University of Zurich^UZH

A Distributed Engine for Processing Triple Streams

Master Thesis Dec 11, 2012

Thomas Hunziker
of Zurich, Switzerland

Student-ID: 07-704-844
thomas.hunziker.87@gmail.com

Advisor: **Lorenz Fischer**

Prof. Abraham Bernstein, PhD
Institut für Informatik
Universität Zürich
http://www.ifi.uzh.ch/ddis

Acknowledgements

I like to thank Professor Abraham Bernstein for giving me the opportunity to write my master thesis at his group. I want also to thank Manuela Züger for giving me feedback on my thesis. I would like Philip Stutz to give me some insights on how he employs the cluster for his work. I also want to thank Hanspeter Kunz for helping me with the setup of Storm on the cluster. I want to thank the DDIS group to provide me with resources on the cluster. I would like to thank the Storm community and especially Nathan Marz for providing the Storm framework and for answering crucial questions about Storm. Last but not least I like to thank Lorenz Fischer for guiding me through my work and giving me the opportunity to work with him on his project. It was always pleasant to work with him.

Zusammenfassung

Die Geschwindigkeit mit der Daten produziert werden übersteigt die Geschwindigkeit mit der neuer Speicher produziert wird [5]. Um die erzeugten Daten nutzen zu können, müssen diese Daten in Echtzeit als Datenstreams verarbeitet werden. Parallel zu dieser Entwicklung werden die Daten immer mehr im semantischen Web gespeichert, dies erlaubt die Kombination der Daten in neuartiger Weise.

Diese Arbeit zeigt eine Implementierung mit horizontaler Skalierung, die in der Lage ist Triple Datenstreams zu verarbeiten mit dem Storm Framework. Die Arbeit evaluiert das System gegen ein Datenset von 160 Millionen Triple mit einer unterschiedlichen Anzahl Maschinen und Prozessoren.

Abstract

The rate at which the data is produced overhaul the rate at which new storage capacity is produced [5]. To use all the created data it must be processed near realtime as data stream. In parallel the data is stored more and more in the sematic web, which allows the combination of data in new ways.

This work shows an implementation with a horizontally scaling, which is capable to process triple stream data with the Storm framework. The work evaluates the system against data set of about 160 million triples with different number of machines and processors.

Table of Contents

1

Introduction

The invention of computers and the Internet introduces new age of data generating and processing. The estimated growth rate of processing power of general-purpose computer is 58% per year [8]. The potential bidirectional telecommunication grows at rate of 28% per year [8]. The stored information grows at rate of 23% [8].

In 2007, the produced data exceeds the available new storage for the first time [5]. But not all of the produced data is stored, because it is only replicated from one device to another or the data is processed in realtime before it is stored.

The filtering and aggregation of big data is key to success in various disciplines, such as high frequency trading, evaluation of search requests or even for the Large Hadron Collider (LHC) project at the CERN[11]. Over the years various tools have been developed to solve the processing task. Examples in this area are Hadoop (see section 2.2) or noSQL databases, such as MongoDB [1].

1.1 Motivation

Parallel to the growing data creation rate the data is stored more and more in the semantic web. The sematic web links the data items in a massive graph. The graph data is stored in triples of subject, predicate and object. Systems that process streams of graph-triples are proposed and they seem to be effective. However, the proposed systems do not show large-scale qualities. They are not designed with a distributed setting in mind and without horizontally scaling capabilities.

Systems with horizontally scaling qualities like Hadoop do not provide functionalities for processing realtime stream data. Hence they are not suited to process triple streams.

1.2 Description of Work

This work implements and evaluates a distributed realtime triple stream-processing engine using the Storm framework. The engine is called KATTS. KATTS stands for "KATTS is a triple torrent sieve". Storm is a realtime-processing framework based on ZeroMQ. ZeroMQ is a high-performance asynchronous message queue (MQ) written in

[1]http://www.mongodb.org/

C++. By using a MQ we expect a horizontal scaling, because the data can be grouped based on certain criteria and distributed in the cluster.

KATTS implements an approach to evict query solutions to maintain stable memory consumption. To evaluate jobs KATTS provides an evaluation framework, which measures various properties of the system such as the messages sent between nodes, the throughput and the speedup.

This work investigates ways to communicate local information about the load between the nodes and how the system can react on this information at runtime.

1.3 Outline

The remainder of this work is organized as follows: Chapter 2 gives an overview of existing applications and systems that can process triples or can execute jobs in a distributed manner such as Hadoop. Chapter 3 describes the implementation of the distributed triple processing engine and introduces the Storm framework. The evaluation chapter covers the results of executing test queries with KATTS in a test environment with different setups. Chapter 5 leads to a conclusion and presents future work to improve the findings shown in chapter 4.

2

Related Work

The data in the semantic web is usually described with the Resource Description Framework (RDF). The data is queried with the Protocol And RDF Query Language (SPARQL) or with extensions of it. In the field of stream reasoning, stream processing and distributed processing several approaches were developed over the past years.

The following sections cover systems that provide similar functionalities to KATTS. The presented temporal and stream reasoning systems provide similar functionalities but not in a distributed setup. Hadoop is an implementation of the MapReduce model with horizontal scaling properties, but without realtime stream processing capabilities. Hoeksema and Kotoulas present in their work a stream reasoning system based on the S4 framework, which provides similar functionalities as the Storm framework.

2.1 Temporal RDF and Stream Reasoning

Perry et al. propose an extension to SPARQL for geospatial and temporal (SPARQL-ST) data. They provide the syntax for the extension and a prototype implementation of query engine. The temporal part of the extension allows the querying of time-based data with expression filters. SPARQL-ST is not designed for stream data. Hence stream related operators are not supported. The prototype query engine employs a relational database [12], which leads to a total different architecture as expected for KATTS.

Barbieri et al. propose Continuous-SPARQL (C-SPARQL), which provides the syntax for querying triple streams. They introduce the concept of sliding windows, time-based aggregations and filters to SPARQL. C-SPARQL is missing from the possibility to join triple streams over complex time constraints. C-SPARQL allows only the stream joining at the same time, but it does not allow the joining with a certain time shift or even over complete different time restriction [3]. However, C-SPARQL provides a good starting point for defining aggregations and filters.

Anicic et al. propose Event Processing SPARQL (EP-SPARQL). EP-SPARQL tries to fill the gap between event processing (EP) and reasoning on background knowledge. EP-SPARQL provides syntax for filtering, aggregation and joins. The join supports also complex time joins expressions. However the prototype implementation of Anicic et al. runs only on one machine; hence the throughput is about ten thousand triples per second [2], which is below our expectation.

2.2 Hadoop

Hadoop[1] is an open-source programming framework to process large-scale data sets in a distributed setup. It provides the Hadoop Distributed File System (HDFS) and an implementation of the MapReduce programming model. The HDFS is used to store and load the data. Hadoop provides with the MapReduce model an approach to massive scale.

Google originally proposed MapReduce for their web search infrastructure. It works as follows [10]:

1. Iterating over the input data.
2. Map each input item to a key and a value.
3. Partition the values into groups.
4. Iterating over all groups.
5. Reduce each group.

The MapReduce model has horizontal scaling properties. However it works with large batches, which is in appropriate for stream processing.

2.3 High-performance Distributed Stream Reasoning using S4

Hoeksema et al. present an implementation of a C-SPARQL query engine based on Yahoo S4[2]. S4 is a distributed streaming framework similar to Strom. The architecture chosen by Hoeksema and Kotoulas is likewise to the KATTS architecture (see chapter 3). Multiple instances of the same building blocks are run in the cluster. By applying a grouping on the key field variable bindings go to the same instance of the building block. Building bocks without a key are instantiated on each node. Their implementation supports also reasoning, which adds additionally complexity to the system [9].

However they implement C-SPARQL, which does not provide complex time joins. It is not possible with their solution to define time with a certain overlapping. Furthermore the usage of S4 enforces the instantiation of new building blocks for each grouping value. Storm can handle multiple grouping values per instance. Hence we expect over all a better performance as with the prototype of Hoeksema and Kotoulas.

[1]http://hadoop.apache.org
[2]http://incubator.apache.org/s4/

3

Architecture

The general concept for stream processing in a distributed setup is as follow:

1. Parse Query
2. Setup the cluster by providing a processing topology.
3. Start reading from the input streams.
4. Write the results to the output stream.

This chapter covers the query execution and topology creation, the architecture of Storm and the building bocks of the KATTS system. The concepts and building blocks presented in this chapter are combined together in chapter 4 to execute a sample query in a cluster with real data.

3.1 Query Execution

Figure 3.1 shows the simplified query execution process. KATTS implements only the transformation of the query tree into the Storm topology and the deployment of the generated topology.

The build process of the query tree includes the parsing of the SPARQL query and the construction of the query tree. The query tree is a graph structure, where each node represents an operator and the edges represents data streams. Figure 4.2 shows such a query tree (see section 4.2.2). KATTS does not support the parsing of SPARQL queries. However it can read the query tree from a XML file.

The build process of the Storm topology maps the query tree into a Storm topology. This mapping replaces the operators with Bolt and Spout implementations and by linking each component with a concrete processing instruction (see section 3.3). The resulting topology can be run on a Storm cluster (see section 3.2).

The generated topology reflects the parallelization of the query tree and grouping of the data. However the topology does not necessarily reflect the underling network structure. The created topology leaves space for improvements by rearranging and merging the nodes of the topology (see section 5.1).

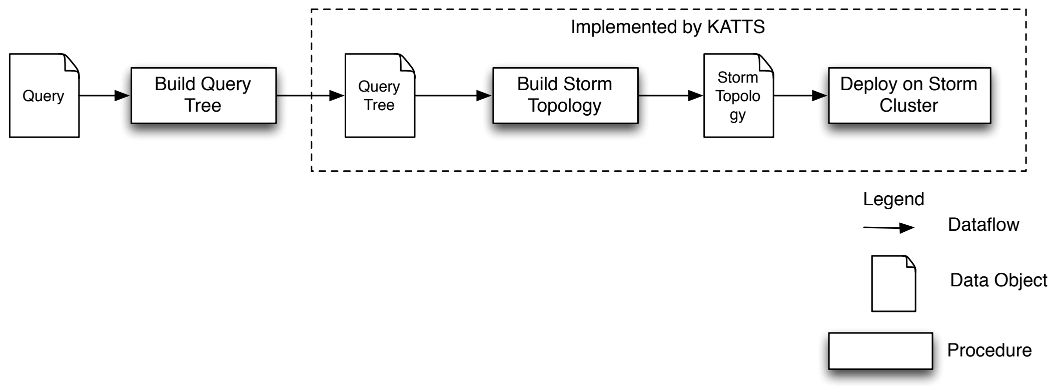

Figure 3.1: Query execution in KATTS. The query execution in consists of parsing the query tree, building the Storm topology and the deployment of the topology on the Storm cluster.

3.2 Storm Architecture

MapReduce and similar technologies can store and process data in large scale. But they are not designed to be realtime-processing systems and they cannot be hacked to turn them into that kind of systems. Storm was built by Nathan Marz to fill the gab of a realtime computation system for massive scale.

Storm uses ZeroMQ to build upon it primitives for realtime processing. These primitives are cluster bootstrapping, process parallelization, data transmission between nodes and code distribution.

3.2.1 Building Blocks

Figure 3.2 shows a Storm cluster with the different building blocks of a Storm based system. The building blocks are distributed on different machines. These blocks allow the communication between each sub system. Beside this low level perspective Strom provides a logical concept in terms of topology. The topology represents the computational task as a graph, where the nodes are processing tasks, collectively called components, and the edges are queues between the components. Each cluster may execute multiple topologies in parallel.

Section 3.2.2 covers more details about Storm topologies. The following sections describe the low level building blocks in more detail.

Bootstrapping

Each node in the network runs some part of the computational task. Therefore the different nodes need to communicate with each other.

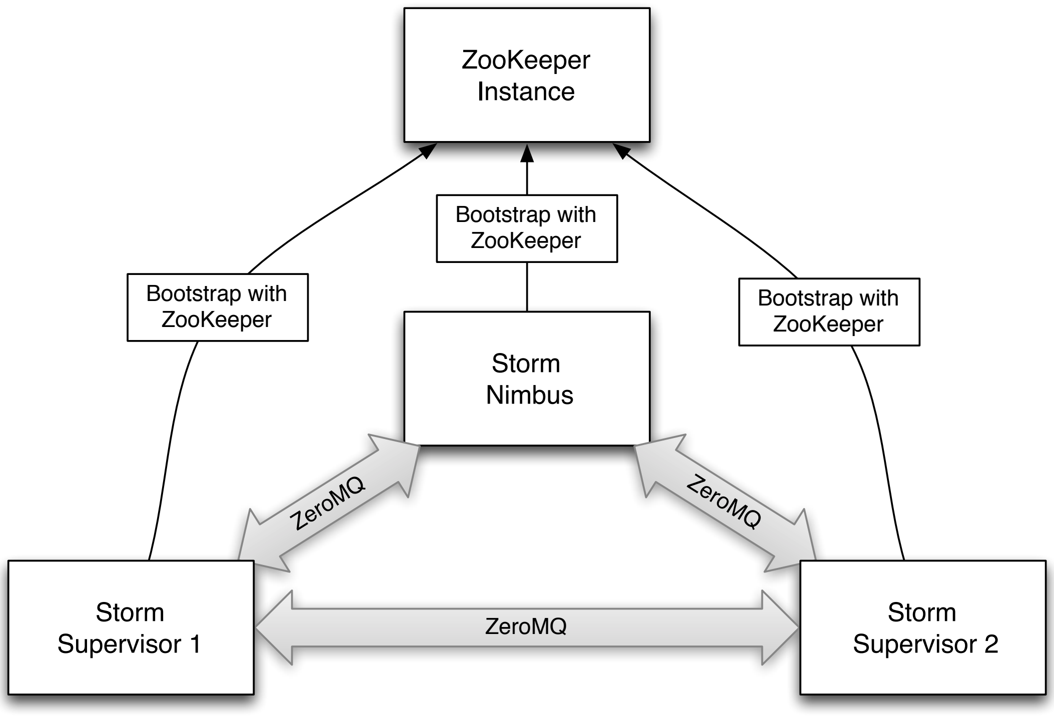

Figure 3.2: Storm Architecture. The Storm framework constists of several building blocks including the ZooKeeper, Nimbus Host, Supervisor Host and ZeroMQ.

The clqustер is bootstrapped over ZooKeeper[1]. Hadoop and S4 uses also ZooKeeper for the cluster management. The nimbus host and supervisor hosts connect first to the ZooKeeper instance and registered them on it. Thus the different processes can start communicate with each other.

Nimbus Host

The nimbus node coordinates the Storm cluster. It is responsible for the distribution of the Storm topology in the cluster, the code distribution, the killing and restarting of topologies and for the work balancing. The distribution of topology contains the scheduling of component instances on supervisors and the starting of the topology tasks (see 3.2.2). For the code distribution the nimbus starts a service on which supervisor host can download the code. The nimbus host provides a rebalancing command, which reassigns the components of the topology to different workers (see section 3.5).

