TUHH
Hamburg University of Technology

Bachelor Thesis

Züleyha Toptas

Implementation of the Linear Road Benchmark on the basis of the real-time stream-processing system Storm

June 24, 2014

supervised by:
Prof. Dr. Ralf Möller
Dr. Özgür Özcep

Hamburg University of Technology (TUHH)
Technische Universität Hamburg-Harburg
Institute for Software Systems
21073 Hamburg

TUHH STS
Software
Technology
Systems

Eidestattliche Erklärung

Ich, Züleyha Toptas, versichere an Eides statt, dass ich die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit wurde in dieser oder ähnlicher Form noch keiner Prüfungskommission vorgelegt.

Hamburg, den 24. Juni 2014

Züleyha Toptas

Contents

1 Introduction

The interest in systems that can handle data streams has grown increasingly. *Stream Data Management Systems* (SDMS) can process potentially infinite data streams, so-called continuous data, as well as produce the output in real-time. A recent example of such an SDMS is *Storm*. It has already been adapted by many companies such as Twitter [1].

The performance of SDMS is of major interest, more specifically the accuracy of the procedures of these systems and their response time. With the help of *Linear Road Benchmark*, a stream data management benchmark, an SDMS can be compared against other existing systems. The Linear Road Benchmark has already been applied on stream data management systems like AURORA [2] and SCSQ [3].

Linear Road simulates a fictional urban area. In this city, *variable tolling* is applied. Variable tolling means that tolls are charged for vehicles, whereas the amount of these tolls vary under certain conditions, such as accident proximity and the number of cars driving in this area. Toll charging demands stream data as input. In order to explode the capacity of the examined systems, scale factors are used to determine the amount of stream data produced that SDMS can process while meeting specific requirements such as response time and accuracy.

Linear Road has not been implemented yet for Storm. This work constitutes an attempt at implementing Linear Road for Storm for the purpose of evaluating the efficiency and the accuracy of the system. The focus will be on the query for toll notification however. Moreover, formally the stream data is typically stored temporarily in databases; here, Maps, which are data collections provided by Java, will serve as a substitution. The implementation will be realized with Storm-0.8.2, the stable version released in 2013, and Java 6 on a Virtual machine with a Linux (Ubuntu) operation system.

This work is organized as follows: Chapter 2 presents the concept and features of the stream processing system Storm. The Linear Road Benchmark is explained in detail in Chapter 3. There, the structure of the fictional city that Linear Road simulates, is presented. In addition, the formalization of the stream data and the functions that are needed to implement toll charging in Linear Road, are provided. In Chapter 4, the concept of the implementation is presented. The results are discussed in Chapter 5. Finally, Chapter 6 provides a conclusion of this thesis.

2 Storm

Storm is a system that can process data streams in real-time [1, 4]. In this chapter, this will be presented in detail. Firstly, Storm will be introduced. Then, the concept of Storm will be shown, consisting of a network of nodes, so-called *spouts* and *bolts*. Furthermore, an example of a cluster in Storm will be illustrated. Finally, the projection of parallelism in Storm will be depicted, however, in this work the parallelism has not been applied yet.

2.1 Introduction

Storm is a system that can handle data streams and process them in real-time. This means that the system consumes the data in the moment as it is being entered. It is a free and open source distributed system that was created by Nathan Marz. Preponderantly, it is written in Clojure and Java. The first major release was in 2011, whereon further versions followed until now. Over 25 companies make use of Storm such as Twitter, Yahoo!, and many others. Stream processing, distributed RPC and continuous computation count to use cases that Storm has.

2.2 Stream Data

Stream data, which constitutes an unbounded sequence of tuples, is what needs to be processed. Each tuple consists of a numbers of certain ordered elements. These elements can be strings, primitive types, or even objects. When using objects, those objects need to be serialized. Stream data have the potential to become infinitely large. Stream data managements systems, like Storm, can handle those data.

2.3 Topology

A topology is a component in Storm that specifies how to process stream data. It is an abstraction that is passed to a *Storm Cluster*, whichever will be explained in Section 2.4. A topology is a network consisting of *Spouts* and *Bolts*. Both, spouts and bolts have processing logic inside.

Spout A spout handles the input stream of a Storm cluster. It is responsible for getting the input stream into the system and passing it on to a bolt. Typically, it reads data from files, databases or from Twitter APIs.

Bolt A bolt has the task to consume the stream it gets. Typical tasks of a bolt are running functions, filtering tuples or communicating to databases, and even more. A bolt can get its input from a spout, a bolt, or multiple bolts, but the output of a bolt can only be passed to other bolts. A bolt can not only run functions or talk to

databases, but can also produce new streams, which can be described as a stream transformation.

Every node in such a network starts with a spout and ends with a bolt. Imagine a topology as a graph of stream transformations. A topology processes forever or until it gets killed. Links between nodes indicate how streams should be passed around between nodes. In a topology, each node executes in parallel.

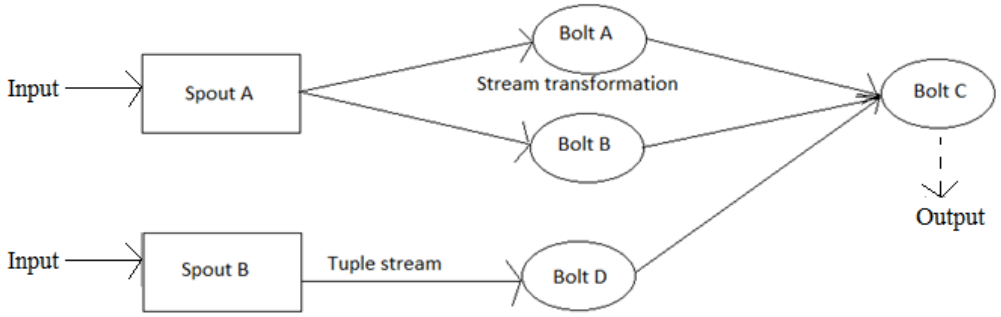

Figure 2.1: An own Illustration of a Topology based on [1].

The Figure 2.1 shows an example of a topology. The edges illustrate which bolt is subscribed to which stream. If a spout or a bolt emits data (tuples) to a stream, then this is sent to every bolt that is subscribed to this certain stream. Spout A emits a number of tuples. Bolt A and Bolt B, which are subscribed to the stream, get this data. If Bolt A and Bolt B have processed and emitted the data, Bolt C will get the transformed stream from Bolt A and Bolt B because it is subscribed to their streams. If Spout B emits tuples to a stream, then Bolt D will get this stream, process it and emit the output. Then, Bolt C will get the stream of Bolt D. The stream that Bolt C gets, is a new, and therefore, transformed stream. In Storm, a tuple consists of fields, and each field can hold an arbitrary object such as strings, byte arrays or primitive types. Every spout and bolt has to declare output fields for the tuples they emit. If, for example, Spout A emits a string and a corresponding integer value, then the Spout has to declare 2 output fields.

2.4 Storm Cluster

A *Storm Cluster* processes the input data as defined by the topology. Such a cluster has two major components: a *master node* and *worker nodes*. The master node runs a

daemon called *Nimbus*. Nimbus is responsible for distributing code around the cluster, handling failures, uploading the computation to the cluster and launching workers. The topology that is submitted to Nimbus is defined in the main function of a project. Each worker node runs a daemon called *Supervisor*. The Supervisor listens to Nimbus and starts or stops a *Worker* if Nimbus tells so. A *worker node*, consisting of a Supervisor and a corresponding Worker, can start a *worker process*. A worker process consists of a Worker, an Executor and a Task. It executes a subset of a topology it belongs to. It can run a number of Executors for a number of spouts or bolts of the same topology. An Executor is a thread spawned by a Worker, which it can run a number of Tasks for the same spout or bolt. One thread always exists for all of its Tasks. A Task processes the data and is a thread in a Worker in that way. The number of Tasks for a spout or bolts is static and therefore it does not change throughout the lifetime of a topology; however, the number of Executors can be changed. Only one Worker per port can be defined.
Figure 2.2 shows an illustration of the interaction between the daemons. The whole communication between Nimbus and Supervisor is realized with *Zookeeper* [6]. Zookeeper is a configuration service for large distributed systems. Nimbus as well as Supervisor is stateless. If you kill the Nimbus, it will start again and will behave like nothing happened. The states are kept in Zookeeper or on a local disk. Therefore, this leads to Storm clusters being stable.

2.5 Features

The main properties of Storm are:

Scalable Huge amounts of messages can run per second with parallel calculations.

