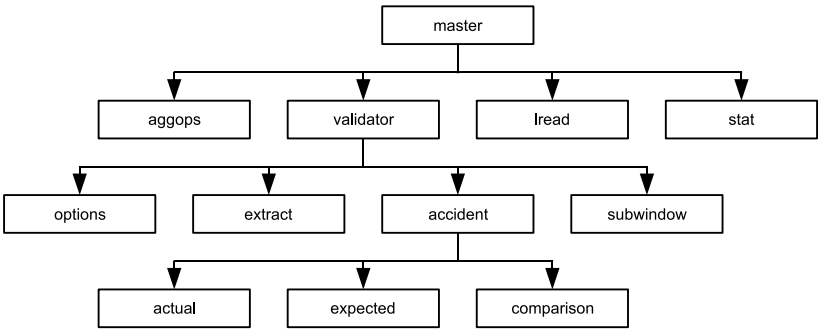

Figure 4.2: File order for the validator

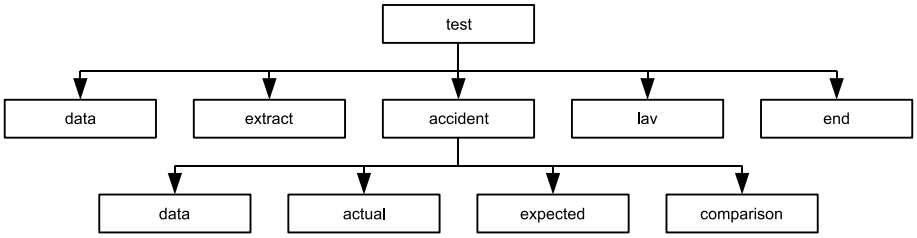

Figure 4.3: File order for the regression tests

4.2 Further Work

There is still need for the implementation of the following modules and also a wrapper file which ties everything together and thus imposes a proper way to run the validator on the stream of the simulation. Modules to be implemented:

- historic queries
- toll calculation
- account balance
- daily expenditure

These left open tasks are for the following reasons:

- *Lack in source code documentation and annotation*: This circumstance complicates the analysis of the existing source code and the cognition of methods in which the validation process is undergone and proceeded. This leads to a very strong trial and error based way of development which is time and effort consuming.
- *Ongoing development in SCSQ*: As the SCSQ DSMS is not a final software system with a static code base, its evolving changes during development raise the need to continuously proof and redefine the implementation. Another factor is the non-existence of needed data process methods which yields the demand to implement and test those on their sustainability. Thus making development a constant struggle with every new version.
- *Narrowed implementation approach*: Thus far the implementation of the validator in SCSQ reached to accomplish the goal of computing expected values and checking time limits at the same time. This has tangled the development in the process of trying to achieve to much at the same time. Further development has to focus on time limit checking to simplify the development and make starts from there easier.

This would be the roadmap for an ongoing implementation of the validator in acs SCSQ to accomplish the paralleled verification process. Further, the work of Bagherjeiran, Cantú-Paz, and Kamath [BCPK05] and Datar and Muthukrishnan [DM02] which explains how to detect anomalies within a data stream could be used for the streamed validation tool in the manner that variations from the defined time limits in the output stream of the simulation can be viewed and recognised as such and by this an implementation can speed up the validation process.

5. Chapter

Conclusion and Future Work

In this paper we have outlined and described the basic definitions and requirements for DSMS and how a possible streamed variant of a validation tool for streaming data can be approached.

The basic problems within streaming data we identified as by the non-persistant and continuous nature of this kind of data which disallows to store this data, makes it impossible to access past data and only process on present data is feasible [BBD+02; GO03].

We showed what kind of implementations and approaches in this area exist, in example the Aurora DSMS [JACc+03], the STREAM DSMS [ABB+04] or the TelegraphCQ [CCD+03b]. We explained on the *Active Mediators Object System* (AMOS) and *Super Computer Stream Query Processor* (SCSQ) systems [FRS93; ZR06] of UDBL to show in what manner the problems of streaming data are advanced and solved. Especially taking into consideration that these two systems were used to achieve the goal of a streamed verification of streaming data.

Furthermore, we explained a widely used scenario for benchmarking the performance in means of scalability, throughput and predictability. This scenario is the *Linear Road* (LR) benchmark [ACG+04] which lets a DSMS simulate the traffic and toll control system on virtual expressways. Scalability is tested by the parallelisation of traffic to a maximum on ten different expressways whereas the amount of the expressways is freely choose-able by the DSMS. Throughput is benchmarked in the manner that traffic becomes heavier over time which lets more and more vehicles needing toll calculation on the expressways. Predictability is assessed by the determination of certain output through given input.

To lay all these benchmarking results on a trustworthy base and by such making these fully compare-able a LR benchmark evaluation tool had been stated in the Master thesis of Tibbets [Tib03]. We have analysed and explained an existing implementation of this evaluation tool and showing that its relational approach is not efficient in matter of time and computing expenditure by measuring the performance of this tool and through the analysis of its source code which showed that the use of temporary tables and relational indexes brings up heavy read/write cycles to access data and by this slows down data processing.