Supervisor Host

The supervisor host runs multiple workers. Each worker can run multiple executors. The nimbus node can assign tasks to the executors by calling the supervisor host. Each worker runs his own Java Virtual Machine (JVM). Each executor runs multiple tasks. A task runs in a single thread. The code for execution is downloaded from the nimbus host.

Communication

The nimbus and supervisor host communicate with Apache Thrift [2]. Apache Thrift is an interface definition language for services. Strom was from the very beginning meant to be multilingual. Apache Thrift allows the integration of multiple languages over sockets. The topology components communicate over ZeroMQ. ZeroMQ is an asynchronous message queue system for high-performance, distributed and scalable applications, which provides better performance than Thrift.

3.2.2 Storm Topology

A Storm topology is the logical view of the processing task. The topology is a graph of Bolts, Spouts and queues. Bolts and Spouts are nodes in the graph. Spouts are adapters to other systems, from where they get the data for processing. Bolts process the incoming data on the edges. The edges are queues between the nodes in the graph.

Bolts and Spouts, more abstract the components, can be executed in parallel on multiple machines in the cluster. Storm provides the facilities to map them to the machines in the cluster, to link the parallelized instances with queues together and to distribute the code in the cluster.

[1]ZooKeeper is a centralized service for distributed synchronization, naming services and configuration management.

[2]http://thrift.apache.org/

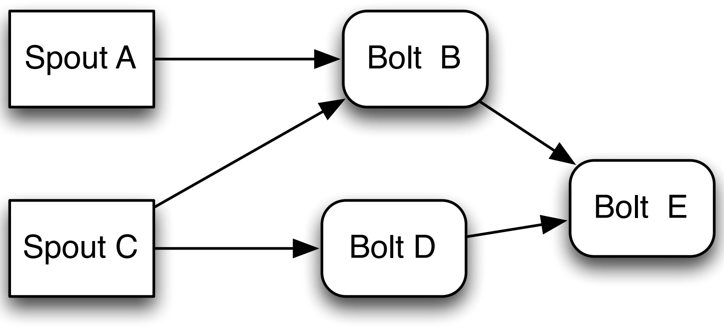

Figure 3.3: Simple Storm Topology. A Storm topology consists of Bolts and Spouts. The edges indicate the queues between the components.

Components

A component is either a Spout or a Bolt, where Spouts are sources of data streams and Bolts process data streams. For example a Spout can read from another message queue, from a file or even from a shared memory. Figure 3.3 shows a simple Storm topology, where two Spouts produce data streams and three Bolts process the data streams. Each of the Bolts and Spouts require an implementation written in Java [3]. KATTS provides concrete implementations for Spouts and Bolts as building blocks for complex queries. Section 3.4 covers these primitives.

Grouping and Parallelization

Storm can run instances of the component on multiple nodes. Therefore each component specifies the order of parallelization. The order of parallelization indicates the number of instances created and executed on the machines. A component instance is collectively called a task. Each task is executed in an executor, where each executor is run on a worker. Each Supervisor can start multiple workers.

In order to instantiate components multiple times in different executors, the data must be routed somehow to the different instances. Strom provides the grouping facility to control the routing of the data. Per default Storm supports the following groupings:

- All Grouping: The data stream is replicated to each task.
- Shuffle Grouping: The data is routed evenly distributed to the different tasks.
- Local or Shuffle Grouping: If a local task is ready to accept a data item it is routed to the local task, if there is no free task the data item is routed to a remote task.
- Field Grouping: All data items with the same value of a certain field go to the same task.

[3]Storm supports also other languages, for example Python.

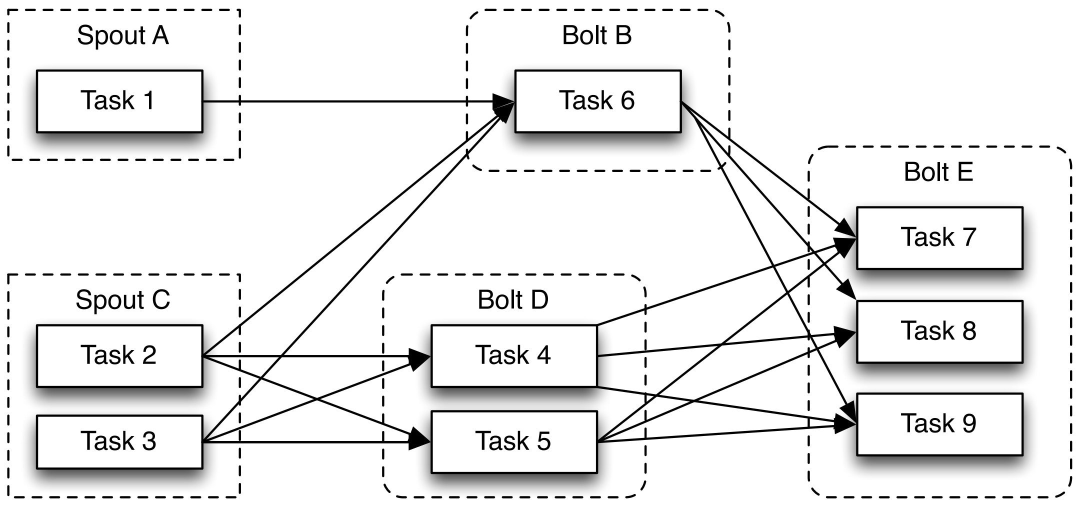

Figure 3.4: The Storm topology from figure 3.3 including tasks and groupings.

- Global Grouping: All data go to the task with the lowest ID.
- Direct Grouping: The producing task defines the destination task directly during the processing.
- Custom Grouping: Other groupings may be added for other use cases.

Figure 3.4 shows the same topology as figure 3.3 the difference is that in figure 3.4 the underlying task structure and the grouping is added. Spout A und Bolt B have a parallelization of one, Spout C and Bolt D have a parallelization of two and Bolt E has a parallelization of three. The sending task decides where to send the message in question. For example the grouping between Spout C and Bolt D is done on the task of the Spout C.

The dynamic decision of the routing leads to unpredictable routings. It is not predefined which task receives a certain message. Section 3.4.2 covers the implication of this design decision of Storm.

Task Distribution

Storm executes tasks in executors. Each executor can execute multiple tasks. A worker can run multiple executors. Figure 3.5 shows the distribution of the tasks from figure 3.4 to the workers, where each worker can be on the same machine or on a different machine. Since KATTS runs always only one task per executor, the executers are not present in the figure.

The mapping between figure 3.4 and figure 3.5 is an inefficient one, because after each processing step the data is sent to another worker. Hence the message is sent over the network. Storm provides a facility for the scheduling of tasks in the cluster. This facility

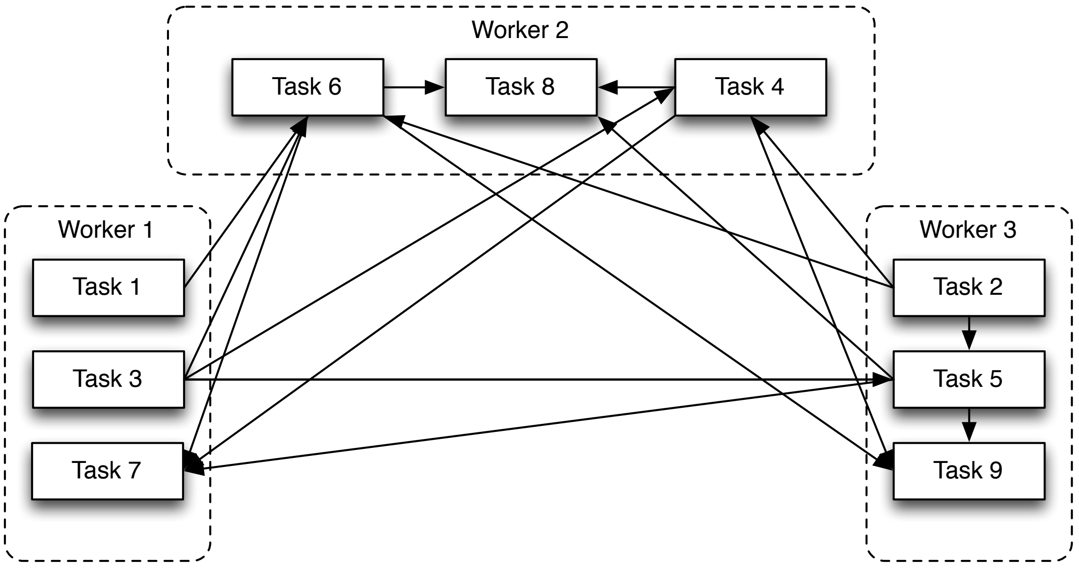

Figure 3.5: The Storm topology from figure 3.4 including workers. Each task is distributed to a worker.

tries to assign tasks evenly distributed over the nodes. This scheduler implementation leads potentially to situations as shown in figure 3.4.

One approach to improve KATTS is to provide a custom scheduler that does take in account which grouping is applied to the queues between the components. The section 5.1 covers more about how such an improvement can be achieved.

Another approach is to change the grouping to local or shuffle grouping in cases, where the grouping does not matter. In case the Bolt is stateless [4] the grouping does never matter, because each task has the same knowledge. The replacement with a local or shuffle grouping adds a preference to the locality, which in turn reduces the messages sent over the network. The sending of a local message is faster than sending a message over the network, hence the throughput and the performance is increased. This approach is similar to the one proposed by Hoeksema and Kotoulas [9]. They place an instance of each stateless component on each node[5]. See chapter 4 for more details about this improvement.

[4]A stateless Bolt does not maintain an inner state.
[5]They called them keyless processing elements (PE).

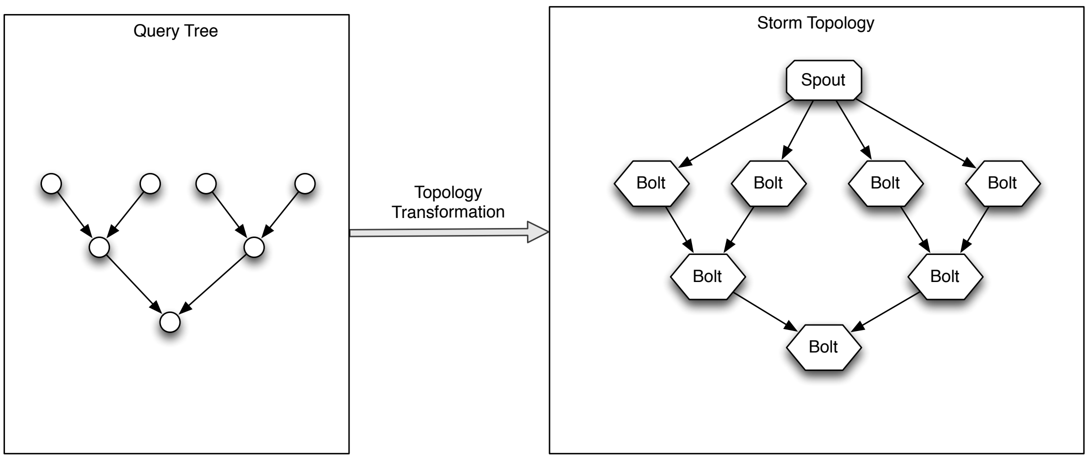

Figure 3.6: Mapping between Query Tree and Storm Topology. The mapping between the query tree and the Storm topology replaces the operators from the query tree with the Bolt and Spout implementation of KATTS.

3.3 Query to Topology Mapping

The general concept of transforming the query to a Storm topology is as follow:

1. KATTS provides implementations of Spouts and Bolts, which accepts configuration for a concrete behavior.
2. They are arranged in a Storm topology.
3. The configuration of the Bolts and Spouts are stored in a XML.

The query tree represents the SPARQL query as a tree structure. This structure contains nodes, which process the data and edges, which represents data connections. Currently KATTS supports only queries represented by trees in XML form. KATTS convert this tree into a Storm topology.

Figure 3.6 shows the mapping of the query to the topology. During the mapping each node of the tree is replaced with a Bolt or a Spout. The mapping adds the configuration of the Bolts and Spouts as defined in the query tree. The root Spout provides input data for the subsequent Bolts. The resulting structure is a Storm topology as explained in section 3.2.2. The following sections cover the mapping process in detail and explain the fundamentals of the data streams.

3.3.1 Triples vs. Variable Bindings

The input format of the query is a stream of triples. Each triple consists of a subject, a predicate and an object. The output of a query is a set of variable bindings, where each variable binding is a map of a variable name with an associated value. Storm uses so

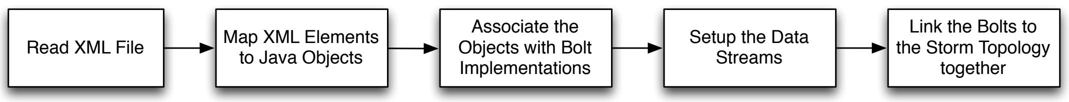
Read XML File
Map XML Elements to Java Objects
Associate the Objects with Bolt Implementations
Setup the Data Streams
Link the Bolts to the Storm Topology together

Figure 3.7: Transformation process of the input query to the Storm topology. The transformation process of the XML file to the Storm topology contains several steps.

called tuples to exchange data between Bolts. Tuples are similar to variable bindings; the only difference it that the mapping between the variable and the value is determined by the position of the value in a list and not over the variable name. The mapping can be still reconstructed, because the variable order and thereby the variable mapping can be preliminarily distributed in the cluster. This reduces the communication overhead, because the variable name is not included in the message.

3.3.2 Transformation Process

Figure 3.7 shows the transformation process of the XML file to the Storm topology. The first step is reading the XML file. The XML file contains the query tree represented with XML elements linked together. The next step maps the single query tree element to Java Objects. JAXB[6] associates each XML element to a corresponding Java Object. Each Java Object contains a method to setup the topology. This method creates the Bolt and Spout objects and links them together with data streams.

3.4 KATTS Components

Storm provides a framework to implement a stream-processing engine with a range of possibilities. However Storm requires the implementation of Bolts and Spouts for the handling of single data items (see section 3.2.2). The following sections cover selected abstract concepts, such as stateful processing nodes and synchronization, and selected concrete implementations. KATTS uses these basic processing blocks to build up the whole complexity of stream query processing.

3.4.1 Stateful Bolts

Some of the processing nodes require combining different variable bindings to a new one. For example if a Bolt builds the sum of all values in a certain stream, then the Bolt must store the previous sum. This state is crucial and the system should never throw it away. Therefore KATTS uses a storage backend for such states, which can be replaced

[6]JAXB is a Java based XML serialization framework.

depending of the actual scenario. Currently KATTS supports the Infinispan[7] framework from JBoss with a local storage engine, which fits the needs for the scenario described in chapter 4. The local storage prevents KATTS to use the rebalance command of Storm, which can relocate the tasks in the Storm topology without restarting the topology. Infinispan supports also distributed approaches. The distributed approach enables the rebalancing of a topology, because the state is transferred between the single machines. However the infinispan distributed approach continuously distribute the state to other machines in the cluster, which in turn adds an unnecessary overhead.