Fault-tolerant If faults appear during an execution, the messages will be reassigned. Storm will run a computation either forever or until it gets killed. If a worker dies, it will be automatically restarted.

No data loss guarantees Storm guarantees that each unit of data will be processed.

Combination with any programming language It does not run on a single platform. The core of Storm contains a Thrift definition for submitting topologies. Thrift is compatible with any programming language. Components of Storm that are Non-Java virtual machines can communicate to Storm over a JSON-based protocol.

Real-time It processes streams and does continuous computation.

Recent It is open-source and is still being developed.

Different modes Storm can be run both, locally and remotely. In the local mode, Storm simulates Workers with threads. On the other side, in the distributed mode, Storm works as a cluster of machines.

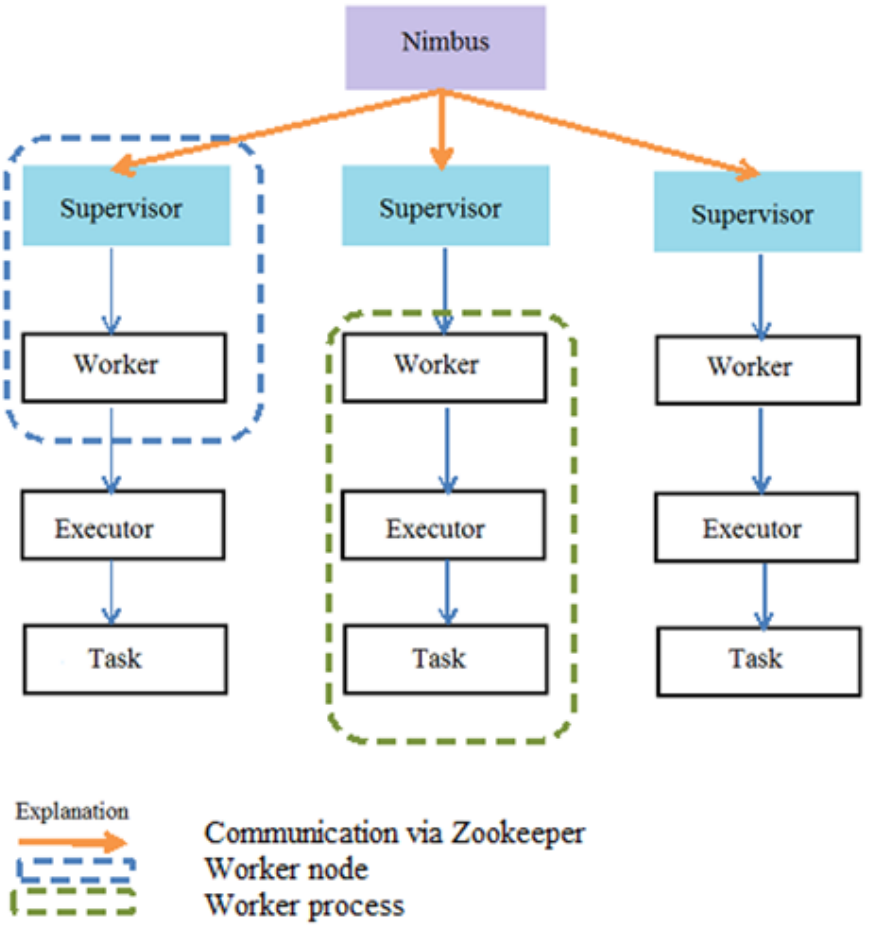

Figure 2.2: An own Illustration of the interaction between Nimbus and Supervisor based on [5].

2.6 Parallelism

In Storm, the amount of parallelism for each node can be specified. Every spout or bolt can run a number of tasks. Therefore, groupings are used, which determine which task in the subscribing bolt, the tuple is sent to. Following groupings can be used:

- Shuffle grouping, streams are distributed randomly. Every bolt gets an equal number of tuples.
- Fields grouping, streams are portioned and grouped by fields.
- All grouping, tuples of a stream are replicated across all bolts.
- Global grouping, the whole stream goes to a single bolt.
- Direct grouping, the node that emits tuples controls which component will receive the tuple.

Storm uses additionally *ZeroMQ* [7] for reliable messaging. It is responsible for avoiding congestion of messages. By default, Storm runs one Task per Executor. When only the number of Executors is defined, then the number of Tasks is equal to the number of Executors. Otherwise, it runs as specified by the developer. It is possible to increase the number of worker processes in a topology across machines in a cluster, the number of Executors per spout or bolt, and the number of Tasks per spout or bolt. The splitting is explained by an illustration of an example shown in the following: Storm spawns that

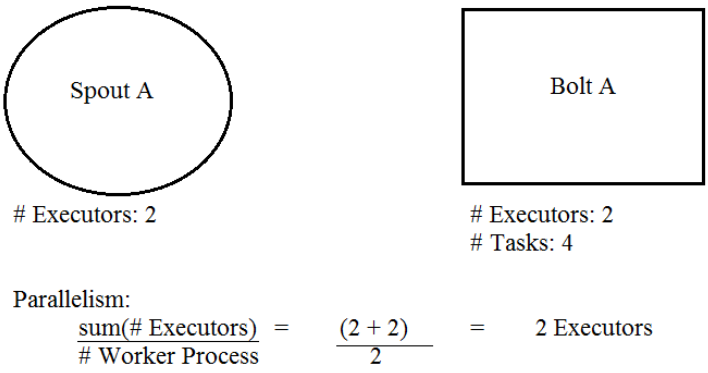

Figure 2.3: An own Illustration of an Example of a Parallelism based on [1].

number of threads that was determined across the cluster for the execution. In Figure

2.3, the number of the worker process is set to 2 and the topology consists of one spout and one bolt. The number of the Executors in Spout A is set to 2, in Bolt A it is set to 2. Bolt A was configured to use 4 tasks. This means that each worker process spawns 2 Executors. Each Executor will run 2 tasks for Bolt A.

3 Linear Road Benchmark

The *Linear Road Benchmark* evaluates stream data management systems based on accuracy and the response time ("based on A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts" [8]). In this chapter, this will be explained in detail. Firstly, the background will be clarified. Then, the structure of the fictional are, the so-called *Linear City* will be presented. Afterwards, the formalization of the stream data and the requirements in Linear Road will be exposed. Though Linear Road provides a number of queries, the only query that will be implemented is the query for *Toll Notifications*. This query uses continuous data and not historical data, which is used for the other queries. Finally, the tools that the Linear Road Benchmark provides, will be explained.

3.1 General

The Linear Road Benchmark can run with different scale factors. Those factors can get increasingly large. The assumption is that the benchmark runs with a scale factor whereby an SDMS cannot meet the requirements anymore. Thereby, an SDMS has to implement queries that are provided by *Linear Road*. Linear Road simulates a fictional urban area. This area consists of expressways. The number of expressways denote the value of the scale factor. The maximum scale factor to which an SDMS can react while meeting specified requirements, is mentioned as L-rating. Amongst others, *Toll Notification* is a query in Linear Road.

3.2 Linear City

Linear City is a fictional metropolitan area that simulates a simple urban setting with a fixed length of expressways. It is 100 miles wide and 100 miles long, whereby a position in that city can be determined using (x, y) coordinates. The coordinate x describes the number of feet east and coordinate y, the number of feet north from the starting point. The origin starts in southwest (0, 0) extending to (527999, 527999). Figure 3.1 shows an example of Linear City.

There are 10 expressways, numbered from 0 to 9 running in parallel in Linear City. For simplicity the expressways run only horizontally. Each of them is running 10 miles apart. Every expressway consists of four lanes in both, east and west direction: 3 lanes for traveling (lane 1-3) and one for entrance (lane 0) and exit (lane 4) ramp. In each expressway, there are 100 on-ramps and 100 off-ramps in each direction, which are divided into 100-mile-long segments. An example of an expressway segment is shown in Figure 3.2.

Each vehicle is appointed with a sensor that sends a position report every 30 minutes. A position report contains data that determines the exact position of a specific vehicle.

