Master Thesis

February 2, 2014

Combining Streams of Linked Data with Rich Background Data

Impact of the Inverse Cache on Recall and Response Time

Frank Neugebauer
of Lichtenstein, Germany (06-711-808)

supervised by
Professor Abraham Bernstein
Dr. Thomas Scharrenbach

University of Zurich
Department of Informatics

Dynamic and Distributed
Information Systems

Master Thesis

Author: Frank Neugebauer, frank.neugebauer@gmail.com

URL: http://www.ifi.uzh.ch/ddis.html

Project period: 02.08.2013 - 02.02.2014

Department of Informatics, University of Zurich

Acknowledgements

I would like to thank everyone who helped me on my way to the completion of this thesis and whose constant support rendered this work possible.

My sincere thanks goes to Prof. Dr. Bernstein for the opportunity to work and research under his patronage. I am especially grateful for the support from my tutor Dr. Scharrenbach. His advice and supervision were invaluable during the whole process of this thesis.

I would also like to thank those who read this thesis several times and offered their advice and time in order to smooth out the last ridges. Many thanks for all the emotional support and encouragement from my friends and family who helped to keep me going through the months of work on this thesis.

Finally, I would like to express my most heartfelt thanks and gratitude to my parents for their patience, love and support.

Abstract

Stream processing engines often need to adhere to QoS contracts during their operation. As they also query external data sources for supplemental information, they might not be able to receive all results in time.

This thesis proposes and implements a local cache for the Esper complex event processing engine. This 'inverse cache' stores the results of Esper's background queries that complete after the Esper query timed out and provides this data for subsequent queries.

The evaluation of the inverse cache shows that it enables Esper to receive additional external results, leading to a higher recall and faster processing time.

Zusammenfassung

Bei der Verarbeitung von Datenströmen sind häufig bestimmte Dienstgüten einzuhalten. Zudem kann Esper bei der Verarbeitung auch externe Quellen anfragen. Hierdurch kann allerdings eine Situation entstehen, in der die Verarbeitung fortgesetzt werden muss, obwohl noch nicht alle Ergebnisse eingetroffen sind.

Diese Masterarbeit schlägt den Einsatz eines “inversen Zwischenspeichers” vor, welcher verspätete Resultate sammelt und diese bei darauffolgenden Anfragen an Esper zurück gibt und implementiert einen solchen Speicher.

Die Evaluation des Zwischenspeichers zeigt, dass Esper hierdurch vollständigere Resultate erhält und die Ausführungszeit verringert wird.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Motivation

This master's thesis aims at creating a cache for streaming background data and evaluating the impact of such a local cache on the response time and recall. This is based on the current trend of accessing and consuming *Linked Open Data* through streams [CCG10, BBC10,KCF12,AF11,LpDtPH11], which uses highly efficient means of analysing the main data stream. The access of additional data, also referred to as 'background data', has not undergone the same amount of academic and professional scrutiny so far, thus this research into the performance implications of using a lookup cache for said background data.

While such cache-systems are usually realized with a local, in-memory hash-map this approach is limited in so far as that distributing the main stream workload over different machines would also need the local cache to be distributed and shared between other machines. Furthermore, a in-memory cache will also suffer from the problem of storing all the necessary data in a limited amount of memory. This leads to either the need to overwrite some data, when the maximum cache size is reached or to increase the maximum memory capacity reserved for the cache. Both variants have their own obvious drawbacks. The focus of this thesis will not comprise the problems arising from the distribution of a cache system across several systems. Instead, the influence of different cache sizes and enforced *Quality of Service* contracts on streaming RDF data and external background stores will be examined. Nevertheless, the endeavour to accomodate the scaling-out of such a system will be made in the course of this work.

Additionally, the current overall trend [NSB13,FSB13] for extensive computation moves towards distributed and parallel computing. An example are clusters, that are divided into a number of nodes that each work on a subset of the original computational problem, while only sharing limited resources among each other. A local cache on each node might assist in their work, but each node could only cache information that it has accessed before. So the overall efficiency of several local cache systems would be lower than for a cache system that keeps all the records in a centralized location.

Another problem that arises from querying external data stores for supplemental data is that these queries might take an unspecified amount of time to complete before a result is returned. This becomes problematic in high-paced computing environments that have to enforce certain time-outs and *Quality of Service* (QoS) requirements of their own. These systems have to either rely on the external store to answer fast enough with a complete set of results or to use only the data that has been received up to a certain point in time and then cut off the transmission - discarding all the data that arrived too late for the processing engine.

To alleviate the problem of unsatisfying results due to time-outs, this thesis proposes the use of a specialized caching system (dubbed *inverse cache*) for static background data that will store external information that has been queried before. This system should also store information that arrived after the cut off time of the main stream's engine and provide said information for all subsequent lookups in a more timely fashion. So it is unlikely that the cache would increase the response time of recall for the first queries, but it should help provide faster and better results for subsequent queries.

1.2 Task description

The main task for the thesis lies in the creation of an evaluation environment for testing the impact of a caching system while taking current methods for combining data streams with static background data into account. A focus of this exploration is the possible improvement of response times and the relation between response time and recall.

This evaluation environment will be using the *Esper* stream-processing engine, while the external storage will be realized using a triple store like *Apache Jena Fuseki*. The local caching system is to be implemented using *Apache Cassandra*, as this system can easily be scaled to a variable number of computing nodes in the future.