Storm does also support stateful Bolts in newer versions of the contrib project. But this implementation does not allow the direct replacement with a local storage engine and it does not support the transfer of the state in case of a rebalancing, it is transferred constantly. Neither the Storm approach nor the KATTS approach support all possible scenarios. Hence there is room for improvement (see section 5.1).

3.4.2 Heartbeat

The groupings provided by Strom are useful to implement aggregations (see section 3.4.6) and joins (see section 3.4.7), because they can route the messages dynamically depending of the message content. This routing ensures that messages, which should be joined or aggregated, are sent to the same task. However the routing becomes unpredictable by introducing dynamic routing, because the stream between tasks contains no continuous information about the state of the sending task. Stateful Bolts require such information; because they typically combine multiple variable bindings together and therefore they need to know how far the previous Bolt is with processing. The heartbeat stream provides this information by sending a message with the current processing state of previous Bolts in a regular interval.

The heartbeat Spout produces at a regular interval a message, which is sent to all data source tasks. The data source task assigns to the heartbeat message the time of the last read in triple. This time is than called heartbeat time. This new heartbeat is sent to all subsequent tasks. Figure 3.8 shows underlying structure of the heartbeat implementation. Parallel to each regular stream also a heartbeat stream is added. In this example task 1 and task 2 do not receive all regular messages. The messages are dynamically routed to one of them. However the heartbeat with the including heartbeat time is sent to both tasks. Both know with the message (HB3: HT9) that the source task will not send any data item with a lower date as 9. This basic promise allows the implementation of time depending Bolts, because the information of the processing state between the tasks can be shared.

3.4.3 Synchronization

Storm guarantees that emitted variable bindings are sent to the receiver. However there is no guarantee that two variable bindings emitted on different tasks arrive in the

[7] http://www.jboss.org/infinispan/

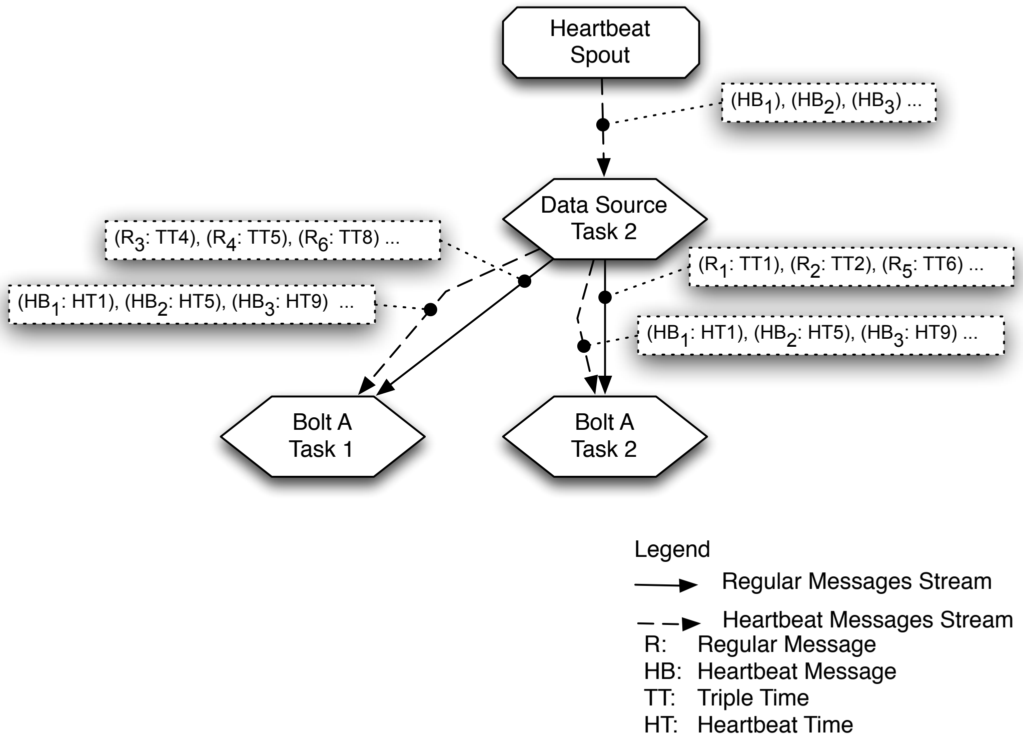

Figure 3.8: Architecture of Heartbeat. The heartbeat ensures the steadily processing of the messages in the buffers by sending on a separate stream constantly synchronization messages.

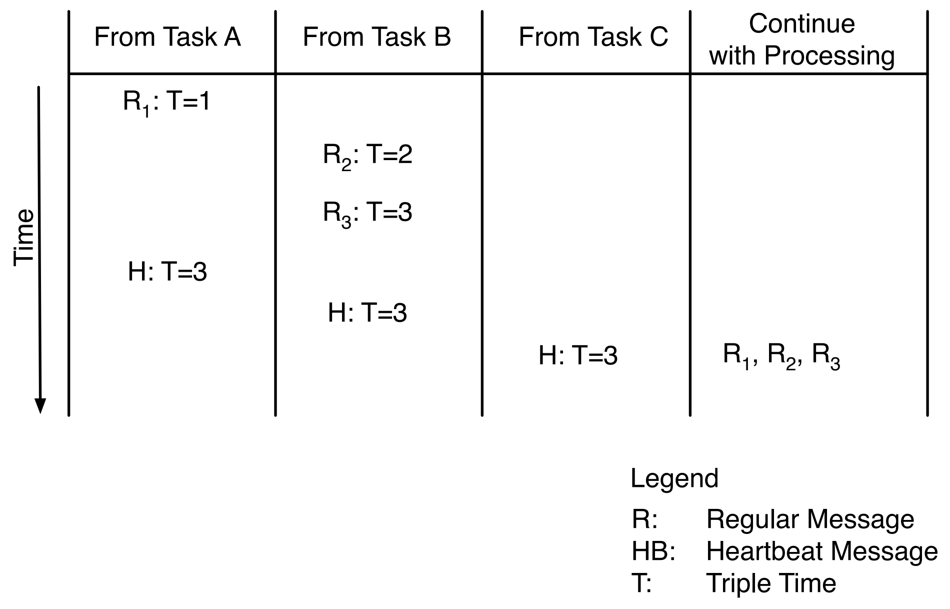

Figure 3.9: Use case of the heartbeat in synchronization. The heartbeat delivers the required information to process the data even if there is no message from task C.

temporal correct order – even if they are from the same Bolt. Storm does also not provide a mechanism for synchronizing different variable binding streams. These two properties are indispensable for stream joins and for stream aggregations. KATTS provides these two properties in abstract implementations based on stateful Bolts and heartbeats.

Figure 3.9 shows a common situation during the synchronization. Each of the three tasks can send a message at any time. Task C does not send any data; hence there is no information about the processing state. The synchronization implementation must assert that the variable bindings are processed in the correct temporal order. Therefore the regular messages are on hold until the heartbeat arrives from task C.

3.4.4 Filters

A filter removes variable bindings from a stream. The core functionality of filter is the decision making of removing or keeping a binding. KATTS provides a filter with an expression-based decision and one for filtering triples. Both do not need to save any state; hence they are stateless.

Expression Filter

The expression filter does allow the encoding of the decision of filtering a variable binding into an expression. The expression syntax is the Spring Expression Language (SpEL)[1]. All values of the variable bindings are available to the expression. The sample expression from listing xxy shows an expression that removes variable bindings that have a priceFactor lower than 1.4 or greater than 2.5. The priceFactor contains the value of the variable with the same key in the variable binding. When the expression evaluates to true the variable binding is kept, when it evaluates to false the variable binding is thrown away.

$$\#priceFactor > 1.4 \quad \&\& \quad \#priceFactor < 2.5$$

Triple Filter

KATTS consumes only triple streams. Internally KATTS uses only variable bindings and not triples. The triple filter allows the filtering of triple streams depending on the subject, predicate or object. The intent of the triple filter is to remove all irrelevant triples from the triple stream. The triple filter emits the triple as a variable binding with the variables subject, predicate and object.

3.4.5 Expression Functions

Functions create new variables based on other variables in the variable bindings. By introducing also expressions for functions – as for filters – functions can constitute complex transformations. The syntax is formulated in SpEL. The following expression shows a computation of a factor from two other variables, where 0.5 multiplies the factor when it is below zero and three multiplies the factor when it is greater than zero:

$$\#priceLow/\#priceHigh > 0 \ ? \ (\#priceLow/\#priceHigh) * 3 : \\ (\#priceLow/\#priceHigh) * 0.5$$

The expression can contain all values from the variable binding, but no values from previous processed variable bindings. Because the expression function Bolt does not maintain any state.

3.4.6 Aggregations

A common use case in query languages is the aggregation of data over the time grouped by a certain field. KATTS supports this feature by using a sliding window. The window is moved over the time and for each window an aggregation is built.

Sliding Window Concept

The sliding window concept divides the time into slides. The window moves then over the slides and builds the aggregation by executing on each slide the aggregation function. All data that falls into a slide is threated as a whole. The slide and window size can be defined independent on the aggregation function. Figure 3.10 shows all possible combinations of slides and window overlaps. The window size can be equal, greater or smaller as the slide size. The green window shows the next window after the black one.

The data of the slides is stored in a hash map. An item inside the map is called slide bucket. Each slide bucket has an index and a corresponding value. The calculation of a slide is implemented as follow:

1. Determine the slide index by dividing the event time by the slide size. This index number is unique over the time.

2. Determine the bucket index of the hashmap by applying a hash function on the slide index. The hash function calculates the residual of the divisions by the window size. This gives a unique number inside the window.

3. The current slide bucket is recalculated by the value of the variable binding.

When the window moves to the next slide, the aggregation function is applied over all slides in the hash map and the aggregate is emitted on the output stream of this Bolt. This structure allows the implementation of arbitrary aggregation functions by providing a recalculation method of the bucket, a storage object and a slide aggregation function.

Aggregation Functions

An aggregation function provides three facilities:

1. Storage object: An object that stores the intermediated results per slide bucket. In case of the average aggregation, the storage must store the sum of all elements in the slide bucket and the number of elements in the current slide.

2. Slide bucket recalculation function: This function merges a new variable binding into given slide bucket. For the calculation of an average value the number of values and the sum are increased.

3. Slide aggregation function: This function aggregates all slides inside a window. For the average value calculation this method adds up all sums per bucket and divides the sum by the number of all elements in all buckets.

These building blocks enable the implementation of arbitrary aggregation functions. KATTS provides aggregation function implementations for calculating the minimum, the maximum and the average.

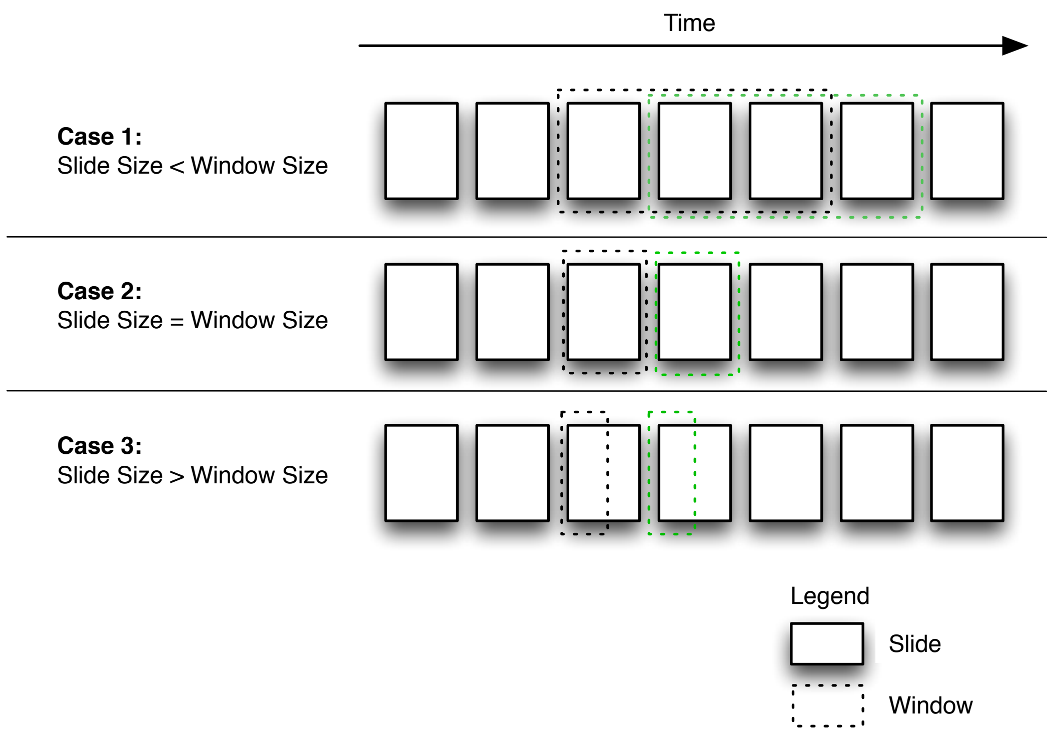

Figure 3.10: Sliding window with three different overlappings. The green window is the succeeding of the black one. Each slide has a fixed size over the time.

Grouping

The data is be grouped by a certain variable value. This group by has the same meaning as the group by of relational databases. In case the aggregation function is executed on a ticker stream, the price of the ticker can be aggregated grouped by the ticker symbol. This kind of aggregation allows the calculation of the average ticker price, the minimal ticker price or the maximal ticker price. See the section 3.4.6 functions for more functions.

The implementation uses a hash map with the group by value as the key. The hash map value contains the data of the slides.

3.4.7 Joins

Relational databases are used for long time to join different data together. They are a good starting point to cover the design space for stream joins.

Relational Database Joins

From traditional database systems there are several join types known [4]:

- Cross-Join: All items on left and the right side are joined together.
- Inner Join: All items matching the join criteria are joined together.
- Outer Join: One side is joined together with the other by keeping the entries of the first side.

Relational databases implement these joins by using the one of following algorithm [4]:

- Nested Loop Join: Two nested loops iterates over each the join sets. The complexity is with two sets $O(n^2)$, where n is the number of elements in the sets.
- Sort Merge-Join: The join attribute sorts both sets. The join merges both sets by iterating over them. The complexity is reduced compared to nested loop join to $O(n)$ for the join. The sorting remains on average at $O(n * \log(n))$.
- Hash Join: For each join attribute a hash table is maintained. This table provides a constant look up for join candidates in the other set. The complexity for the join is $O(1)$.