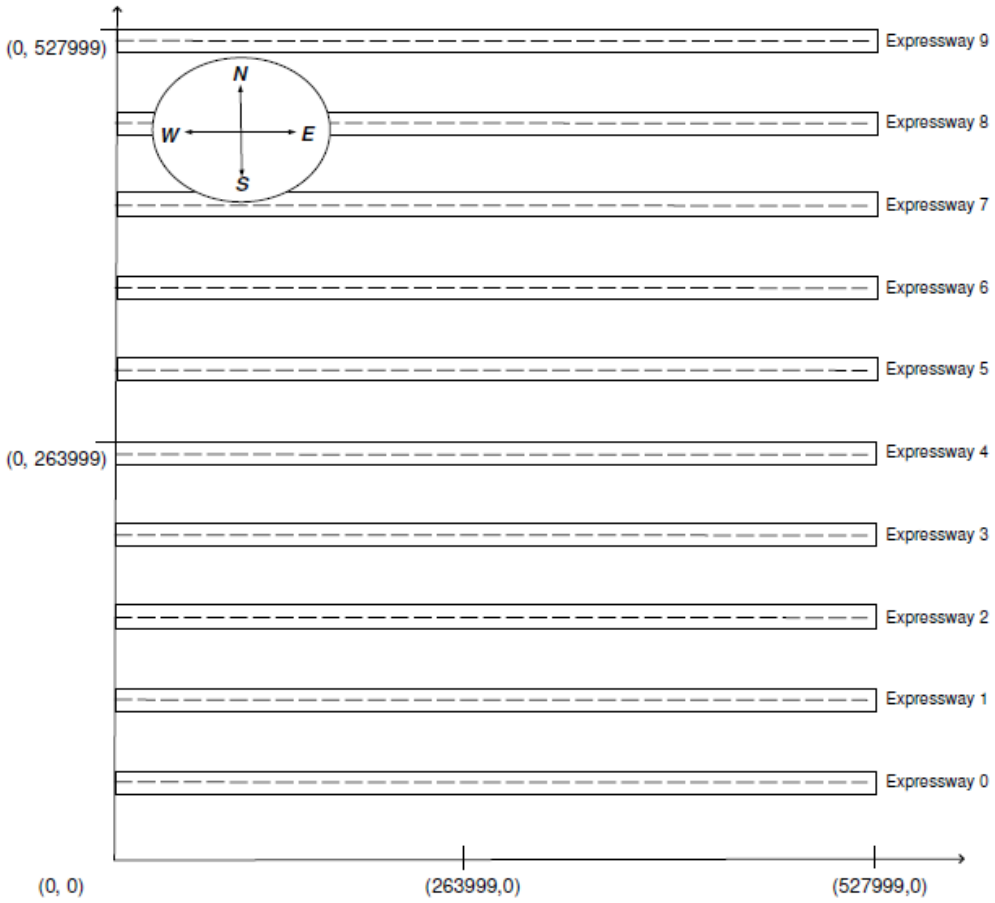

Figure 3.1: Geometry of Linear City based on [8].

With these position reports it is possible to calculate tolls for a specific segment that a vehicle enters.

3.3 Stream Data

The stream data can be distinguished into 2 types of tuples: *Continuous Query Requests* and *Historical Query Requests*. Historical Query Requests asks a database that stores previous actions for information. Such a request is either an *Account Balance*, or a *Daily Expenditure*, or a *Travel Time Estimation*.
The Account Balance requests the total amount of assessed tolls for a vehicle. The request that asks for a vehicle's total toll on a specified expressway, on a certain day of the prior 10 weeks, is the Daily Expenditure Request. The Travel Time Estimation requests

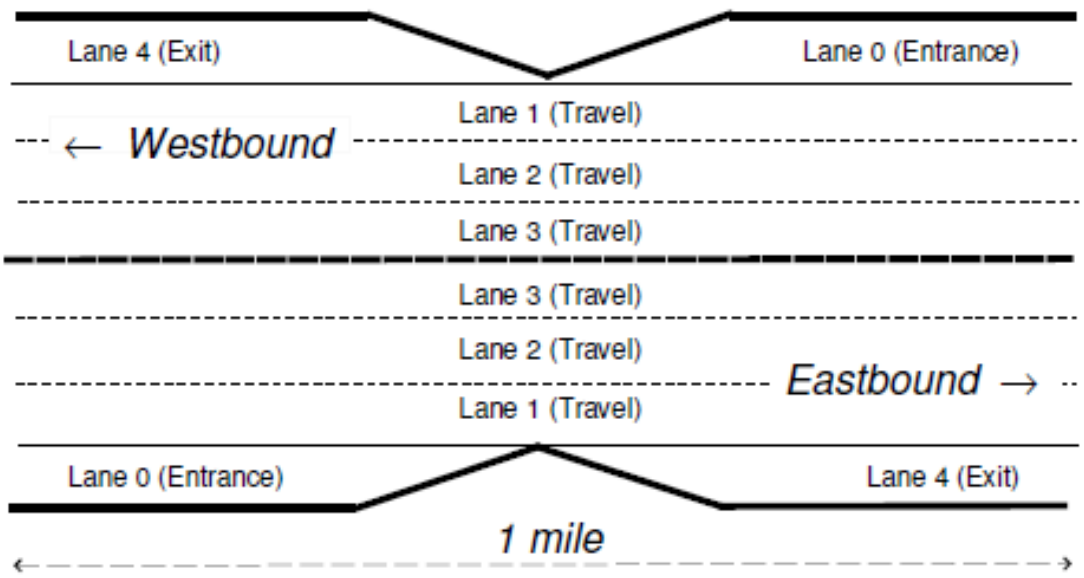

Figure 3.2: An Example of an Expressway Segment based on [8].

the estimated toll and travel time for a journey on a given day of the week, at a given time, and on a given expressway.
Continuous Queries update its results constantly based on the contents of data. A Position Report is an example of a continuous query. Because the toll notifications are based on continuous queries in Linear Road, we will have a closer look at the function for a toll notification. A position report p is a tuple and has the form as follows:

$$p = (Type = 0, Time, Vid, Spd, XWay, Lane, Dir, Seg, Pos)$$

Type specifies a tuple as position report, if Type = 0.

Time presents a time stamp from 0 to 10799, identifying the emit time of a position report. Note that in a 3 hour simulation period, there are 10800 seconds.

Vid is a unique integer number from 0 to MAX INT in a position report. It identifies a vehicle; therefore, *Vid* is shorthand for vehicle identifier.

Spd is shorthand for speed. It is an integer number that reflects the speed of a vehicle in miles per hour (mph), remaining between 0 and 100. Therefore, a vehicle never drives faster than 100 mph.

XWay numbered from 0 to 9, identifies the expressway the position report was received from.

Lane identifies the type of a lane on an expressway. There are four lane types: 0 if the lane is an entrance ramp, 1 to 3 if it is a travel lane, and 4 if it is an exit ramp.

Dir points the direction, in which the vehicle is traveling. If the vehicle is traveling in eastbound, the direction is 0, otherwise it is 1 for westbound.

Seg indicates the mile-long segment of an expressway. The number of a segment remains between 0 and 99.

Pos indicates the horizontal position of a vehicle. It describes the gap between the vehicle's position and the origin, which is the westernmost position. This means nothing but Pos = x coordinate. The range of a position resides between 0 (the origin) and 527999 (the easternmost position).

XWay, Lane, Dir, Seg, and *Pos* are functions over a vehicle's (x, y) coordinate. The set of all position reports is defined as P. Therefore, $p \in P$.

3.4 Requirements for Linear Road

One requirement for Linear Road is the response time. That is, every tuple p has to contain two timestamps: one time that is the time stamp of the tuple when p was generated (p.Time), and one that specifies the emit time of p (p.Emit). The time stamp p.Time is already given; it is contained in each input tuple for the toll notification already included for every vehicle reporting its position, since this is generated by the Linear Road. The time stamp p.Emit has to be created with the help of a system call. Another requirement is to have enough memory for exploring the L-rating, which is the maximum scale-factor that a system can reach while fulfilling certain conditions, for example, the response time and the accuracy. Before applying Linear Road, 512 MB RAM and approximately 3 Gigabytes per expressway have to be provided. The more expressways are used, analogously, the more memory has to be provided. More details regarding running the benchmark is presented in Section 3.6.

3.5 Toll Charging

In this section, it is shown when and how tolls are charged in Linear Road. Preconditions, requirements, output, and background information of some formalization are presented. Finally, the functions for calculating a toll notification will be explained. Those functions will be needed for the implementation in Chapter 4. A continuous query has two types that should be implemented: *toll processing* and *accident detection*. These types are contained in the query for toll notifications. Account balance, daily expenditure, and travel time estimation belong to historical queries. In this thesis, only the toll notification is implemented, since it deals with continuous queries. Linear Road distinguishes between a toll notification and a toll assessment. Whenever a vehicle emits a position report with entering in a new segment, a toll for that certain segment has to be calculated and

the vehicle has to be notified of that toll. The amount of a toll depends on the current congestion on that segment and the proximity of accidents. This is the scenario of a toll notification. A toll assessment is calculated whenever a vehicle is identified by a position report changing a segment. Then the reported toll for that segment that is being quited is assessed to the account of the vehicle. However, the part that will be observed is the toll notification. The requirements for a toll notification are like so:

Trigger	Position report, q
Preconditions	q.Seg $\neq \overleftarrow{q}$.Seg, l $\neq$ EXIT
Output	(Type = 0, VID: v, Time: t, Emit: t' Spd: Lav(M(t), x, s, d), Toll: Toll(M(t), x, s, d))
Recipient	v
Response	t' - t $\leq$ 5 Sec

Table 3.5: Toll Notification Requirements in Linear Road based on [8].