Based on the analysis of the relational evaluation tool we lined out an implementation of it in a streamed manner. To reach this goal we utilized the SCSQ system. We

described the current status of development which gives a partly implementation in matters of verifying the accident notification and segment statistics. At this early stage it is already foreseeable that the parallel use of the validation is feasible and reduces time and computing cost which shows that the goal of implementing a streamed validator for streaming data is manageable and achievable. We described what problems occurred during and complicated the development, in example short documentation and ongoing development in the SCSQ. Further we explained what and how future work has to look like, especially the left open verification of toll notification or historical requests.

During the work at this thesis no other DSMS benchmarks became known to us nor any other such work to implement a streamed validation tool of the LR benchmark.

Bibliography

[ABB+04] Arvind Arasu, Brian Babcock, Shivnath Babu, et al. *STREAM: The stanford data stream management system.* Technical Report 2004-20. Stanford InfoLab, 2004.

[ABW03a] Arvind Arasu, Shivnath Babu, and Jennifer Widom. *An abstract semantics and concrete language for continuous queries over streams and relations.* Research Report. Stanford University, 2003.

[ABW03b] Arvind Arasu, Shivnath Babu, and Jennifer Widom. "CQL: A language for continuous queries over streams and relations". In: *DBPL '03: Proceedings of the 9th international workshop on database programming languages.* 2003, pp. 1–19.

[ACG+04] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, et al. "Linear road: a stream data management benchmark". In: *VLDB '04: Proceedings of the 30th international conference on very large data bases.* Toronto, Canada: VLDB Endowment, 2004, pp. 480–491.

[Agg06] Charu C. Aggarwal. *Data streams: models and algorithms.* 3rd ed. Springer, Berlin, 2006.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, et al. "Models and issues in data stream systems". In: *PODS '02: Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems.* Madison, Wisconsin: ACM, 2002, pp. 1–16.

[BCPK05] Abraham Bagherjeiran, Eric Cantú-Paz, and Chandrika Kamath. *Design and implementation of an anamoly detector.* Tech. rep. UCRL-TR-213599. Lawrence Livermore National Labratory, 2005.

[BDM02] Brian Babcock, Mayur Datar, and Rajeev Motwani. "Sampling from a moving window over streaming data". In: *SODA '02: Proceedings of the 13th Annual ACM-SIAM symposium on discrete algorithms.* 2002, pp. 633–634.

[Bec03] Kent Beck. *Test-driven development by example.* The Addison-Wesley Signature Series. Addison-Wesley, 2003.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. “Towards sensor database systems”. In: *MDM ’01: Proceedings of the 2nd international conference on mobile data management*. 2001, pp. 3–14.

[CcC+02] Donald Carney, Ugur Çetintemel, Mitch Cherniack, et al. “Monitoring streams - a new class of data management applications”. In: *VLDB ’02: Proceedings of 28th international conference on very large data bases*. 2002, pp. 215–226.

[CCD+03a] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al. “TelegraphCQ: Continuous dataflow processing for an uncertain world”. In: *CIDR ’03: Proceedings of the 1st biennial conference on innovative data systems research*. 2003.

[CCD+03b] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, et al. “TelegraphCQ: Continuous dataflow processing”. In: *SIGMOD ’03: Proceedings of the 2003 ACM SIGMOD international conference on management of data*. 2003, p. 668.

[CDT+00] Jianjun Chen, David J. DeWitt, Feng Tian, et al. “NiagaraCQ: A scalable continuous query system for internet databases”. In: *SIGMOD ’00: Proceedings of the 2000 ACM SIGMOD international conference on management of data*. 2000, pp. 379–390.

[CF02] Sirish Chandrasekaran and Michael J. Franklin. “Streaming queries over streaming data”. In: *VLDB ’02: Proceedings of the 28th international conference on very large data bases*. Hong Kong, China: VLDB Endowment, 2002, pp. 203–214.

[CGJ+02] Charles D. Cranor, Yuan Gao, Theodore Johnson, et al. “Gigascope: High performance network monitoring with an SQL interface”. In: *SIGMOD ’02: Proceedings of the 2002 ACM SIGMOD international conference on management of data*. 2002, p. 623.

[DGI+02] Mayur Datar, Aristides Gionis, Piotr Indyk, et al. “Maintaining stream statistics over sliding windows”. In: *SODA ’02: Proceedings of the 13th annual ACM-SIAM symposium on discrete algorithms*. San Francisco, California: Society for Industrial and Applied Mathematics, 2002, pp. 635–644.

[DM02] Mayur Datar and S. Muthukrishnan. “Estimating rarity and similarity over data stream windows”. In: *ESA ’02: Proceedings of the 10th annual european symposium on algorithms*. London, UK: Springer-Verlag, 2002, pp. 323–334.