Often relational databases supports the following join conditions:

- String Equality: The two attributes of the sets must equal / not equal. Possibly a string operation is applied first on them. For example to lower chars or conversion to a lexicographical representation.
- Mathematical Comparison: The join depends in this case on the given mathematical expression.

- Combinations of the above: The join condition can contain any combination of the above condition concatenated with a logical AND or OR connector and may be with a leading logical NOT.

Stream Joining

The joining of data streams requires also temporal join criteria. The tuples in the data stream are always temporal ordered and queries potential want to join tuples depending on their temporal order. Further in streaming application the data arrives, when the query is already known. These fundamental differences require other approaches in handling joins as for relational databases.

The main goal of this work is to show the effects of grouping. Hence no complete and full-featured join is a requirement. The provided join is able to preform joins based on the time and a join expression. The definition of this join consists of three structures:

1. Before Join Eviction Rules: This set of rules is applied on all variable bindings buffered, when a new variable binding arrives. The rules may evict not reasonable tuples depending on time relations between them.
2. Join Condition: The condition on which a join between two variable bindings is executed.
3. After Join Eviction Rules: This set of rules is applied on all variable bindings stored in the buggers after the join is executed.

Additionally KATTS provides a more efficient join that implements an equality join with a same time temporal condition.

Both implementations are not capable to implement all join types and they do not exert the most efficient algorithms. Section 5.1 covers some hints how to implement this more generally and with better performance.

3.5 Dynamic Adaption to Environment

The environment of the system changes over the time. For example one data source has suddenly a bigger latency or the interval between triples decreases. In both examples the data of other sources must be buffered longer and more memory is required. Beside the additional memory the processing distribution changes due to the changes of the environment, because some Bolt must process more data as before. In consequence a distributed triple stream-processing engine must support load balancing and eviction of interim results.

3.5.1 Eviction

The eviction of data allows freeing memory by removing items from buffers, which may not be used in future. The selection of items to remove may be done randomly or with any model, which provides a good prediction about likelihood of future usage of an item in the buffer.

In any case the tasks must communicate with each other on a separate channel, because they need to exchange information about the processing state and the eviction of items. KATTS provides by the heartbeat a mechanism to determine which variable binding can be processed respectively removed. The heartbeat uses a separate queue to communicate with the other Bolts. Section 3.5.3 covers also other approaches to communicate.

3.5.2 Load Balancing

Storm provides a command to rebalance the load in the cluster. Potentially this process moves tasks from one node to another. This implies that the state of stateful Bots can be transferred to other machines. As described in section 3.4.1 the state transfer is hard to achieve.

Another approach provided by Storm to get better load balance is the local or shuffle grouping (see section 3.2). Depending on the current load of the node, variable bindings are sent to a local or remote task. By using signals between remote tasks this approach can be extended to consider also global information about the load to distribute the workload. Section 3.5.3 covers ways to implement such signals.

Storm requires a fixed parallelization degree of a component. Often it is not clear during the bootstrapping, which of the Bolts requires the most processing time. At least KATTS accept a parallelization weight, which indicates the number of task for a component compared to the other components in the topology. It is also possible to define the number of tasks per processor, which adapts the number of threads to the number of processors in the cluster. Section 5.1 presents an additional approach, which can overcome these limitations of Storm.

3.5.3 Communication Channels

In context of load balancing and eviction the following communication channels are reasonable in KATTS:

- Storm provides with the message queues and direct grouping the possibility to send messages over ZeroMQ to other tasks. Since ZeroMQ is employed this approach is potentially an effective channel even with hundred thousands of messages per minute.
- The Storm contribution project[8] contains an extension to Storm for sending signals between tasks.
- ZooKeeper provides also facilities to exchange information. ZooKeeper is suitable for massive information exchange, but it has the possibility to use it like a black board, which in turn can lead to a better architecture.

[8]The Storm contribution project contains additional examples and reusable code for the Storm project. https://github.com/nathanmarz/storm-contrib

4 Evaluation

Grouping strategies allow us to partition data horizontally, which in turn enables us to distribute the workload of the query engine across many machines in a cluster; this process scales. We evaluate two different grouping strategies by executing them in a cluster of twelve nodes each with twenty-three CPUs.

4.1 Test Setup

The University of Zurich, more precise the Dynamic and Distributed Information Systems Group, provides a cluster for the tests. The following sections cover the details about the infrastructure, the query submission and the available analyses on the results.

4.1.1 Infrastructure

The cluster consists of twelve machines each with twenty-four cores and 64 GB of memory. They run under a Debian Squeeze with Terascale Open-Source Resource and QUEue Manager (TORQUE) installed. TORQUE is a distributed resource manager, which allows assigning resources to jobs and the execution of jobs in the cluster.

One out of the twenty-four cores is reserved for the operating system (OS) and therefore cannot be used for experiments. Hence the scenarios consists only of variations of the cores between one and twenty-three cores.

The machines are connected with a one GBit LAN and one switch. Each user's home directory is mounted from network storage connected with a separate one GBit connection. This gives a simple access to all required scripts and the test data. The JVM and some system libraries are pre-installed. ZooKeeper and ZeroMQ must be loaded during the job submission.

4.1.2 Automation of Query Submission

The following steps are involved in the query submission:

1. Deploy the dependencies in the cluster.
2. Start the required daemons.

3. Submit the query to the Storm nimbus host.
4. Wait for query completion.
5. Evaluate the run.

Figure 4.1 shows the above steps in a sequence diagram. The first step is to start ZooKeeper on the master node. Since ZooKeeper is only used for bootstrapping and minor data logging one instance is enough for our evaluation. Then ZeroMQ, the Java bindings for ZeroMQ and Storm are deployed on all involved machines. Next the Nimbus and Supervisor hosts are started. At this point the system is ready for accepting Storm topologies. The query can be executed by providing the corresponding XML file with the query tree in it and the KATTS jar that contains all the Spout and Bolt implementations.

A real live streaming application does never terminate. The system has to be reproducible. Hence our test data set is finite and the system must terminate. The termination of the query is detected by listening on the heartbeat. When at the bottom of the query tree the heartbeat indicates the file end, all the data seems to be processed and the query is terminated.

When the query is completed, the run can be evaluated by aggregating the collected results (see section 4.1.3). The last step is to kill all daemons and to clean up the involved machines.

4.1.3 Result Collection

Strom collects some basic metrics during the topology execution. The following metrics can be retrieved from Storm over the Thrift interface on the Nimbus node:

- The number of messages consumed per Bolt.
- The number of tuples messages per Bolt and Spout.
- The overall processed messages in the cluster.

However Storm does not provide metrics on task level. By injecting[1] a task monitor for each task in the cluster allows the collection these metrics. The monitor logs the number of messages sent from and to each task. The data is stored in ZooKeeper. With the additional data the proportion between local and remote messages can be inferred.

The starting time, the end time and the cluster setup is stored in ZooKeeper. The evaluation process extracts the information from Storm and the additional information in ZooKeeper and produces the following data per query run:

- Start Time
- End Time

[1]The hook system of Storm allows the injection of task monitors.

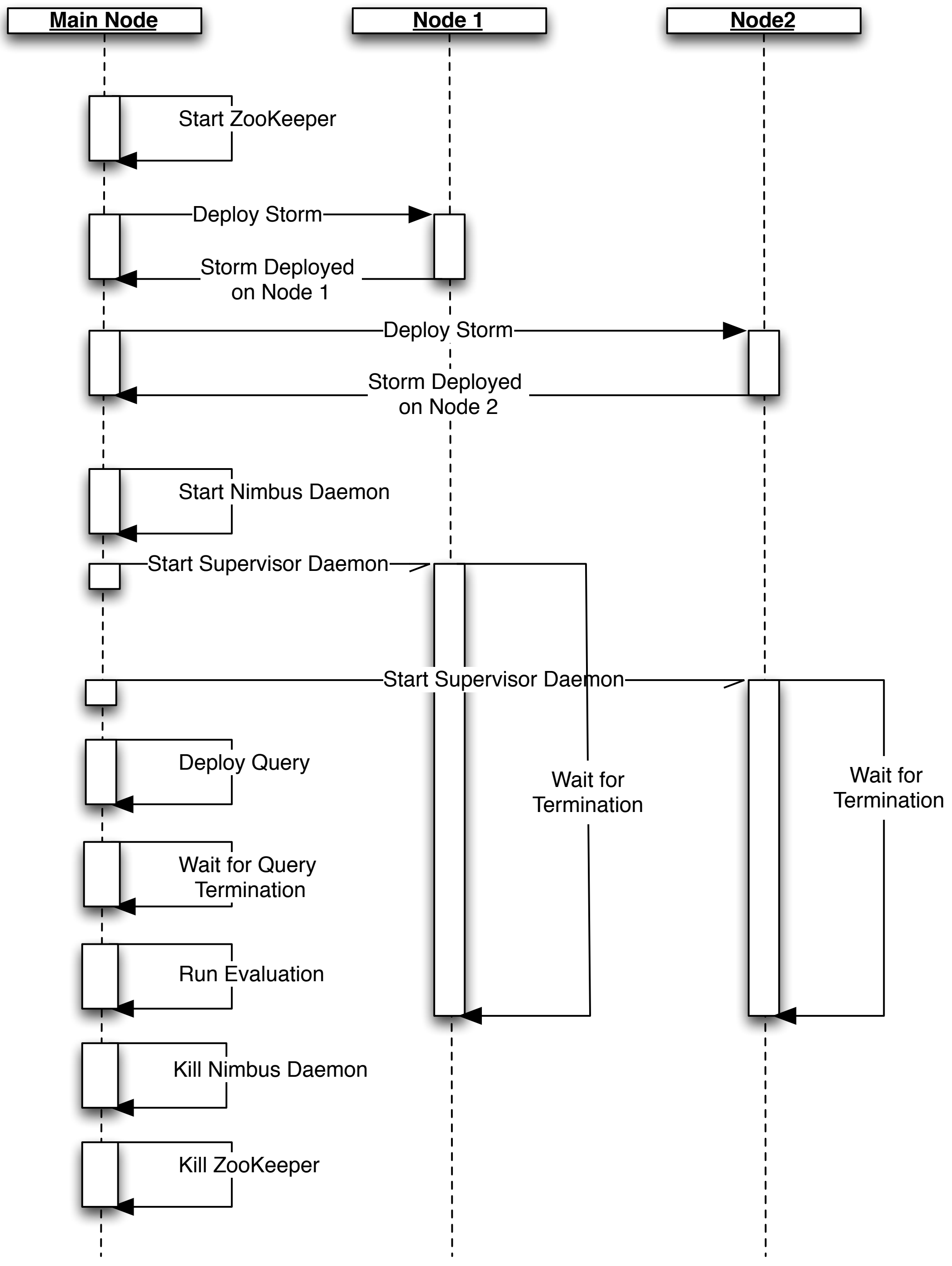

Figure 4.1: Sequence diagram of the query submission. The query submission requires to start first ZooKeeper, then deploy Strom on each node in the cluster, then start the Nimbus host and Supervisor hosts and then the query can be run. The run is evaluted and all the started services are stopped.

- Duration
- Total Number of Message Processed
- Total Number of Remote Messages
- Total Number of Local Messages
- The number of machines involved in the run
- The total number of processors in the cluster
- The number of processors per machine
- The number of expected tasks
- The total number of produced output tuples.
- The proportion between tasks and processors
- The number of input triples
- The number of messages send between the tasks

4.1.4 Analyses

A query runs on one or more machines with a defined number of CPUs and it executes a specific query. The collected data enables the evaluation of the job by different analyses. Also different Bolt implementation can be compared with this analyzing framework. The collected data allow the inferring the following metrics:

Throughput: The throughput tells how many triples can be processed per second. It can be calculated by dividing the number of input triples by the duration.

Speedup: Gustafson defines the speedup as follows [6]:

$$Speedup = (s + p)/(s + p/N)$$

Where N is the numbqer of processors, s is the amount of time spent on the part of the program, that is not parallelizable and p is the amount of time spent on the part that is parallelizable. If the system scales perfectly, the speedup equals N. This is because s is zero in this case.

Message Sending Behavior: The message sending over the network consumes more resources than local messages. The comparison of the local and remote message behavior can give a good insight to the system. The graph can visualize the communication between the different network nodes. Alternatively also Sankey charts can be used to show the message exchange in the network. Also analyses of the communication between Bolts are interesting or the message sending between tasks. These analyses can give insights about the bottleneck of the system.

4.2 Test Scenario

To verify the hypotheses a set of tests are run on the cluster.

4.2.1 Data Used for the Evaluation

The test data contains stock exchange and US government contracts data. Both data sets are from 2001. The stock exchange data contains the ticker symbol, the ticker price, the company name and other information, that is not relevant for our query[2]

The US government contracts contains information about orders the US government has give to companies. The contract signed date, the contract amount, the department, which place the order, the contract company and other information is provided by the http://www.usaspending.gov.

The data sets are available as CSV files, where each line contains one contract resp. one ticker. We convert this data to triple CSV files, with four columns: Triple Date, Subject, Predicate and Object.

The triple date indicates the creation date of the triple. In the contract case it is the signed date and in the ticker case it is the date on which the ticker information was generated. The subject is a unique identifier for each row in the source CSV. The predicate is the column name and the object is the value of the corresponding row and column in the source CSV.

The resulting files are much more verbose and hence they are by a factor of 10 bigger as the source files.

4.2.2 Query Used for the Evaluation

We execute the following query: All companies should be selected, that have a price increase of factor greater than 3 with in 20 days and have signed a contract with the US government with in 20 days.

SPARQL Representation of the Query

The above verbal description of the query can be expressed in the following pseudo SPARQL query:

[2]Source is https://wrds-web.wharton.upenn.edu/wrds/

```
PREFIX wc: <http://wrds.crsp>
PREFIX ug: <http://usaspending.gov>

SELECT ?agency, ?ticker, ?cname, ?contract, ?contract_date, ?vol
WHERE {
    TEMPEX(${timeRange("?fct").contains(timeRange("?cntrct").beginTime())}) {

      OVER (?ticker, 20 DAY, 1 DAY) {
        CONCURRENT {
           ?record wc:compname ?cname;
                    wc:ticker   ?ticker;
                    wc:price    ?price.
        }

        max(?price) AS ?mx
        min(?price) AS ?mn
        BIND (?mx / ?mn) AS ?fct)
        FILTER (?fct > 3)
      }

      CONCURRENT {
          ?cntrct ug:id          ?cid;
            ug:contractor ?cname;
            ug:agency     ?agengy;
            ug:amount     ?amount.
      }
    }
}
```

The keyword TEMPEX restricts the result set by a temporal expression. This keyword joins all sub sets together, when the temporal expression evaluates to true.

The OVER keyword indicates an aggregation over the time. The first argument indicates the group by field. The second indicates the window size and the third indicates the slide size.

The keyword CONCURRENT constitutes the join of variable bindings, when they occur concurrently. This expression is shorthand for a TEMPEX expression with a condition that all sets must be at the same time for a join.