The Table in 3.5 describes the preconditions, output, recipient and the requirement for the response time for toll notifications. Therefore, some background information will be clarified in the following.
The set of all position reports is denoted by P. The tuple p is the current position report. The prior position report is denoted as $\overleftarrow{p}$. It identifies the position report 30 seconds prior p for the same vehicle. Similarly, $\overrightarrow{p}$ identifies the position report following to p for the same vehicle. Linear Road formalizes these position reports like follows:

$$\overleftarrow{p} = \text{q} \in \text{P s.t.}$$
$$(\text{q.Vid} = \text{p.Vid} \wedge \text{p.Time - q.Time} = 30)$$

$$\overrightarrow{p} = \text{q} \in \text{P s.t.}$$
$$(\text{q.Vid} = \text{p.Vid} \wedge \text{q.Time - p.Time} = 30)$$

Formula 3.5.2: Formalization of position reports in Linear Road based on [8].

A position report is denoted as p. Similarly, a position report for a toll notification is denoted by q. The parameter q.Time belongs to the position report of a toll notification, which will be described afterwards. The first term describes that $\overleftarrow{p}$ is equal to a position report for a toll notification and belongs to the set of all position reports, whereas the vehicle identifier is the same in these reports and the difference of the time

in the position report p and q is equal to 30. The second term is defined similarly. The minute of a second t is calculated by Linear Road like following:

$$\mathrm{M(t)} = \lfloor \frac{t}{60} \rfloor + 1$$

Formula 3.5.3: Formula for transforming a second into a minute based on [8].

Note that the first minute in Linear Road starts with 1. Finally, the function $Last(v,t)_i$ identifies the i^{th} position report that was emitted by vehicle v prior time t.

$$Last(v,t)_i = \mathrm{p} \in \mathrm{P} \text{ s.t.}$$
$$(\mathrm{p.Vid} = \mathrm{v} \wedge (30(\mathrm{i} - 1) \leq \mathrm{t} - \mathrm{p.Time} < 30\mathrm{i}))$$

Formula 3.5.4: Formalization of a certain position i in a position report based on [8].

A position report, q, for calculating a toll notification is defined as follows:

q = (Type: 0, Time: t, VID: v, Spd: spd, XWay: x, Seg: s, Pos: p, Lane: l, Dir: d)

Each time a vehicle enters a new segment without being on an exit lane, namely not lane 4, the precondition for a toll notification is fulfilled and the toll notification is triggered. The output of a toll notification is a tuple that consists of following fields:

- *Type = 0*, the type identifies a tuple as position report if it is equal to 0,
- *Vid*, identifies the vehicle that has to be notified of the toll,
- *Time*, identifies the time stamp when the vehicle emitted the position report q,
- *Emit*, identifies the time stamp when the toll notification was emitted,
- *Spd*, identifies the average speed of the last 5 minutes on a certain expressway, on a certain segment, and a certain direction whereon the vehicle resides,
- *Toll*, is the amount of toll that was calculated for that certain segment.

The recipient of a toll notification is vehicle v. The response time between time t and emit time t' must be equal or below 5 seconds. The fields *Spd* and *Toll* are calculated by functions that are expressed as *Lav* and *Toll*, respectively. Linear Road defines these functions as stated in Table 3.5.5 and Table 3.5.6:

$$\begin{aligned} Cars(m,x,s,d) = \{p.Vid \quad & | p \in P \\ & , m = M(p.Time) \\ & , p.(XWay; Seg; Dir) = (x; s; d)\} \end{aligned}$$

$$Avgsv(v,m,x,s,d) = AVG(\{|p.Spd|p \in P \\ ,p.Vid = v, m = M(p.Time) \\ ,p.(XWay;Seg;Dir) = (x;s;d)|\})$$

$$Avgs(m,x,s,d) = AVG(\{|Avgsv(v,m,x,s,d)|v \in Cars(m,x,s,d)|\})$$

$$Lav(m,x,s,d) = [AVG(\{|Avgs(m-1,x,s,d),...,Avgs(m-5,x,s,d)|\})]$$

$$Toll(m,x,s,d) = \{2 \cdot (|Cars(m-1,x,s,d)| - 50)^2 \\ if Lav(m,x,s,d) < 40 \ and \\ |Cars(m-1,x,s,d)| > 50 \ and \\ \forall_{0 \leq x \leq 4}(\neg(Acc\_In\_Seg(m-1,x,Dn(s,d,i),d))) \\ 0, \ Otherwise\}$$

Formulas 3.5.5: Notation used to define tolls based on [8].

$$Stop(v,t,x,l,p,d) \leftrightarrow \\ \forall_{0 \leq x \leq 4}(Last(v,t)_i.(XWay;Lane;Pos;Dir) = (x;l;p;d))$$

$$Acc(t,x,p,d) \leftrightarrow \\ \exists_{v1,v\,2,l}\,(l = TRAVEL \ \wedge \ v1 \ \neq \ v2 \ \wedge \ Stop(v1,t,x,l,p,d) \ \wedge \ Stop(v2,t,x,l,p,d))$$

$$Acc\_in\_Seg(m,x,s,d) \leftrightarrow \exists_{p,t}\,(t \in m \ \wedge \ Acc(t,x,p,d) \ \wedge [\frac{p}{5280}] = s)$$

Table 3.5.6: Notation used to define accidents based on [8].

The function *Lav(m,x,s,d)* is shorthand for *Latest Average Velocity*. This function calculates the average speed on a given expressway x, segment s and direction d over 5 minutes prior minute m, whereby m = M(t). Therefore, this function makes use of the functions *Avgs* and *Avgsv*. Avgs(m,x,s,d) calculates the average speed of all vehicles within a minute on a given expressway, segment and direction. It could be the case that a vehicle emits two position reports during the same minute. Therefore, Avgsv(m,x,s,d) computes the average speed of that vehicle v, which reports several position reports during the same minute. The amount of all vehicles driving during minute m on expressway

x, on segment s, in direction d returns the function *Cars(m,x,s,d)*.
The notation that is used for p.(Xway; Seg; Dir) = (x; s; d) is shorthand for p.Xway = x, p.Seg = s, and p.Dir = d.
The function *Toll (m,x,s,d)* is based on the speed over 5 minutes, on the number of cars and the proximity of an accident during the minute before (M(t) - 1) on expressway x, segment s and direction d, respectively.
The formula for calculating a toll for a given minute, expressway, segment and direction is calculated if the function Lav (latest average velocity) is below 40 mph, and if the number of cars is above 50 and if there is no accident on certain segments regarding a given expressway, segment and direction, respectively. Otherwise, the toll is set to 0.
If congestion is high, the amount of tolls increases. This should demotivate drivers using these roads and avoid worse congestion. If an accident was detected at most 4 segments downstream of the current segment s during minute M(t), no toll will be charged as described in Table 3.5.5 and Table 3.5.6.

Accident detection works like so: If a vehicle v emits four consecutive position reports while being on the same expressway, lane, position and direction, then a vehicle is considered as a stopped car. An accident exists if at least two different vehicles are considered as stopped within the same time being on a travel lane. The function *Acc_in_Seg(m,x,s,d)* returns a segment whereupon an accident exists during a given time. If this segment is at most 4 segments downstream of segment s, no toll will be charged.

3.6 Running the Linear Road Benchmark

The Linear Road Benchmark evaluates the performance of stream data management systems. The benchmark only runs on Linux-based operating systems. System and software requirements that have to be met, e.g., PSQL and Perl, can be found on the website of Linear Road in [9]. Download tools for running the Linear Road Benchmark can be found on the website as well. In this section, this will be presented in detail. Linear Road provides following download tools: a *Data Generator*, a *Data Driver*, and a *Validator*.