[DM06] Mayur Datar and Rajeev Motwani. “The sliding-window computation model and results”. In: *Data streams: models and algorithms*. 3rd ed. Springer, Berlin, 2006. Chap. 8, pp. 149–167.

[FRS93] Gustav Fahl, Tore Risch, and Martin Sköld. "AMOS - An architecture for active mediators". In: *NGITS '93: Proceedings of the international workshop on next generation information technologies and systems*. 1993, pp. 47–53.

[GKM+01] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, et al. *QuickSAND: Quick summary and analysis of network data*. Technical Report 2001-43. AT&T Labs-Research, 2001.

[GKS01] Johannes Gehrke, Flip Korn, and Divesh Srivastava. "On computing correlated aggregates over continual data streams". In: *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on management of data*. 2001, pp. 13–24.

[GM04] Johannes Gehrke and Samuel Madden. "Query processing in sensor networks". In: *IEEE Pervasive Computing* 3.1 (2004), pp. 46–55.

[GO03] Lukasz Golab and M. Tamer Öszu. *Data stream management issues - a survey*. Technical Report CS-2003-08. School of Computer Science, University of Waterloo, 2003.

[JACc+03] Daniel J. Abadi, D. Carney, U. Çetintemel, et al. "Aurora: A data stream management system". In: *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on management of data*. San Diego, California: ACM, 2003, p. 666.

[JKR03] Vanja Josifovski, Timour Katchaounov, and Tore Risch. "Functional data integration in a distributed mediator system. Modeling, Analyzing and Integrating Heterogeneous Data". In: *Functional approach to data management*. Ed. by P. Gray, L. Kerschberg, P. King, et al. First. Springer, 2003. Chap. 9, pp. 1–29.

[KCC+03] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, et al. "TelegraphCQ: An architectural status report". In: *IEEE Data Eng. Bull.* 26.1 (2003), pp. 11–18.

[Lof] *LOFAR (LOw Frequency ARray) - Website*. 2009. URL: http://www.lofar.nl (visited on 02/01/2009).

[MWA+03] Rajeev Motwani, Jennifer Widom, Arvind Arasu, et al. "Query processing, approximation, and resource management in a data stream management system". In: *CIDR '03: Proceedings of the 1st biennial conference on innovative data systems research*. 2003.

[Mye04] Glenford J. Myers. *The art of software testing, second edition*. 2nd ed. Wiley, 2004.

[Poo03] Robert W. Poole. *HOT Networks: A new plan for congestion relief and better transit*. Tech. rep. Transportation Studies Reason Foundation, 2003.

[Ris07a] Tore Risch, ed. *AMOS Website*. 2007. URL: http://user.it.uu.se/~udbl/amos/ (visited on 02/01/2009).

[Ris07b] Tore Risch, ed. *SCSQ Linear Road Website*. 2007. URL: http://user.it.uu.se/~udbl/lr.html (visited on 02/01/2009).

[RJ01] Tore Risch and Vanja Josifovski. "Distributed data integration by object-oriented mediator servers". In: *Concurrency and Computation: Practice and Experience* 13.11 (2001), pp. 933–953.

[SG06] Sharmila Subramaniam and Dimitros Gunopulos. "A survery of stream processing problems and techniques in sensor networks". In: *Data streams: models and algorithms*. 3rd ed. Springer, Berlin, 2006. Chap. 15, pp. 333–352.

[Sin04] Munindar P. Singh, ed. *The practical handbook of internet computing*. Boca Raton, Florida: CRC Press, 2004.

[Sve07] Mårten Svensson. "Benchmarking the performance of a data stream management system". MA thesis. Uppsala Universitet, 2007.

[Tib03] Richard S. Tibbets III. "Linear Road: Benchmarking stream-based data management systems". MA thesis. Massachusetts Institute of Technology, 2003.

[Tra07] Nga Tran, ed. *Linear Road Benchmark - Website*. 2007. URL: http://www.cs.brandeis.edu/~linearroad/index.html (visited on 02/01/2009).

[ZR06] Erik Zeitler and Tore Risch. "Processing high-volume stream queries on a supercomputer". In: *ICDEW '06: Proceedings of the 22nd international conference on data engineering workshops*. Los Alamitos, CA, USA: IEEE Computer Society, 2006, pp. 147–152.

[ZR07] Erik Zeitler and Tore Risch. "Using stream queries to measure communication performance of a parallel computing environment". In: *ICDCSW '07: Proceedings of the 27th international conference on distributed computing systems workshops*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 65–75.

[ZS02] Yunyue Zhu and Dennis Shasha. "StatStream: Statistical monitoring of thousands of data streams in real time". In: *VLDB '02: Proceedings of 28th international conference on very large data bases*. 2002, pp. 358–369.

Selbstständigkeitserklärung

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Magdeburg, den 23. März 2009

Stefan Kegel