The keyword max and min are aggregation functions. They build the aggregation of the given field over the time expressed in the OVER expression. BIND executes the given expression and assigns the resulting value to a new variable.

The FILTER keyword removes all items from the result set, which do not satisfy the given expression.

Graphical Representation of the Query as a Query Tree

Figure 4.2 shows the query as a graphical representation. The transformation from the SPARQL query to this representation is a manual process. In future a query parser may do this transformation automatically (see section 5.1). The graph indicates already many details about execution. Especially it shows the order of execution, the grouping strategies, the Bolts to use and the configurations of the Bolts. KATTS does accept this kind of structure serialized as XML.

XML Query Tree Representation

Appendix A.2 shows the query in a XML representation. It contains all details required for the execution. Figure 4.2 left out some low lever details. For example it does not show information about the parallelization or the file locations of the sources.

The XML consists of a set of components. Each component represents a Bolt respectively Spout. The id attribute identifies a component. The following names refer to this id.

The tickerSource and contractSource read in the data and emit the triples to the tripleFilter components. The triple filters remove the spare triples and transfer them to the OneFieldJoin_Ticker and OneFieldJoin_Contract. These joins merge the triples from the tickerSource together respectively contractSource. The tickers are aggregated in the Partitioner. The expressionFunction calculates the factor of the minimal and maximal price. Before the contracts and tickers are jointed, the expressionFilter removes tickers that have a factor below four. Finally the TemporalJoin_TickerAndContractData joins the contracts and tickers together.

4.2.3 Evaluated Grouping Strategies

Appendix A.2 shows the query with the field grouping strategy. The differences from the field grouping to the local or shuffle grouping reside in the triple filter and in the grouping of expressionFunction and expressionFilter.

The local or shuffle grouping strategy does not force a routing of the message, except where it is required due to the nature of the Bolt[3]. In theory Storm must perform better with a field grouping, since already at the beginning the data is sent to the machine, which is responsible for handling a certain ticker or contract. This reduces the network traffic which intern improves the overall performance.

As described in section 3.2.2 Storm tries to distribute the tasks evenly in the cluster. Hence the above hypothesis should not hold, because the grouping is not considered during the scheduling of the tasks.

In our evaluation we want to show the performance improvement by using a shuffle or local grouping strategy compared to a field grouping strategy.

[3]For example in case of a join or an aggregation the grouping is required, because they have a internal state which adds a dependency between single messages

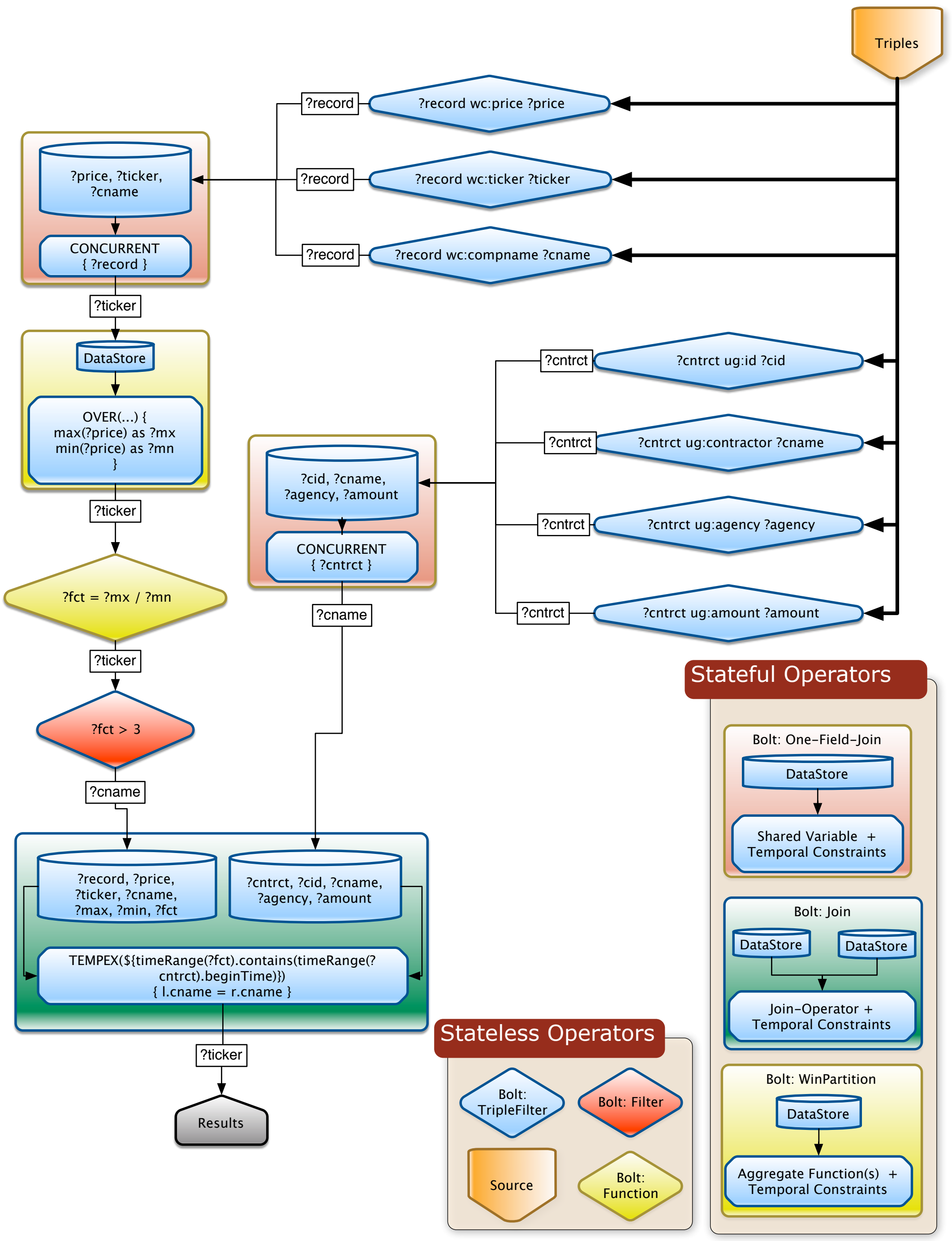
Triples
?record
?record wc:price ?price
?record
?record wc:ticker ?ticker
?record
?record wc:compname ?cname
?price, ?ticker, ?cname
CONCURRENT { ?record }
?ticker
DataStore
OVER(...) { max(?price) as ?mx min(?price) as ?mn }
?ticker
?fct = ?mx / ?mn
?ticker
?fct > 3
?cname
?cntrct
?cntrct ug:id ?cid
?cntrct
?cntrct ug:contractor ?cname
?cntrct
?cntrct ug:agency ?agency
?cntrct
?cntrct ug:amount ?amount
?cid, ?cname, ?agency, ?amount
CONCURRENT { ?cntrct }
?cname
?record, ?price, ?ticker, ?cname, ?max, ?min, ?fct
?cntrct, ?cid, ?cname, ?agency, ?amount
TEMPEX(${timeRange(?fct).contains(timeRange(?cntrct).beginTime)}) { l.cname = r.cname }
?ticker
Results
Stateful Operators
Bolt: One–Field–Join
DataStore
Shared Variable + Temporal Constraints
Bolt: Join
DataStore
DataStore
Join–Operator + Temporal Constraints
Bolt: WinPartition
DataStore
Aggregate Function(s) + Temporal Constraints
Stateless Operators
Bolt: TripleFilter
Bolt: Filter
Source
Bolt: Function

Figure 4.2: Graphical representation of the query tree. The graphical representation shows all operators of the query tree including some configuration. Each of the operator is replaced during the mapping with a Bolt or Spout implementation.

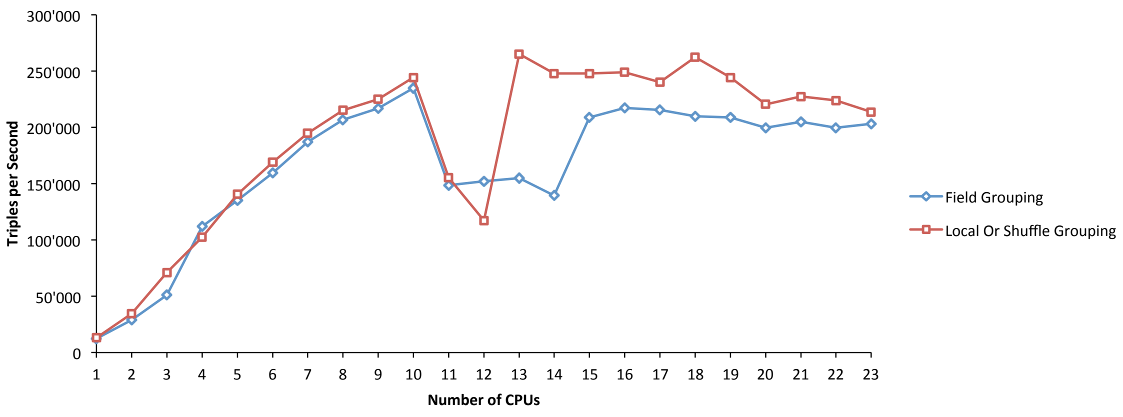

Figure 4.3: Troughput with one node and one to twenty-three CPUs. The detailed results can be found in table A.1 and in table A.2.

4.3 Results and Discussion

We evaluate the two different grouping strategies by comparing their performance on processing the query described in section 4.2.2 against the contract and ticker data described in section 4.2.1 on a cluster of twelve machines with a setup described in section 4.2.

4.3.1 Throughput

Figure 4.3 shows the throughput with one node and one to twenty-three CPUs. It does not show dramatic differences between the grouping strategies. The drop of local or shuffle grouping is not clear. It can be due to the chosen parallelization of the Bolts. A suboptimal parallelization degree of the Bolt can lead to an insufficient distribution of the tasks to the processors. Potentially the reading of the data is a bottleneck, because the input is read from only twenty files and therefore is not parallelized perfectly.

Figrue 4.4 shows the throughput with one CPU and one to twelve nodes. As for only one node with one to twenty-three CPUs there is no big difference between the two strategies. The local or shuffle grouping strategy shows a bit higher throughput as the field grouping.

Figure 4.5 shows the throughput with twenty-three CPUs and one to twelve nodes. Up to two nodes both strategies lead to nearly the same throughput. More nodes lead to a better performance of the local or shuffle grouping strategy. With twelve nodes it has a throughput, which is by a factor of three higher as the field grouping strategy. This confirms the hypothesis that the local or shuffle grouping can compensate at certain level the inadequate scheduler of Storm for the KATTS setup. Overall the local or shuffle grouping performs only better in case also the number of CPUs is increased.

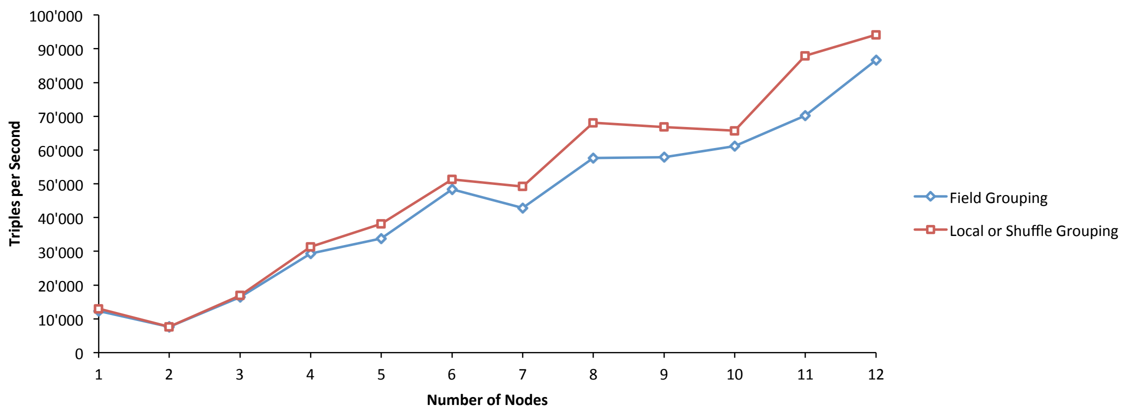

Figure 4.4: Troughput with one CPU and one to twelve nodes. The detailed results can be found in table A.3 and in table A.4.

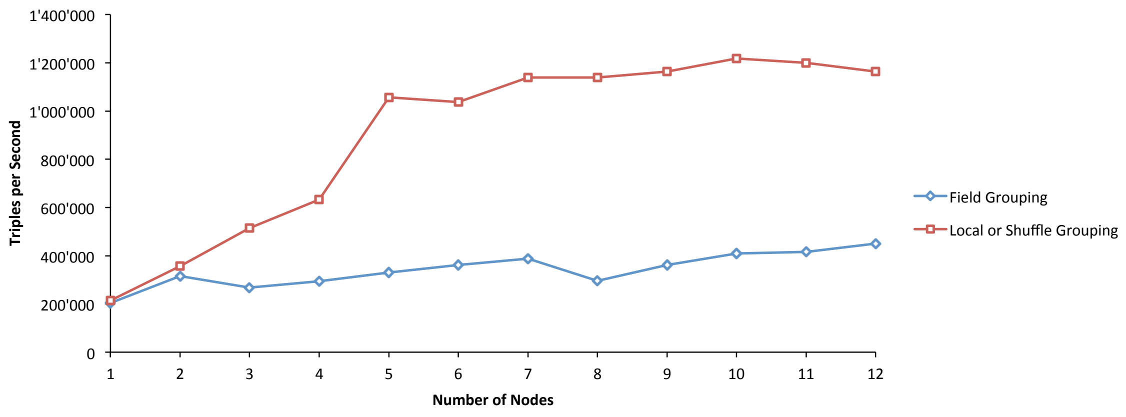

Figure 4.5: Troughput with twenty-three CPUs and one to twelve nodes. The detailed results can be found in table A.5 and in table A.6.

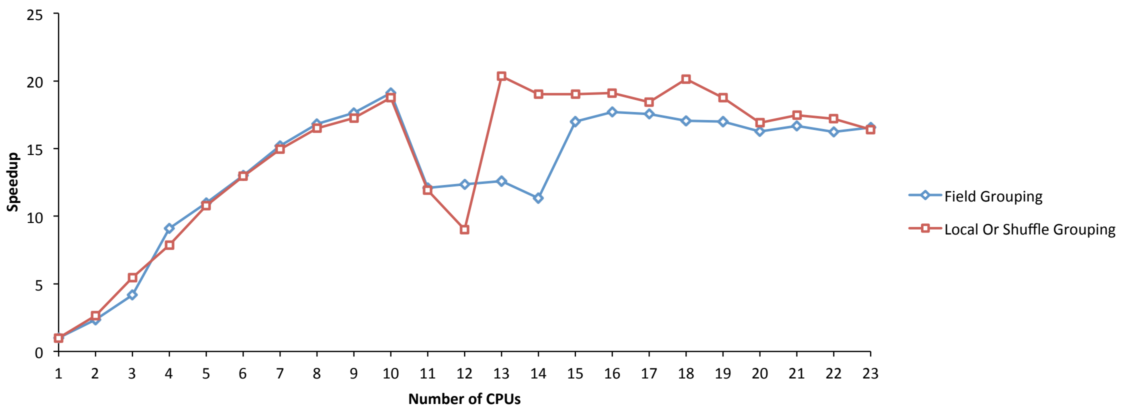

Figure 4.6: Speedup with one node and one to twenty-three CPUs. The detailed results can be found in table A.1 and in table A.2.