The Data Generator produces data for the Linear Road Benchmark. This is realized with a simulator, namely, the MIT Traffic simulator (MITSIM)("based on Q.Yang" [10]). MITSIM simulates traffics on different lanes with a number of vehicles driving on these. Thereby, it produces input data for the Linear Road Benchmark. The generated data is stored in flat files. The generated flat files can be distinguished: One set of flat files the generator produces is historical toll data with a summarized tolling activity over 10 weeks prior the simulations run. Another set of files that is produced, contains streaming input data that will be needed for continuous query requests.
In the generator, the number of expressways that MITSIM should generate has to be determined. Starting from 0.5, the scale factor can be increasingly large. It has to be considered that 512 MB RAM and approximately 3 Gigabytes of memory

per expressway are needed. The running time of MITSIM is between 3 to 5 hours per expressway depending on the computer speed. For this project the scale factor has been set to 1.

The Data Driver is used for the implementation. It provides the data to the system, that implements Linear Road, as it becomes immediately available. The data driver is written in C/C++. It takes the file generated by the *Data Generator* and sends the data to the Linear Road application. This is done in such a way just as the time stamp is consistent with the delivering. Thereby, a *data provider*, that is contained in the Data Driver tool, reads the input data and puts it in its own buffer. If the Data Generator is called by a system that implements Linear Road, then it delivers only the data that can fulfill the time stamp. Each tuple is transformed to a string; strings are sent and not the single values of the tuples. Repeatedly, the driver sleeps randomly between 5 and 15 seconds and then returns the data that is consistent with its timestamps. The maximum number of tuples that can be returned at once when the driver wakes up, is 100. The default buffer size of the provider is 10000 tuples, which is consistent with 30 seconds.

The Validator is used for checking whether Linear Road has been implemented correctly and whether the response time requirements were met by the system.
Before starting the validator, a new database has to be created. This is done with PSQL, which provides open sourced databases. The validator includes a number of Perl-scripts. It requires a file generated from the generator, which contains the data being the same for the SDMS, and the output of the implementation for the toll notification, the daily expenditure, the account balance and the travel time estimation. The validator takes the file that has been generated by the data generator as input file and implements the functions for its own. The output is stored in the database that has been generated before. The name of the database, as well as the owner, password and the working directions have to be specified in the configuration file of the validator. It then compares the values stored in the database with the output files that contain the output of the implementation. The order of the tuples in the output does not play a role. The validator is started on the console. For a toll notification, firstly, the tuple time and vehicle identifier are compared, then the speed, and the toll. If the output has no deviations, then the validator prints success information on the console. Otherwise, it prints failure information. In the case of a failure, either it prints that the validator contains data, that the output files do not contain, or it prints that the output files contain data, which the validator did not generate. So, in the case of a deviation, the fault may be detected. Linear Road provides some result queries that can be used to see the deviations. In the case of the toll notification, you can request the query that shows the toll alerts that the validator generated but the output file did not generate. On the opposite, you can request the query that shows toll alerts that the output file contains but the validator did not generate. Moreover, the number of toll alerts can be compared if requested. Finally, it is possible to see the single values

that are stored in the tables, which depend on the input file. For the case of a file containing a 3 hour simulation, the validator has a running time of approximately 2 to 3 hours.

4 Implementation

The following section deals with the implementation of the toll notification of Linear Road based on Storm. Instead of a database, Maps are used to realize the temporary storage. Thereby, the implementability, efficiency and the accuracy of this implementation, based on Storm, will be observed. However, the aspect of the parallelism will not be observed for now. For the implementation, Storm 0.8.2 is used, this has only been compatible with Java 6. The environment is a Ubuntu-Virtual machine with 4 GB RAM. In this chapter, the structure of the topology and the interaction between the spout and bolts will be explained. The project consists only of one topology, which is supposed to run in local mode.

4.1 The Topology

The topology consists of one spout and twelve bolts. The main function is contained in the topology. In the main function, a topology is built with a *Topology-Builder*, whereas the function is provided by Storm. With that, not only the arrangement of spout and bolts is determined, but also the subscription of bolts to certain streams. Then, a local cluster is built so that the topology can be pushed into this cluster. Also a configuration is built so that the default settings of Storm can be overwritten. Depending on the length of the input data, the time that Storm has to wait until it should shutdown, has to be configured. Afterwards, the Data Driver, that has been presented in Section 3.6, is started. The Data Driver passes the data to a socket. As soon as the configured time has passed, the cluster is shut down. Each spout and bolt emits tuples. A stream is a sequence of tuples. The streams in this topology use the shuffle grouping, which means that each bolt gets an equal number of tuples. A short example the shuffle grouping and the subscription looks like so:

```
(1) TopologyBuilder builder = new TopologyBuilder();
(2) builder.setSpout("tuple-reader", new TupleReaderSpoutLR());
(3) builder.setBolt("normalizer", new NormalizerLR()).
(4) shuffleGrouping("tuple-reader");
```

Firstly, in line (1) you see the creation of the topology. The definition of a spout consists of a string field for identifying a stream and the constructor of a component, which is in this case a spout. In line (3), a bolt is defined with the corresponding component. The stream of this bolt is called "normalizer". Line (4) determines the subscription to the stream called "tuple-reader" and the form of the grouping, namely shuffle grouping.
Notice that the more nodes a topology contains, the more the timeout for the messages in that topology has to be increased in the configuration, as the default timeout is 30 seconds. Figure 4.1 shows the topology for the toll notification and how the data stream is passed between the spout and bolts.

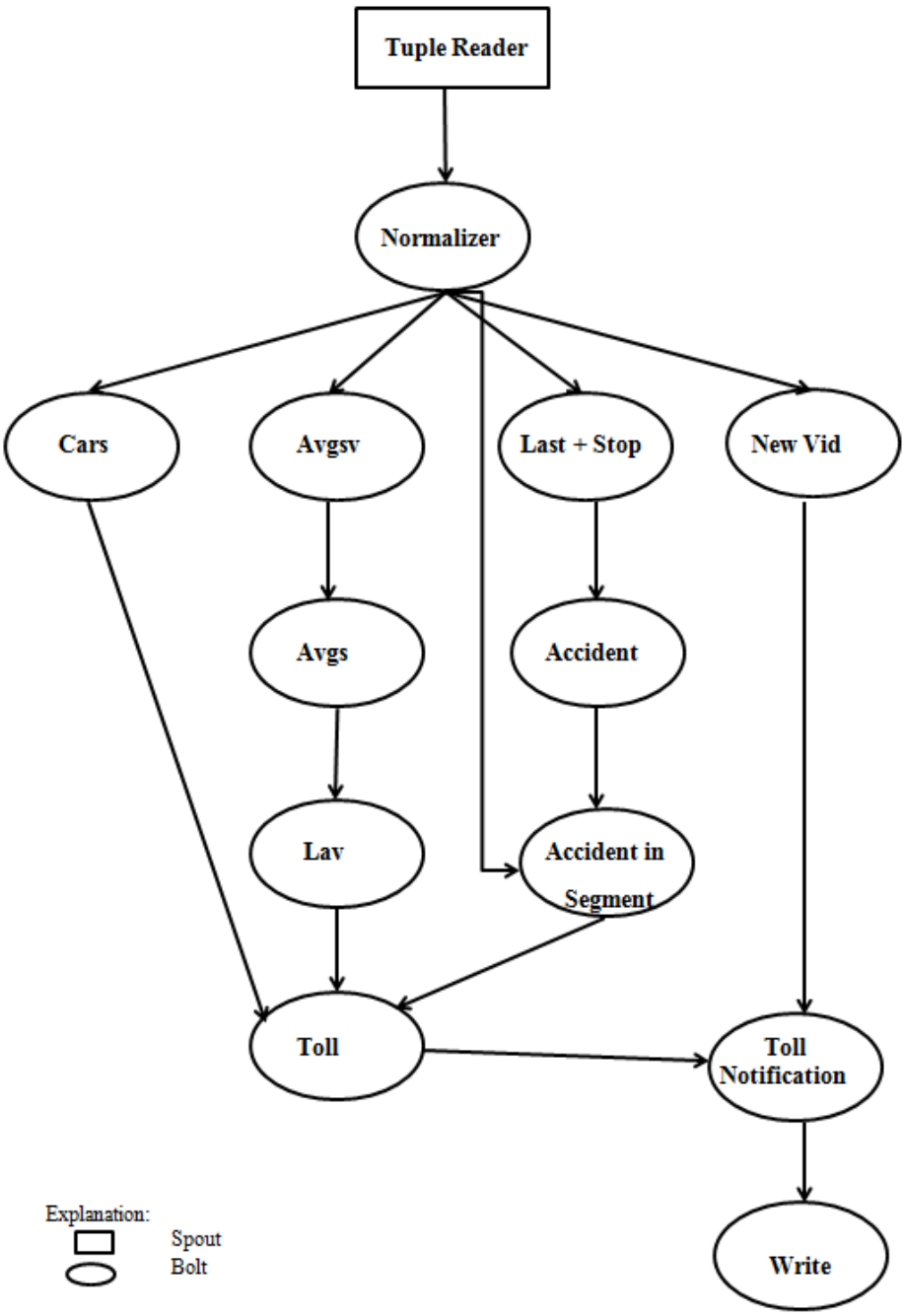

Figure 4.1: An Illustration of the Topology for calculating a Toll Notification.

The process logic of the spout and bolts will be explained in detail in Section 4.2 and Section 4.3. Therefore, in the beginning only the flow of streams will be explained. In Figure 4.1, only one spout is defined, namely the *TupleReader*. It makes the input stream available to other components in that topology, provided that they are subscribed to the stream. In this case, a bolt called *Normalizer* subscribes to the stream out of the *Tuple Reader*. The bolts that implement the functions *Cars*, *Avgsv*, *Last* and *Stop*, and the bolt that searches for new vehicles, namely *New Vid*, subscribed to the stream out of the *Normalizer*. The bolt that calculates the average speed (*Avgs*) subscribed to the stream out of *Avgsv*. The bolt called *Lav*, gets the stream out of *Avgs*. Similarly, the bolt that detects *Accidents*, subscribed to the bolt that implements 2 functions, namely *Last* and *Stop*. *Accident in Segment* gets streams out of 2 bolts, which are *Last and Stop*, and *Normalizer*. The bolt that calculates *Tolls* gets streams out of 3 bolts, namely *Cars*, *Lav*, and *Accident in Segment*. The result of that bolt is passed to the bolt that is responsible for the *Toll Notification*, because it subscribed to this stream. Moreover, the stream out of *New Vid* is passed to it. Finally, the bolt that writes the output of the toll notification, namely *Write*, subscribed to *Toll Notification*.

4.2 The Spout

As mentioned in Section 3.6, the *Data Driver* passes the input data to the SDMS. The functionality of this spout is to connect to the socket, to which the Data Driver sends the data, and to read that data. Therefore, the spout extends *BasRichSpout*, an abstraction provided by Storm. With this, two functions are provided called *ack* and *fail*. The function *ack* is executed for every tuple that the spout has emitted, whereas the corresponding message identifier must have been fully processed. Otherwise, if this fails, the function *fail* is called, that shows the message identifier that has not been fully processed. If this function is called, the message will be replayed. Furthermore, the spout contains following functions: *open*, *close*, *nextTuple*, and *declareOutputFields*. The function *open (Map conf, TopologyContext context, SpoutOutputCollector collector)* is called when a task of this component is initialized. The parameter *context* contains the topology data, *conf* is created in the topology and *collector* is needed to emit tuples. The spout connects to the socket and declares the collector, which will be used in the function *nextTuple* to emit tuples. *Close* is called when the spout is about to shutdown. In *nextTuple*, the stream that came out of the socket, is read with the help of a *scanner*, a class provided by Java. This scanner is read line by line. The data driver has sent a tuple as one string; therefore, one line is in this case is one string. Each string is then emitted, with the help of the collector, to an output field. Because the toll notification needs the emit time, a function after emitting a tuple is called, which measures the time from then. The function *declareOutputFields* declares the fields for the output. This means that the bolt emits the fields determined by the function. The function *nextTuple*, as well as the functions *ack* and *fail*, are executed from the same loop. When emitting values, elements of an Array List are passed to the constructor. The constructor is called when defining a component in a topology. Output fields define the order and the fields that will be

emitted. The entire stream is defined in the topology (-class) as a string field, as already mentioned.

4.3 The Bolts

The topology contains 12 bolts. Most of these bolts make use of maps, which Java provides. Following bolts are used to implement the toll notification of Linear Road:

- NormalizerLR, returns the normalized the input
- CarsBoltLR, returns the number of cars
- AvgsvBoltLR, returns the average speed of a vehicle
- AvgsBoltLR, returns the average speed during a minute
- LavBoltLR, returns the average speed over the last 5 minutes
- AccBoltLR, returns an accident
- AccInSegBoltLR, returns the segment on which an accident occurs
- TollBoltLR, returns the toll to be charged
- TollNotificationBolt, returns the output of a toll notification
- WriteBolt, writes the output into a file

Additionally, there exists a class called *TimeComparator* for sorting the tuples by time. The bolts will be explained in such an order as shown in Figure 4.1, beginning with the NormalizerLR. The typical structure of these bolts is that every bolt extends *BaseBasicBolt*, an abstraction provided by Storm. This supplies two important functions. One is *declareOutputFields (OutputFieldsDeclarer declarer)*, that has been already explained in Chapter Section 4.2, and the other function is *execute (Tuple input, BasicOutputCollector collector)*. In this function, it is possible to get the input that the bolt subscribes to and to emit even new streams with the parameter collector. This function is called whenever a tuple receives that bolt. In each bolt, the calculations are performed in functions, that are called in the *execute* function, typically, for deleting old hash maps.

NormalizerLR This bolt normalizes the input string. It subscribes to the stream out of the spout. The function *execute* is called if a tuple, that the spout delivers, receives this bolt. In this case, the spout emits data string by string. This bolt receives a string and splits it by a comma, because the data driver parsed the generated values of a line from the Linear Road Benchmark to a string. The order of the single values is changed from a position report p to a position report q, for calculating the toll notification. The elements are parsed to an Integer type and added to a list, that is transformed to an array. This array is then emitted. After each emit

of the bolt, the list is cleared for emitting only the elements per line. Therefore, nine output fields are declared in the form of a position report q, according: type, time, vid, speed, xway, seg, pos, lane, and dir.

CarsBoltLR This bolt counts the number of cars driving during a minute on the same expressway, the same segment, and the same direction. The bolt receives the stream out of the *NormalizerLR*, tuple by tuple. In the function *execute*, another function is called that implements the actual function. In this function, following happens: The time, which is measured in seconds, is transformed to the corresponding minute in that bolt, for example, the tuples that contain the time stamp from 0 to 59 belong to minute 1, tuples containing the time stamp from 60 to 119 belong to minute 2, and so on. This transformation rule (M(t)) has been defined by Linear Road (Formalization 3.5.3). Moreover, a *Hash Map* is used. Such a map contains a unique key and a corresponding value. Therefore, the parameters for the expressway, segment, direction, and time(the calculated minute) are parsed to strings and unified to one string. Thereby, a colon, that is parsed to a string, is replaced between two strings, such that it is possible to split the string later by a colon. This string is a key used in the map. If the map does not contain this key, then the key is put to the map with the corresponding value 1. Otherwise, the corresponding value is increased by 1. This procedure is continued until a new minute starts. If this happens, a new map will be generated, that copies the values of the old hash map. The new generated map is returned to the *execute* function and the old map is cleared. The returned map is passed through, whereas each key is split by a colon. Therefore, the time, expressway, segment, direction, and the corresponding number of cars driving in this area, are emitted in the *execute* function of that bolt.

AvgsvBoltLR This bolt also subscribes to the stream out of the *NormalizerLR*. The task of that bolt is to calculate the average speed of the same vehicle that drives during the same minute, on the same expressway, the same segment, as well as the same direction. This is also realized with a hash map, whereas a key consists of the string unification of the vehicle identifier, the expressway, the direction, the segment and the minute. If the map does not contain the key, then it is put to the map with the corresponding speed value of those parameters. Otherwise, the speed that is contained in that map is summed up to the new speed and divided by 2, to calculate the average speed. This new calculated value replaces the old speed value in the map. The procedure is also continued until a new minute starts. Then, the map is copied to a new generated map that is returned to the function *execute*. When the map is returned, it is passed through, whereas the single values are emitted. These values accord: the vehicle identifier, the time (minute), expressways, segment, direction, and speed.

AvgsBoltLR This bolt gets the emitted tuples from *AvgsvBoltLR* and returns the average speed of a certain expressway, segment, and direction during the same minute. Here, the same procedure is used as in *AvgsvBoltLR*. During the same minute, a

hash map is used. The key consists of a string unification of expressway, segment, direction, and minute. The corresponding value is a list that adds the speed values of the same keys. If a new minute starts, then for every key during the prior minute, the corresponding list is passed. Thereby, the values are summed up and divided by the number of elements contained in the list. The key with the calculated average speed is put to a new map and returned to the *execute* function. The emitted values are: expressways, direction, segment, average speed and time.