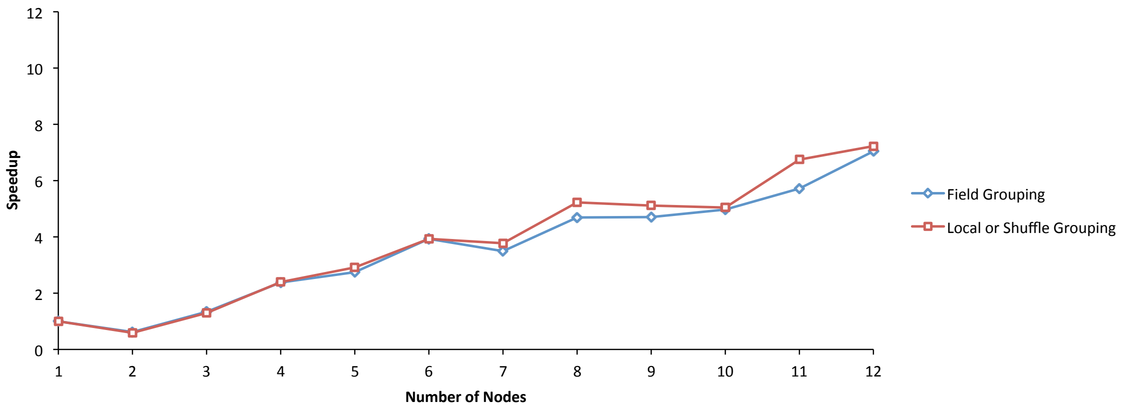

Figure 4.7: Speedup with one CPU and one to twelve nodes. The detailed results can be found in table A.3 and in table A.4.

4.3.2 Speedup

If a system scales perfectly the speedup (see section 4.1.4) would be as high as the added resources. Figure 4.6 shows the speedup with one node and one to twenty-three CPUs. For a low number of CPUs the speedup is higher as the added resources. This implies that the KATTS waste a lot of resources, when it can use only a few processors. With ten CPUs the speedup drops as the throughput (see section 4.3.1).

Figure 4.7 shows the speedup with one CPU and one to twelve nodes. Both strategies lead nearly to the same speedups. The scaling is not perfect; it is always below the perfect speedup of N (see section 4.1.4), where N is the number of resources added.

Figure 4.8 shows the speedup with twenty-three CPU and one to twelve nodes. Also with twenty-three CPUs KATTS does not scale perfectly. The local or shuffle strategies scales at the beginning with up to five nodes. The speedup equals nearly the factor of added resources. With more than five nodes both strategies does not scale. The field

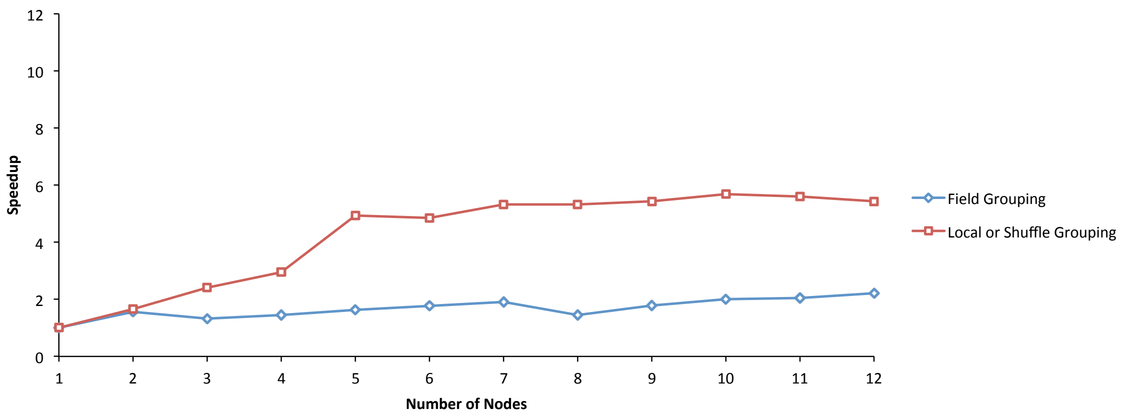

Figure 4.8: Speedup with twenty-three CPUs and one to twelve nodes. The detailed results can be found in table A.5 and in table A.6.

grouping does not scale at all and with twelve nodes the speedup is the local or shuffle grouping by a factor of three higher than the field grouping. Section 4.3.5 discuses the message sending exactly of this case. It investigates a potential network bottleneck.

Even if both strategies does not scale the local or shuffle grouping is much better. Hence it makes sense to investigate other approaches to improve the performance by distributing the tasks more evenly in the cluster. Section 5.1 covers possible improvements of the scheduling process.

4.3.3 Message Sending Behavior

The message sending behavior gives insights about the network usage. Figure 4.9 shows the message sending behavior with one Node and one to twenty-three CPUs. Since there is only one node, no remote traffic is shown. The local or shuffle grouping sends over all more messages. The local or shuffle grouping strategy requires information about the load of the local tasks. If no task is available the message is sent to a remote node. Perhaps the additional messages inform the tasks about their load.

Figure 4.10 shows the sending behavior with one CPU and one to twelve nodes. Both strategies lead to nearly the same message behavior. There is almost no local traffic, because on each node only a few tasks are executed. Hence most messages must be sent over the network. Interestingly with more than ten nodes the number of local messages increase. Potentially this is due to parallelization adaption of KATTS. In case more processors are involved also more tasks are created except the file source, which have a constant parallelization of ten.

Figure 4.11 shows the sending behavior with twenty-three CPUs and one to twelve nodes. As in the other scenarios the message load of the local or shuffle grouping is higher. But the remote message exchange is much lower with the local or shuffle grouping strategy as with the field grouping strategy. Maybe this difference causes the much higher throughput of the local or shuffle grouping. The local traffic of the field

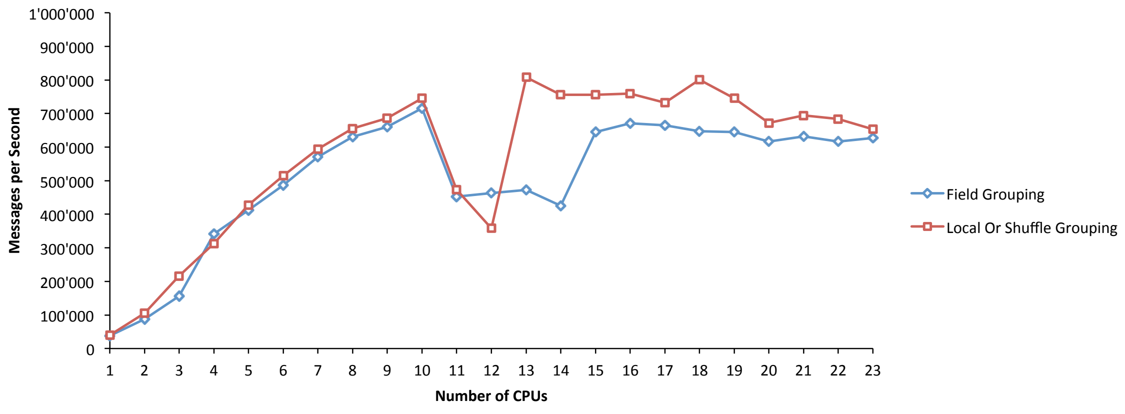

Figure 4.9: Messages sent with one node and one to twenty-three CPUs. The detailed results can be found in table A.1 and in table A.2.

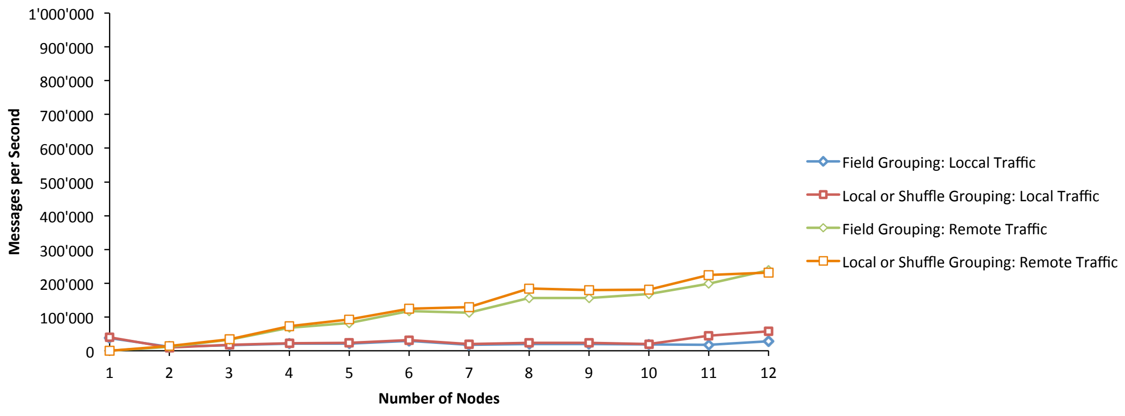

Figure 4.10: Messages sent with one CPU and one to twelve nodes. The detailed results can be found in table A.3 and in table A.4.

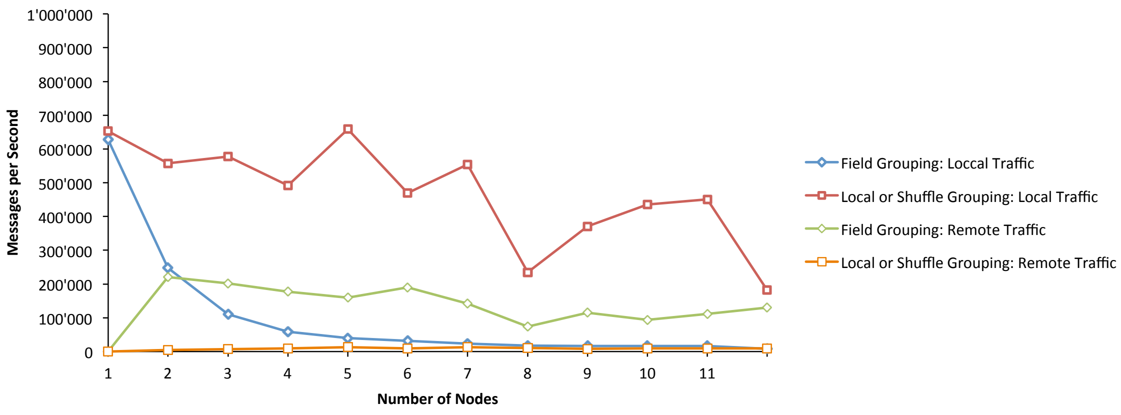

Figure 4.11: Messages sent with twenty-three CPUs and one to twelve nodes. The detailed results can be found in table A.5 and in table A.6.

grouping constantly decreased with more nodes.

4.3.4 Message Distribution

Figure 4.12 shows the Sankey chart with the message sending between the different Bolts with field grouping on 9 nodes with 23 CPUs. Figure 4.13 shows the Sankey chart with the message sending in the same setup as Figure 4.12, but with the shuffle or local grouping strategy. They show a similar behavior. The streams between the data source and the triple filter cause the most traffic. For example between the contract-Source and TripleFilter_ContractCompany about ten million triples are sent. Between the TripleFilter_ContractCompany and the Contract_Join only about forty thousand variable bindings are sent. Overall between the triple filters and the sources about 55 million messages are sent and between the other Bolts only about two million messages are sent. Possibly this explain why the local or shuffle grouping strategy outstands the field grouping.

4.3.5 SendingBehaviorBetweenNodes

We recognized a drop in sense of throughput for the local or shuffle grouping when the number of nodes is changed from five to six (see section 4.3.1). The CPU number is in both cases twenty-three. Figure 4.14 shows the network traffic for five nodes and figure 4.15 shows the network traffic for six nodes. All nodes in both cases send and receive more or less the same number of messages. But the number of messages sent over the network is different. In case of five nodes about 1.8 million messages and in case of six nodes only 1.4 million messages are sent remotely. Potentially this drop is the root cause of the throughput drop, but it cannot be caused by a network bottleneck, because the overall sent traffic is smaller in case of six nodes. If the network would be the bottleneck the number of messages sent should be constant. Hence the bottleneck is more likely an

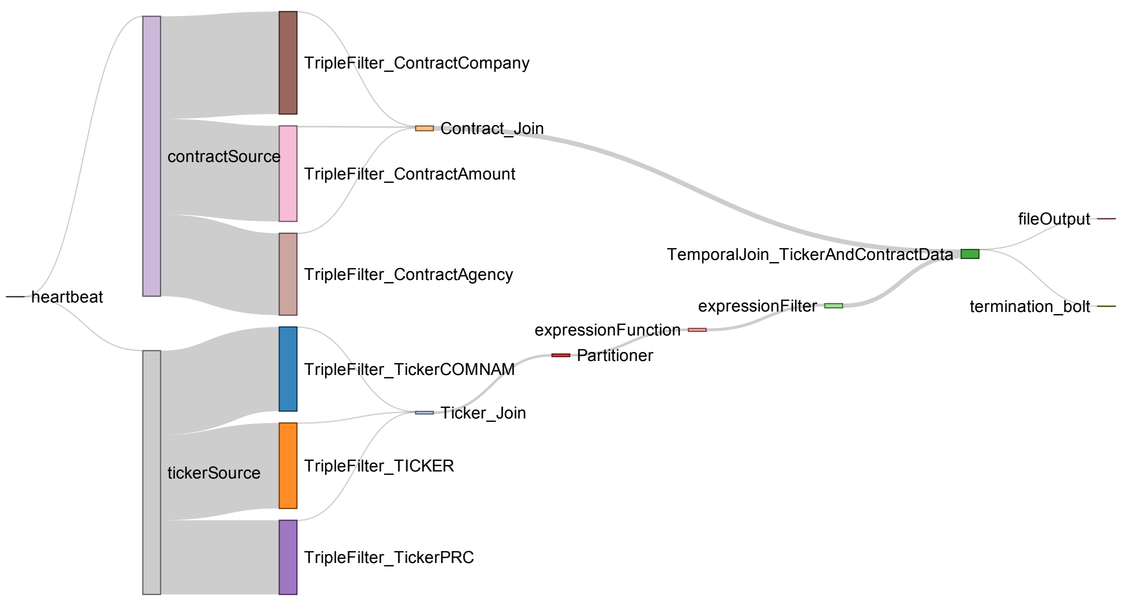

Figure 4.12: Sankey chart of the message sending behavior of a setup of nine nodes with 23 CPUs with the field grouping strategy.