LavBoltLR This bolt gets the data from *AvgsBoltLR* and returns the average speed of the last 5 minutes on a certain expressway, segment, and direction. The procedure looks like following: the bolt receives the input tuples that come from *AvgsBoltLR*, tuple by tuple. A function, that is called in the *execute* function, returns a map, which contains a string key, consisting of the unification of an expressway, segment, and direction. The corresponding value is a list that contains the corresponding average speed. If a key already exists in the map, the value will not be overwritten, but rather added to the list. If a new minute starts, a new generated map, that has copied the content of the map, is returned, such that the the old map can be cleared.
The returned map is copied to another map that does not consist of the data for only one minute, but also of the following minutes. Moreover, another function calculates finally the average speed of the last 5 minutes with this map and a parameter that accords to the time. Since the first minute starts with 1, and the average speed of the last five minutes are calculated in form of $m-1$, ..., $m-5$, for the first minute, the value for the latest average velocity is -1.
Special cases cover the cases for calculating minutes below 5. For example, for the minute 3, the average speed during minute 1 and minute 2 for the same expressway, direction and segment have to be calculated. Otherwise, the average speed of the last 5 minutes prior minute m are calculated. To sum up, when calculating the latest average velocity, a map is given and certain time. This map consists of and integer key, which represents the minute, and a corresponding sub-map. For the required past minutes, which can be at most 5, the sub-maps are spitted. A sub-map consists of a string key, which represents the area (string unification consisting of expressway, segment, and direction), and a corresponding value, which consists of a list. This list contains the average speed for the corresponding area. The average for all speeds of the certain minutes for a certain area is calculated and returned to the *execute* function. This bolt then emits the time, expressway, direction, segment and the latest average velocity.

LastBoltLR This bolt implements two functions of Linear Road, which are $Last(v,t)_i$ and Stop(v,t,x,l,p,d). It subscribes to the stream of the *NormalizerLR*. Firstly, the position reports for every vehicle have to be saved. As mentioned, a vehicle emits every 30 seconds a position report, which is guaranteed by the traffic simulator in Section 3.6. Therefore, the values of a position report are stored in an integer array. In this bolt, 2 hash maps are used. One map stores the data of the first coming

position reports, which start by the time being equal to 0 until 29 (in seconds). The other map stores the other position reports during the same minute, which are then arrays with time being equal to 30 until 59 (in seconds). These two maps are compared by the vehicle identifier. If the vehicle identifier is the same, then the expressway, lane, position, and direction will be compared. If these values are equal, respectively, then a map collects the position reports for that certain vehicle. If the size of the map is greater than four, then there exists a stop and the position reports are returned. If a stop exists, then the values of a position report will be emitted by this bolt. Otherwise, all values that will be emitted will be set to -1, except the time.

AccBoltLR This bolt gets the tuples emitted by *LastBoltLR* and makes as well use of a map. If this map does not contain the received vehicle identifier, this identifier is put into the map with the corresponding position report. The position report is stored in an integer array. If the size of the map is greater than 2, the the corresponding position reports will be analyzed. If at least 2 position reports contain the same area, namely same expressway, segment, direction, lane, and position, respectively, with different vehicle identifiers at the same time, then there exists an accident. If this is the case, then the bolt emits the time, expressway, lane, direction and the position. Otherwise, all values are set to -1, except for the time.

AccInSegBoltLR This bolt subscribes to two streams. One stream comes from the *NormalizerLR* and the other comes from *AccBoltLR*. The single tuples can be distinguished by the size of the tuples. For example, if the input size is equal to 5, then we know that this tuples comes from *AccBoltLR* because it declared five output fields in its bolt. Otherwise, if the input size is equal to 9, then the tuple comes from the *NormalizerLR*. If the elements of the tuple, received from *AccBoltLR* are different from -1, then the segment downstream of that accident is calculated, which can be at most 4 segments apart. Otherwise, the values are set to -1. There are two maps, whereas one map consists of the time (in minutes) and a list of values, which are equal -1 or reside between 0 and 99, that would be the segment being 4 segments downstream an accident, and the segment on which the accident occurs. The other map consists also of the time (in minutes) and a list of all segments being driven during this minute. If the vehicle, being on a certain segment, resides between the segment downstream of the accident and the accident, then there exists an accident. This case is checked for both, westbound and eastbound. If this is not the case, then there exists no accident on that segment. If there exists an accident, then this bolt emits the time, expressway, the segment on which an accident occurs and the direction. Otherwise, the time is emitted with the other values being -1.

TollBoltLR This bolt consumes three streams. It subscribes to streams out of *AccInSegBoltLR*, *CarsBoltLR*, and *LavBoltLR*. Again, the types of the tuples are distinguished by their size. After an incoming tuple is categorized, it will be stored in a map for its own category. Therefore, we have three maps; one for the number of cars driving during a certain minute on a certain expressway, segment and direc-

tion(map1), one for the latest average velocity on a certain expressway, segment and direction for a certain minute(map2), and finally one map for storing certain expressways, accident segments, and directions for a certain minute(map3). Some of the values of the latter map could be almost all -1, since not on every segment an accident can occur during a minute. The data for a certain minute is fetched of the maps. Therefore, if there were no accidents during a minute, at least the time has not been set to -1. Passing through the map that saves the data of the number of cars, every expressway, segment and direction is compared with the other 2 maps. Thereby, the corresponding data is fetched and the toll is calculated for that. This means, that for a certain minute m, the corresponding data in these maps are fetched. Then, for each expressway, segment, and direction of map 1, the corresponding values of the other 2 maps are fetched. For each of these areas, namely expressway, segment, and direction, the toll is calculated. If the number of cars is greater than 50, and the latest average velocity is smaller than 40, and there exists no accidents on the segments downstream the current segment, which can be at most 4 segments, then the toll is calculated like following:

$$Toll = 2 \cdot (|Cars(m-1, x, s, d)| - 50)^2$$

If the requirements cannot be fulfilled, then the toll is 0. This bolt emits the time (in minutes), expressway, segment, direction, toll, and the latest average velocity.

NewVidLR This bolt checks whether a vehicle has entered a new segment without driving on an exit ramp. The input tuples come from the *NormalizerLR*. The elements of a tuple are stored in an array of type integer. The temporary storage is also realized with maps. A function called by the *execute* function, is responsible for discovering new vehicles. A variable helps for checking the time interval from 0 to 29 (seconds). A map collects the arrays that contain time stamps suiting in that time interval. The key of type string of this map consists of the unification of the vehicle identifier and the segment of the array. The corresponding value is the integer array, containing the elements of a position report. During the collection, each array is immediately returned to the *execute* function and emitted. If the map should already contain the key, which should not be possible if Linear Road generates the correct data, then the array is only stored in a map and will not be returned. So, for the seconds from 0 to 29, all arrays are stored in a map. For the following 30 seconds, it is checked whether the key (unification of the vehicle identifier and the segment) of the incoming array is already contained in the map. If this is the case, then another map(map2), stores this array. Otherwise, the array is not only stored, but also returned. Therefore, it is checked if the vehicle identifier and the segment are the same as the one in the prior position report, which are stored in the first map. If also these 30 seconds are over, which accords that the incoming tuple contains a time stamp being equal to 60(seconds), then the first map(map1) will be cleared. Afterwards, the content of map2 is copied to map2, such that map2 then can be cleared. Thereafter, the time period is shifted, such that the following position reports can be processed. Therefore, the procedure

will be the same for the next incoming arrays. Before emitting the elements of an array it is checked whether the type is equal to 0 and the lane is not an exit lane. Therefore, the lane must not be equal to 4. If these conditions are fulfilled, then this bolt can emit this position report, whereas the values accord: type, time, vehicle identifier, speed, expressway, segment, position, lane, and direction.

TollNotificationBolt Finally, this bolt computes the toll notification. Therefore, this bolt subscribes to two streams that come from *NewVidLR* and *TollBoltLR*. Therefore, the input is categorized to distinguish the tuples. This is done with checking the tuple sizes of the input. Every vehicle has to be notified of the toll on a certain expressway, a certain segment, and a certain direction. In this bolt, the time is stopped, which started in the spout. Every tuple is stored in a corresponding map, whereas there exist 2 of them. For the same minute, the expressway, segment, and direction will be compared. Following corresponding values will be emitted: type (Type = 0), the vehicle identifier, the time (in seconds), the emit time (the stopped time in seconds), the speed (the latest average value in mph) and the toll, for the expressway, segment, and direction, respectively.

Write Bolt This bolt subscribes to the stream out of *TollNotificationBolt*. It only writes the tuples it receives into a text-file. The tuple consists of the type (Type = 0), the vehicle identifier, time (in seconds), the emit time(in seconds), the speed (latest average speed) and the toll. Therefore, the elements of a tuple are unified to a string, whereas between two elements a comma is added, which is also parsed to a string. This bolt allows to get the data out of Storm so that it can be checked by the validator of the Linear Road Benchmark.

5 Evaluation

In this section, the results of the implementation of the toll notification, based on Storm, will be discussed.

This project has been realized on a laptop, in the environment of a Ubuntu - Virtual machine with 4 GB RAM and 1.73GH. Storm 0.8.2 was only compatible with Java 6. Moreover, the number of expressways has been set to 1, which corresponds to an L-rating = 1.

The text file, that contains the output of the toll notification, is checked by the validator of the Linear Road Benchmark. In the beginning, the validator printed success information, despite the emit time is not meeting requirements for the response time about the difference being at most 5 seconds. Even when the difference between the emit time and the tuple time was manipulated for test cases by making the the difference greater than 60 seconds, the validator still printed success information. Therefore, the response time of this SDMS could not be checked perfectly by the validator. The validator originates from 2005 and can therefore be regarded as out-dated.

In the early stages of the output of the implementation, the difference between the emit time and the tuple time is approximately 5 seconds:
Periodically, the emit time approached and distanced to the tuple time. The amount of the response time, which is the difference between the emit time and the tuple time, resides between -1 to 10 seconds. In the beginning, the tuple time is 0 seconds, and the emit time is 5 seconds. After approximately 20 minutes have elapsed, the computation stopped. Hence, for testing a big amount of stream data, a better computer performance is preferable. It could be the case that maps can be unfitting when testing big stream data because the streams increase and therefore a hash collision might occur, but this has not been tested yet.

The emit time is different when starting the system call before and after reading tuples out of the scanner. When the call is started before reading the tuples, as it is shown in Figure 5.1, the emit time is mostly greater than the tuple time. But when starting the system call between when a tuple is read and when a tuple is emitted, then the emit value is mostly less than the tuple time, and the response time reside so between -14 and 1. Hence, the first option has been chosen.
Figure 5.1 shows the development of the emit time and the tuple during a minute.

The Uppsala University published a master thesis dealing with the implementation of the Linear Road Benchmark for the SCSQ, which is another SDMS("based on Marten Svensson" [3]). This implementation is realized with a database, whereas an L-rating of even 1.5 could be reached. Moreover, the maximum response time for an L-rating = 1

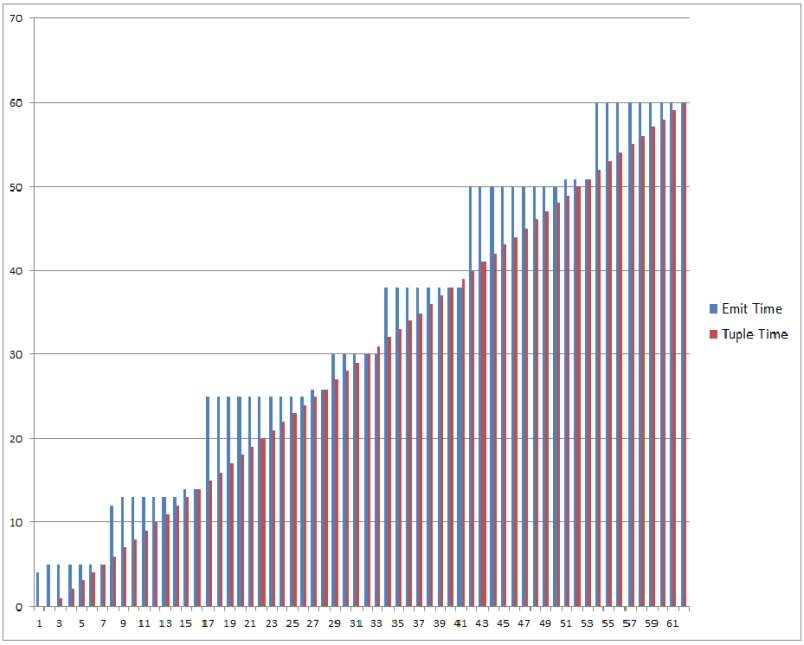

Figure 5.1: The Development of the emit time corresponding to the tuple time

is 1.09 seconds, which is much less than the maximum response of this implementation, which can be seen when looking at the distance between the emit time columns and the tuple time columns.

Hence, the toll notification is implementable and the accuracy of this implementation is correct. However, the response time could be improved.

6 Conclusion and Future Work

The aim of this work was to evaluate the accuracy and the efficiency of Storm, a stream data management system, by implementing a query for toll notification of Linear Road without using databases for temporary storage.

Storm is capable of processing data-streams in real-time and producing output streams. The main components, namely the topology, the spout, and bolt were presented, whereby the interaction of these has been illustrated. Also, a cluster, consisting of daemons of Storm, namely Nimbus and Supervisor, were shown. These are components that work in the background of Storm and do the actual computation. Additionally, the concept of the parallelism in Storm has been shown, but not performed yet.

In this implementation, only the query for toll notifications of Linear Road is computed, since this query is based on continuous data. Linear Road simulates a fictional urban area, which is called Linear City. Therefore, the structure of Linear City is described. The stream data, that has been generated by the Linear Road Benchmark is introduced. Thereby, the formalization and conditions were pointed out. Requirements for Linear Road were also pointed out.
Next, the way how toll notifications should be processed is summarized. The formulas and the conditions were introduced. Hence, tolls had be calculated and certain vehicles had to be found out.

Afterwards, the concept of the implementation of the query for toll notifications has been presented. Commonly, the temporary storage of those data streams was realized with a database. This type of storage was replaced with maps, provided by Java. There has been only one topology implemented. Within this topology, there is only one spout, that imported the stream data into Storm. The query for the toll notification has been distributed to twelve bolts.

The output of the implementation has been checked by the validator of the Linear Road Benchmark. Meanwhile, the implementation is accurate, but the response time could be better, regarding the results published by the Uppsala University [3].
Despite the emit time did not meet the requirements perfectly, the validator printed success information.

In this work, following problems attracted attention:
It could be the case, that the hash maps collide in future, when the stream data increases. In this project, the scale factor has been set to 1. Other scale factors should be tested as well. Moreover, the implementation could be computed on a PC with a good computer speed, in which the operating system is not based on a virtual machine. Because Storm can process a big amount of tuples quite quick, it should be tested whether the response time is less when using a big database.
Furthermore, the Data Driver could be optimized, which has not been tested yet. Currently, the Data Driver sleeps randomly between 5 and 15 seconds before sending data.

This time could be shortened. Analogously, the buffer size in the Data Driver should be increased, when the amount of tuples get large.
In addition, the parallelism in Storm could be implemented. Thereby, the number of Workers and the number of Tasks could be increased. Finally, *Apache Kafka* [11] could be tried out in combination with Storm. Kafka follows to the concept of a producer and a consumer. Thereby, the producer takes the stream and immediately pushes it to the consumer, which is done with the help of queuing messages. Since Kafka has a high throughput, this could be tried out as well; Therefore, Kafka could be set in between the Linear Road Benchmark and Storm; however, this has also not been tested yet. Due to the limited scope, this is left to be done as future work.

Bibliography

[1] N. Marz, "The Storm website." `http://storm.incubator.apache.org`, 2014.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: A new model and architecture for data stream management," *The VLDB Journal*, vol. 12, pp. 120–139, Aug. 2003.

[3] M. Svensson, "Benchmarking the performance of a data stream management system marten," Master's thesis, Uppsala University, 2007.

[4] J. Leibiusky, G. Eisbruch, and D. Simonassi, *Getting Started with Storm*. O'Reilly Media, Inc., 2012.

[5] "Real-time streams and Logs website." `http://pixelmonkey.org/pub/streams/notes/`, 2013.

[6] "The Zookeeper website." `http://zookeeper.apache.org`, 2010.

[7] "The Zeromq website." `http://zeromq.org`, 2014.

[8] A. Arasu, M. Cherniack, E. Galvez, D. Maier, A. S. Maskey, E. Ryvkina, M. Stonebraker, and R. Tibbetts, "Linear road: A stream data management benchmark," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pp. 480–491, VLDB Endowment, 2004.

[9] "The Linear Road website." `http://www.cs.brandeis.edu/~linearroad/index.html`.

[10] Q. Yang, "A Microscopic Traffic Simulator for evaluation of dynamic traffic management systems," *Transportation Research Part C: Emerging Technologies*, vol. 4, pp. 113–129, June 1996.

[11] "The Apache Kafka website." `https://kafka.apache.org`, 2014.

List of Figures