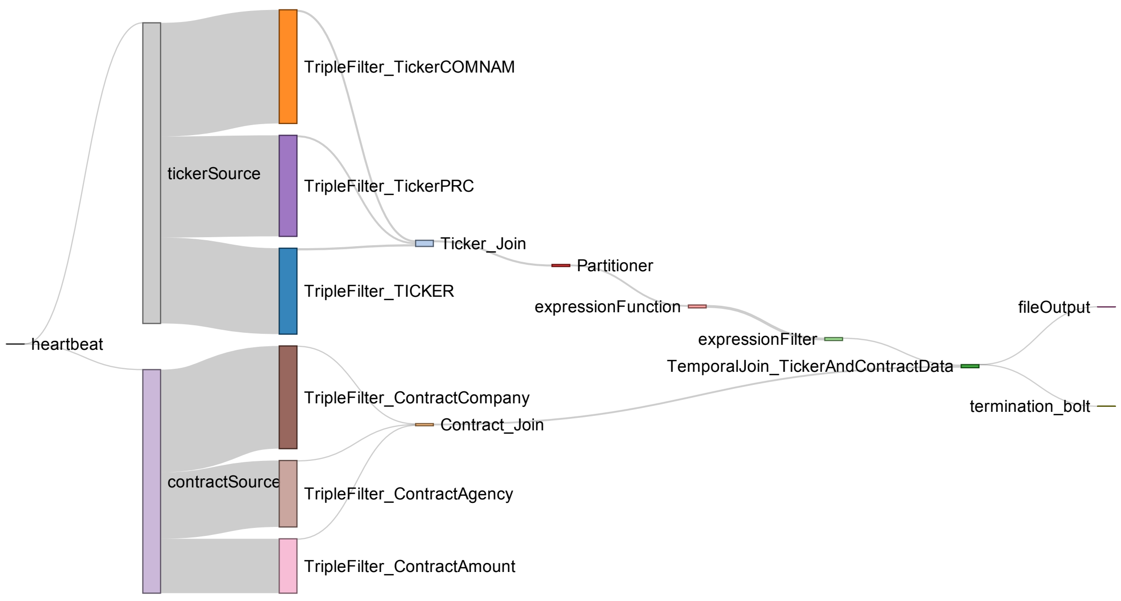

Figure 4.13: Sankey chart of the message sending behavior of a setup of nine nodes with 23 CPUs with the local or remote grouping strategy.

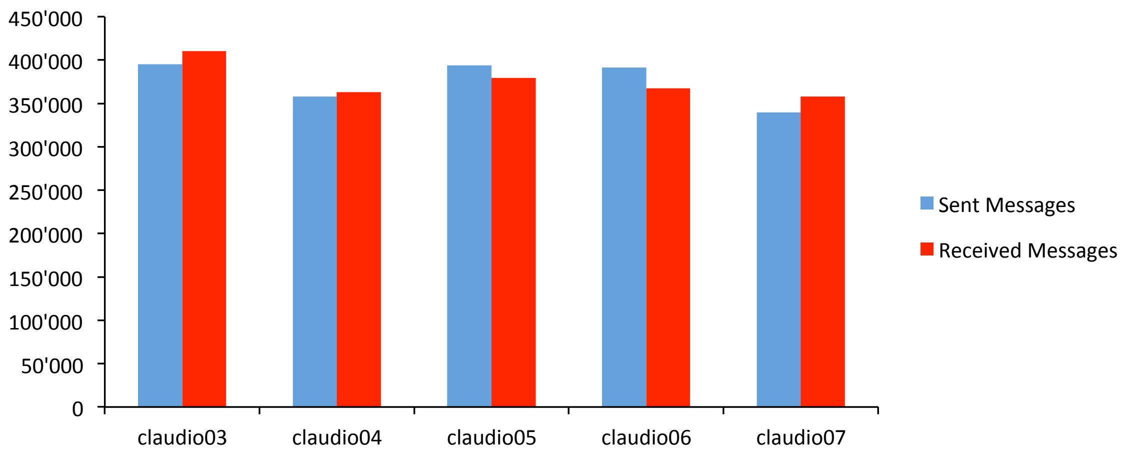

Figure 4.14: Per node traffic for a setup of twenty-three CPUs on five nodes. Detailed results can be found in table A.8.

unbalanced processor load.

However this is not proven and requires further investigations. Section 5.1 covers details about narrowing down the encountered bottleneck.

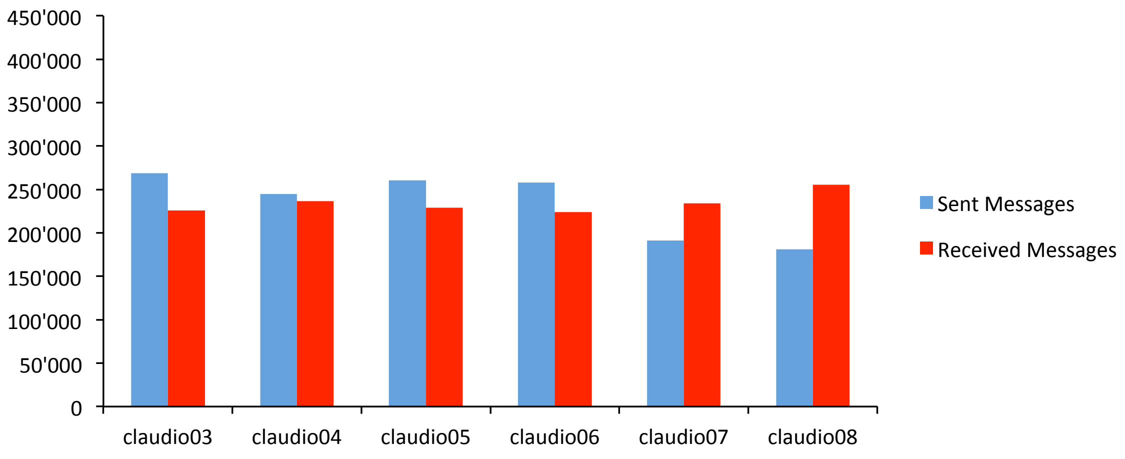

Figure 4.15: Per node traffic for a setup of twenty-three CPUs on six nodes. Detailed results can be found in table A.7.

5

Summary

In context of this master thesis a review of existing approaches to process triple stream was done. The findings help to understand the problem domain and to support with first approaches for the implementation. The implementation provides a set of basic building blocks for distributed query processing. The constructed building blocks allow the execution of queries in a distributed environment. The queries can contain join operators, filter operators, expression function and aggregations. The query must be provides as a query tree in XML. For running and evaluating of the implementation a query submission environment was built for a distributed setup.

We evaluate the scalability of KATTS by comparing two different grouping strategies. The results show that the system scales at a certain degree. Nevertheless there is room for improvement.

5.1 Future Work and Limitations

KATTS is a research prototype; hence there are multiple limitations. This section shows the limitations of KATTS and provides insights for future development and possible improvements.

5.1.1 Measure the Processor and Network Load per Task

As the findings in the chapter 4 show the implementation does not scale perfectly. The evaluation framework does not collect data about the processor load and the network load. The network load is only measured indirectly by the measurement of the messages sent over the network. Therefore no statement about the bottleneck of the system is viable.

To narrow down the limitiqng resource the processor and the network load should be investigated. The processor seems more likely to be the limiting factor, because the local or shuffle grouping reduces the remote messages by a factor of 1000 compared to the field grouping. But the throughput is only increased by a factor of three.

Analyses of the consumed CPU time per task can give insights about the processing distribution in the cluster. The measurement can be achieved by logging the consumed time to process a variable binding in a Bolt. Storm provides a facility to measure the processing latency. Potentially this can be used for these measurements. However Storm throws this information away in regular intervals. Anyway KATTS requires a facility to store time series.

A measurement on system level is not that useful, because it is not clear which Bolt actually runs on a processor. However a system wide processor load logging, can give insights about the load balancing between processors and it shows if all processor work on the problem.

The injected task monitor provides a network analyses on task level. It logs the number of messages sent per task respectively per Bolt. Nevertheless it is important to see if the one GBit LAN reached the maximal capacity to be sure that not the network is the bottleneck.

5.1.2 Stateful Bolts and Rebalancing

KATTS supports currently only a simple facility to store data. It does not allow the transfer of the storage between nodes in the cluster. This limitation prevents the usage of the Storm rebalancing functionality. Hence KATTS cannot adapt the structure depending on the actual load.

The Storm contribution project[1] contains a stateful Bolt implemented with a distributed database. As mentioned in section 3.4.1 this implementation makes sense if also reliability is a goal. If only the state should be transferred then other implementation may perform better, because they can send only data during the relocation of the Bolt. A reliable implementation has to send constantly the state to other nodes in the cluster.

he local or shuffle grouping does share knowledge about the local load; hence it can distribute the load better as other strategies. This advantage is achievable for other strategies by rebalance the topology without the overhead of additional messages.

5.1.3 Load Balancing with Dynamic Parallelization

Depending what findings emerge from the measurement of the processor load a dynamic adaption of the processor load can remove the bottleneck. As described in section 3.5.2 Storm is not capable to change the number of tasks. Storm 0.8.0 introduces the concept of executors, which removes the static coupling of one task per thread. With the new version its possible to assign in the rebalancing command the number of executors per component. The stable Storm 0.8.0 was released during the project.

This new feature can be used to dynamically adapt the number of threads per component. For the adaption the load per component must be logged. The load information should be stored in ZooKeeper. At regular bases KATTS then checks if the load is not evenly distributed, if so a rebalancing is executed. But a successful rebalancing requires the implementation of stateful Bolts with state transferring.

[1]The Storm contribution project contains reusable Spout and Bolt implementations.

5.1.4 Grouping dependent Scheduler

The application of field grouping allows the identification of similar data items. KATTS does not use this knowledge to place tasks, which process similar data, on the same node. Currently KATTS employs the default Storm scheduler, which tries to distribute the tasks evenly in the cluster. Hence the local or shuffle grouping performs better than the field grouping (see section 3.5.1).

We claim that a better scheduler, who considers the grouping, shows a better performance with field grouping than local or shuffle grouping. The results of the message sending behavior indicates that the local or shuffle grouping strategy sends more messages as the field grouping. When the scheduler removes the drawbacks of field grouping, possibly it outperforms the local or shuffle grouping.

5.1.5 Joins

KATTS supports only a subset of joins. It supports only a simple equity join with a concurrent time constraint and a more complex temporal join with eviction rules. Both does not support joins based on similarity. They also missing features to join streams based on mathematical expressions in an efficient way.

In a first step the temporal join should be extended to support multiple threads per task. The eviction takes long, because only one thread iterates over all the buffers. Maybe performance improvements can be achieved by using hash maps to split up the time and to build smaller buckets. Smaller portion of data can help reducing to compare data, which is irrelevant for the actual join. Potentially there exist other approaches without using one huge buffer.

Currently KATTS does not analyze the join expression. Maybe a better performance can be conducted by employing different join algorithms on one expression. For example the “and” operator in a join condition can be used to join subsets with a hash join.

5.1.6 Parsing of SPARQL Queries and Query Tree Optimization

SPARQL queries are more readable than XML files. KATTS accepts currently only XML files as input. Parsing of SPARQL queries leverage the usability, because it is hard to write the XML file and it requires a deep understanding of the system. For example which join is best for which join expression or which grouping strategy leads to best performance.

A SPARQL query parser for KATTS has to implement the following:

1. Parse in the SPARQL query to access the single elements.
2. Initiate the query tree.
3. Add the input streams to the query tree.
4. Add the triple filters to the query tree.

5. Add the elements from the SPARQL query to the query tree.
6. Configure each component in the query tree depending on the parameters set in the SPARQL query.
7. Setup the streams and groupings depending on the SPARQL query.

This process leads to a query tree, which can be processed by KATTS. Nevertheless it leads not automatically to an optimized tree.

By altering the order of the query tree nodes the throughput can be optimized. Currently KATTS executes the query tree as it is. Potentially the optimization criteria can be formulated in rules. For example if a stream is joined with another one, and then it is filtered based on a criterion, which is known before, the order should be changed. A rule engine can apply these rules on the query tree as proposed by Hartig et al. [7].

Maybe this query optimization should be also applied during runtime, because the load may affects also the order of the components. Currently rearranging of the topology during rebalancing is not supported by Storm. Therefore the underlying framework must be changed to allow this kind of optimizations.

5.2 Conclusion

KATTS provides basic functionalities to process triple streams such as filters, expression functions, aggregations and joins. The system can evict data depending on various conditions.

It does not scale perfectly as shown in the evaluation. Anyhow the system is capable to process more than a million triples per second with about ten machines each with twenty-three processors. The provided evaluation infrastructure enables the evaluation of different setups in the cluster. It measures the messages sending between tasks, Bolts, Spouts and nodes. It is able to evaluate key numbers as throughput, local and remote traffic and speedup.

For load balancing and eviction methods based on local information are investigated and applied. For global load balancing strategies are shown. Since Storm limits the possibilities in changing the topology structure during runtime, there is room for improvement in this area.

Overall a system is capable to process triple streams in a distributed environment and outperforms related systems [2] regard the throughput by a factor of eight.

[2]The system proposed by Jesper Hoeksema and Spyros Kotoulas based on S4 is able to process 160'000 triples per second with 32 machines each with 4 CPUs. [9]

References

[1] The spring expression language (spel) syntax., dec 2011.

[2] Darko Anicic, Paul Fodor, S. Rudolph, and N. Stojanovic. EP-SPARQL: a unified language for event processing and stream reasoning. In *Proceedings of the 20th international conference on World wide web*, pages 635–644. ACM, 2011.

[3] Davide Francesco Barbieri, Daniele Braga, Stefano Ceri, Emanuele Della Valle, and Michael Grossniklaus. C-SPARQL : SPARQL for Continuous Querying. In *18th International World Wide Web Conference*, volume 427 of *WWW '09*, pages 1061–1062. ACM, 2009.

[4] Ramez Elmasri and Shamkant B. Navathe. *Database Systems*. Addison-Wesley, 2010.

[5] J Gantz, C Chute, and A Manfrediz. The diverse and exploding digital universe, an updated forecast of worldwide information growth through 2011. IDC White Paper. *Director*, 2008.

[6] John L. Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31:532–533, 1988.

[7] Olaf Hartig and Ralf Heese. The SPARQL Query Graph Model for Query Optimization. In *Proceedings of the 4th European Semantic Web Conference (ESWC '07)*, volume 21436 of *Lecture Notes in Computer Science*. Springer, 2007.

[8] Martin Hilbert and Priscila Lopez. The world's technological capacity to store, communicate, and compute information. *SCIENCE*, April 2011.

[9] Jesper Hoeksema and Spyros Kotoulas. High-performance Distributed Stream Reasoning using S4. In *FIRST INTERNATIONAL WORKSHOP ON ORDERING AND REASONING*, 2011.

[10] Ralf Laemmel. Google's mapreduce programming model – revisited. *Science of Computer Programming*, 70(1):1 – 30, 2008.

[11] Clifford Lynch. How do your data grow? *Nature*, 2008.

[12] Matthew Perry and Amit P Sheth. SPARQL-ST : Extending SPARQL to Support Spatiotemporal Queries. *Design*, 2009.

A

Appendix

A.1 Evaluation Results

The following tables show the results from the evaluations. All results including the raw results are on the disc distributed with this thesis.

Number of CPUs	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	37’554	12’294	1.00
2	0	87’929	28’827	2.34
3	0	156’265	51’254	4.17
4	0	341’415	111’972	9.11
5	0	411’943	135’139	10.99
6	0	487’019	159’722	12.99
7	0	570’326	187’055	15.22
8	0	630’227	206’795	16.82
9	0	660’784	216’842	17.64
10	0	715’773	234’912	19.11
11	0	452’978	148’640	12.09
12	0	463’283	152’015	12.37
13	0	472’125	154’947	12.60
14	0	424’692	139’358	11.34
15	0	644’622	208’947	17.00
16	0	670’753	217’429	17.69
17	0	665’384	215’678	17.54
18	0	647’205	209’765	17.06
19	0	644’681	208’947	17.00
20	0	616’689	199’851	16.26
21	0	632’417	204’949	16.67
22	0	615’994	199’602	16.24
23	0	627’684	203’392	16.54

Table A.1: This table shows the throughput, speedup and the sending behavior of the query execution with field grouping on one node and one to twenty-three CPUs.

Number of CPUs	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	39'779	13'038	1.00
2	0	105'143	34'488	2.65
3	0	216'620	71'066	5.45
4	0	312'568	102'540	7.86
5	0	428'162	140'455	10.77
6	0	515'133	168'959	12.96
7	0	593'773	194'764	14.94
8	0	655'457	215'100	16.50
9	0	685'814	225'042	17.26
10	0	745'238	244'566	18.76
11	0	473'579	155'397	11.92
12	0	357'642	117'370	9.00
13	0	808'215	265'149	20.34
14	0	755'666	247'963	19.02
15	0	755'552	247'963	19.02
16	0	759'194	249'116	19.11
17	0	731'971	240'179	18.42
18	0	800'120	262'549	20.14
19	0	745'231	244'566	18.76
20	0	672'055	220'412	16.91
21	0	693'943	227'592	17.46
22	0	683'512	224'100	17.19
23	0	653'190	213'955	16.41

Table A.2: This table shows the throughput, speedup and the sending behavior of the query execution with local or shuffle grouping on one node and one to twenty-three CPUs.

Number of Nodes	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	37'554	12'294	1.00
2	11'577	11'639	7'608	0.62
3	33'419	16'695	16'433	1.34
4	68'563	20'883	29'316	2.38
5	82'152	20'799	33'785	2.75
6	117'801	29'615	48'383	3.94
7	112'946	17'672	42'871	3.49
8	155'882	19'912	57'612	4.69
9	156'239	19'991	57'840	4.70
10	168'001	18'354	61'165	4.98
11	198'195	17'913	70'227	5.71
12	239'041	27'995	86'620	7.05

Table A.3: This table shows the throughput, speedup and the sending behavior of the query execution with field grouping on one to twelve nodes on one CPUs.

Number of Nodes	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	39'779	13'038	1.00
2	13'610	9'720	7'660	0.59
3	34'468	17'223	16'953	1.30
4	73'173	22'285	31'285	2.40
5	92'648	23'456	38'103	2.92
6	124'941	31'412	51'319	3.94
7	129'496	20'261	49'153	3.77
8	184'140	23'532	68'085	5.22
9	180'309	23'072	66'755	5.12
10	180'428	19'712	65'691	5.04
11	224'448	44'726	87'899	6.74
12	231'152	57'728	94'130	7.22

Table A.4: This table shows the throughput, speedup and the sending behavior of the query execution with local or shuffle grouping on one to twelve nodes on one CPUs.

Number of Nodes	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	627’684	203’392	1.00
2	220’119	247’297	316’299	1.56
3	201’792	109’803	268’247	1.32
4	177’550	58’990	293’748	1.44
5	159’382	40’025	330’617	1.63
6	189’873	31’820	361’079	1.78
7	141’770	23’545	387’181	1.90
8	74’231	17’759	295’368	1.45
9	114’746	16’448	361’892	1.78
10	93’778	16’486	408’855	2.01
11	111’652	16’526	415’194	2.04
12	130’030	7’755	450’084	2.21

Table A.5: This table shows the throughput, speedup and the sending behavior of the query execution with field grouping on one to twelve nodes on twenty-three CPUs.

Number of Nodes	Remote Messages sent per Second	Local Messages sent per Second	Throughput Triples / Second	Speedup
1	0	653’190	213’955	1.00
2	5’031	557’274	356’275	1.67
3	7’486	577’058	515’000	2.41
4	9’503	492’274	632’598	2.96
5	12’358	658’301	1’057’105	4.94
6	9’060	469’194	1’036’645	4.85
7	12’314	553’936	1’139’574	5.33
8	10’120	234’289	1’139’574	5.33
9	8’594	370’971	1’164’348	5.44
10	9’249	435’401	1’217’273	5.69
11	8’887	451’345	1’199’104	5.60
12	8’802	182’250	1’164’348	5.44

Table A.6: This table shows the throughput, speedup and the sending behavior of the query execution with local or shuffle grouping on one to twelve nodes on twenty-three CPUs.

Host	Messages Sent	Messages Received
claudio03	268’474	225’706
claudio04	244’871	236’569
claudio05	260’457	229’073
claudio06	258’233	223’697
claudio07	191’129	234’063
claudio08	181’193	255’249

Table A.7: The table shows the messages sent and received in a local or shuffle grouping with six nodes and twenty-three CPUs.

Host	Messages Sent	Messages Received
claudio07	395’115	410’524
claudio08	358’065	363’155
claudio09	393’873	379’725
claudio10	391’347	367’248
claudio11	339’986	357’734

Table A.8: The table shows the messages sent and received in a local or shuffle grouping with five nodes and twenty-three CPUs.

A.2 Query

The following XML represents the full query tree used for the evaluation:

```xml
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<query xmlns="http://uzh.ch/ddis/katts/query">
        <heartBeat interval="100" />
        <fileSource id="tickerSource">
                <files>
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files0.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files1.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files2.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files3.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files4.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files5.csv.gz" />
                        <file csvFieldDelimiter=","
```

```
                                mimeType="text/comma-separated-values"
                                path="ticker_files6.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files7.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files8.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="ticker_files9.csv.gz" />
                </files>
        </fileSource>
        <fileSource id="contractSource">
                <files>
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts0.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts1.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts2.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts3.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts4.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts5.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts6.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts7.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts8.csv.gz" />
                        <file csvFieldDelimiter=","
                                mimeType="text/comma-separated-values"
                                path="contracts9.csv.gz" />
                </files>
        </fileSource>
        <tripleFilter groupOn="subject" applyOnSource="tickerSource"
                id="TripleFilter_TickerPRC">
                <conditions>
                        <condition restriction="PRC" item="predicate" />
                </conditions>
                <produces>
                        <stream id="tickerPrice">
```

```
                                <variable type="xs:double" name="ticker_price"
                                        referencesTo="object" />
                                <variable type="xs:string" name="ticker_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <tripleFilter groupOn="subject" applyOnSource="tickerSource"
                id="TripleFilter_TickerCOMNAM">
                <conditions>
                        <condition restriction="COMNAM" item="predicate" />
                </conditions>
                <produces>
                        <stream id="tickerCompanyName">
                                <variable type="xs:string" name="company_name"
                                        referencesTo="object" />
                                <variable type="xs:string" name="ticker_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <tripleFilter groupOn="subject" applyOnSource="tickerSource"
                id="TripleFilter_TICKER">
                <conditions>
                        <condition restriction="TICKER" item="predicate" />
                </conditions>
                <produces>
                        <stream id="tickerSymbol">
                                <variable type="xs:string" name="ticker_symbol"
                                        referencesTo="object" />
                                <variable type="xs:string" name="ticker_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <tripleFilter groupOn="subject" applyOnSource="contractSource"
                id="TripleFilter_ContractAgency">
                <conditions>
                        <condition restriction="agencyid" item="predicate" />
                </conditions>
                <produces>
                        <stream id="contractAgency">
                                <variable type="xs:string" name="agency_name"
                                        referencesTo="object" />
                                <variable type="xs:string" name="contract_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <tripleFilter groupOn="subject" applyOnSource="contractSource"
                id="TripleFilter_ContractAmount">
                <conditions>
                        <condition restriction="obligatedamount" item="predicate" />
                </conditions>
                <produces>
                        <stream id="contractAmount">
                                <variable type="xs:double" name="contract_amount"
                                        referencesTo="object" />
                                <variable type="xs:string" name="contract_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <tripleFilter groupOn="subject" applyOnSource="contractSource"
                id="TripleFilter_ContractCompany">
                <conditions>
                        <condition restriction="vendorname" item="predicate" />
                </conditions>
                <produces>
                        <stream id="contractCompany">
                                <variable type="xs:string" name="company_name"
                                        referencesTo="object" />
                                <variable type="xs:string" name="contract_id"
                                        referencesTo="subject" />
                        </stream>
                </produces>
        </tripleFilter>
        <oneFieldJoin maxBufferSize="20"
                joinPrecision="20000" joinOn="ticker_id" id="Ticker_Join">
                <consumes>
                        <stream maxBufferSize="5" streamId="tickerSymbol">
                                <variableGrouping>
                                        <groupOn variableName="ticker_id" />
                                </variableGrouping>
                        </stream>
                        <stream maxBufferSize="5" streamId="tickerPrice">
                                <variableGrouping>
                                        <groupOn variableName="ticker_id" />
                                </variableGrouping>
                        </stream>
                        <stream maxBufferSize="5" streamId="tickerCompanyName">
                                <variableGrouping>
                                        <groupOn variableName="ticker_id" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <produces>
                        <stream id="tickerStream">
                                <variable type="xs:string" name="ticker_symbol"
                                        referencesTo="ticker_symbol" />
                                <variable type="xs:double" name="ticker_price"
                                        referencesTo="ticker_price" />
                                <variable type="xs:string" name="company_name"
                                        referencesTo="company_name" />
                        </stream>
                </produces>
        </oneFieldJoin>
        <oneFieldJoin maxBufferSize="20"
```

```
                joinPrecision="20000" joinOn="contract_id" id="Contract_Join">
                <consumes>
                        <stream maxBufferSize="5" streamId="contractAgency">
                                <variableGrouping>
                                        <groupOn variableName="contract_id" />
                                </variableGrouping>
                        </stream>
                        <stream maxBufferSize="5" streamId="contractAmount">
                                <variableGrouping>
                                        <groupOn variableName="contract_id" />
                                </variableGrouping>
                        </stream>
                        <stream maxBufferSize="5" streamId="contractCompany">
                                <variableGrouping>
                                        <groupOn variableName="contract_id" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <produces>
                        <stream id="contractStream">
                                <variable type="xs:string" name="contract_id"
                                        referencesTo="contract_id" />
                                <variable type="xs:double" name="contract_amount"
                                        referencesTo="contract_amount" />
                                <variable type="xs:string" name="company_name"
                                        referencesTo="company_name" />
                                <variable type="xs:string" name="contract_agency"
                                        referencesTo="agency_name" />
                        </stream>
                </produces>
        </oneFieldJoin>
        <partitioner slideSize="P1D" windowSize="P20D"
                partitionOn="ticker_symbol" aggregateOn="ticker_price"
                id="Partitioner">
                <consumes>
                        <stream maxBufferSize="5" streamId="tickerStream">
                                <variableGrouping>
                                        <groupOn variableName="ticker_symbol" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <produces>
                        <stream inheritFrom="tickerStream" id="tickerStreamMinMax">
                                <variable type="xs:double" name="ticker_min"
                                        referencesTo="min" />
                                <variable type="xs:double" name="ticker_max"
                                        referencesTo="max" />
                        </stream>
                </produces>
                <components>
                        <minPartitioner />
                        <maxPartitioner />
                </components>
        </partitioner>
```

```xml
        <expressionFunction expression="#ticker_max / #ticker_min"
                id="expressionFunction">
                <consumes>
                        <stream maxBufferSize="5" streamId="tickerStreamMinMax">
                                <variableGrouping>
                                        <groupOn variableName="ticker_symbol" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <produces>
                        <stream inheritFrom="tickerStreamMinMax" id="tickerStreamfct">
                                <variable type="xs:double" name="ticker_fct"
                                        referencesTo="result" />
                        </stream>
                </produces>
        </expressionFunction>
        <expressionFilter expression="#ticker_fct > 2"
                id="expressionFilter">
                <consumes>
                        <stream maxBufferSize="5" streamId="tickerStreamfct">
                                <variableGrouping>
                                        <groupOn variableName="ticker_symbol" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <produces>
                        <stream inheritFrom="tickerStreamfct" id="filteredTickerStream" />
                </produces>
        </expressionFilter>
        <temporalJoin id="TemporalJoin_TickerAndContractData"
                parallelismWeight="10">
                <consumes>
                        <stream maxBufferSize="5" streamId="contractStream">
                                <variableGrouping>
                                        <groupOn variableName="company_name" />
                                </variableGrouping>
                        </stream>
                        <stream maxBufferSize="5" streamId="filteredTickerStream">
                                <variableGrouping>
                                        <groupOn variableName="company_name" />
                                </variableGrouping>
                        </stream>
                </consumes>
                <evictBefore>
                        <evict from="filteredTickerStream" on="filteredTickerStream"
                                if="#from.endDate lt #on.endDate" />
                        <evict from="contractStream" on="filteredTickerStream"
                                if="#from.endDate lt #on.startDate" />
                        <evict from="filteredTickerStream" on="contractStream"
                                if="#from.startDate lt #on.startDate" />
                        <evict from="contractStream" on="contractStream"
                                if="(#from.endDate + 20*24*3600*1000) lt #on.startDate" />
                </evictBefore>
                <sameValue onField="company_name" />
                <produces>
                        <stream id="tickerAndContractStream">
                                <variable type="xs:string" name="ticker_symbol"
                                        referencesTo="ticker_symbol" />
                                <variable type="xs:string" name="company_name"
                                        referencesTo="company_name" />
                                <variable type="xs:string" name="ticker_fct"
                                        referencesTo="ticker_fct" />
                        </stream>
                </produces>
        </temporalJoin>
        <fileOutput filePath="output.csv" id="fileOutput">
                <consumes>
                        <stream maxBufferSize="5" streamId="tickerAndContractStream">
                                <shuffleGrouping />
                        </stream>
                </consumes>
        </fileOutput>
        <termination />
</query>
```

List of Figures

List of Tables