The evaluation is to be performed with a proven benchmark and a widely available data set. In this case the SR-Bench benchmarks are to be implemented in the *Esper* engine, which uses the weather observations taken from sensors during storms. The background data store will hold a subset of data from Geonames, so that additional queries can be performed.

1.3 Goal and hypothesis

The goal of this thesis lies in the evaluation of the influence of a caching system for background data, that can be used for streaming engines as well as in the dimensions recall and concerning response time.

Thus there are two main hypotheses which will be tested against the results from chapter 5 on page 33ff.

1. A *Complex Event Processing* stream engine with additional cache will be able to work more rapidly due to a faster lookup of background data through the cache.
2. A *Complex Event Processing* stream engine with additional cache will yield more comprehensive results on the background data queried, given time constraints for the response from the background data storage.

1.4 Overview of the following chapters

The following work is structured into seven main chapters, which will present the theoretical foundation and groundwork for this thesis first. This encompasses an introduction into Linked Data, stream processing, CEP, benchmarking and the software solutions *Esper* and *Cassandra*. The next chapter will describe the implementation in general and certain key points in increasing detail. The design decisions that were made during the creation of the software are also outlined here. Furthermore this chapter will present a few code samples for analysis.

After these preparing chapters the succeeding chapter will describe the background for the evaluation: What will be tested, how will it be tested and what is the objective of these tests. These questions are answered through the proposed test cases. Finally, the underlying computer system for the tests is described.

The next chapter lists the raw results from the earlier test cases and visualizes relevant parts of the obtained data. The complete raw data from the test runs can be found on the accompanying optical disc, as it would be too extensive to be directly included in this work. The next chapter offers an interpretation and discussion of the results attained. Furthermore, work related to this thesis is presented and the connections between this thesis and the body of work surrounding it are considered.

The conclusion of the thesis presents the overall results and summarized findings. Additionally, an outlook for supplemental work is given.

Chapter 2

Fundamentals

The following chapter will lay out the basis for this thesis. It will mainly encompass a short explanation of Linked Data, RDF, data streams, events and their processing, the difference between traditional data bases and triple stores and discuss a few software packages that are being used in the context of this thesis in closer detail. Furthermore, it will introduce the basic ideas behind caching systems and the different approaches to benchmarking. Lastly, the different data sets that were used during the evaluations will be introduced.

2.1 Linked Data / RDF

With the advent of the internet the sharing of information has entered a new era. While information exchange was mainly undertaken by humans and facilitated by computers, the last decade bears witness to a different trend: computers are no longer just used as helpers in acquiring raw information for the operator to process, but we try to use them more and more as suppliers of knowledge or inferred information.

An example of this movement is *Linked Data*, which is structured data that draws its usefulness from being interlinked with other data sources. Linked Data does not only represent data values as an answer to a query, but also the relationship between the elements. These are often called triples, which are tuples of the length three, and consist of subject, predicate and object.

As *Linked Data* is aimed for consumption by computers, it is mostly formatted using the *Resource Description Framework* (RDF) data model, which is based on XML and URIs. The RDF model can be represented in a number of formats like N3, RDF/XML or Turtle. The term *semantic web* is closely interlinked, as it mainly consists of a critical mass of *Linked Data*, that enables users to gain new insights through the relationships expressed through *Linked Data*.

2.2 Processing streaming data

Data streams are different from conventional data, in so far as a data stream is considered to be an unbound sequence of data, while normal data and its transmission are characteristically marked by static start and end. Thus streaming data is defined as a continuous, unlimited sequence of discrete data elements that are created as varying points in time [BBD$^+$02] and might also arrive in such a fashion at the receiver.

These kinds of data streams arise from various sources; among them are sensor networks, the monitoring of computer networks, stock and other financial markets and many more [TGB05]. Often data streams are generated by sources that ideally demand an almost immediate response by the receiver (e.g a Tsunami warning), which are hard to handle for traditional *Database Management Systems* (DBMS).

The underlying DBMS structures were created with the processing of business data in mind - this means that they are geared towards the storage of vast quantities of data, for which only the current value of the data is important. DBMS expect a persistent data set, which is also evident by the atomic processes of changing values in a DBMS [BSW01] while a real-time response to queries is not important. Additional features like triggers and alerts were added later in the development of DBMS, which explains why many of these systems fail to scale a large amount of trigger statements [HCH$^+$99] which would be necessary to accommodate the needs of data stream processing.

In contrast *Data Stream Management Systems* (DSMS) execute continuous queries over data streams, that are only stored as long as processing dictates; a DBMS stores data while a DSMS stores queries [BBD$^+$02]. Furthermore DSMS will need to process a large number of streams in a short amount of time and be able to extract events from a window (e.g. all sensors that reported a certain value during the last five minutes). These requirements also merge into the domain of *Complex Event Processing* (CEP), which aims at extracting relevant information and detecting events from streams of data [LF98].

2.3 Complex Event Processing

Event processing analyses information streams that contain events; *Complex Event Processing* CEP supersedes this concept by combining data from several sources to infer additional knowledge not originally represented by the received data [Dek07, Luc02]. Event processing is often employed when dealing with a high volume of data, especially if this data is constantly being generated and needs to be analysed in near real time. Application areas for this are manifold, but traffic analysis (car and computer alike), financial markets and medical analysis are among the most popular ones in scientific exchange [CM12, ABNS06, SLG08].

CEP furthermore uses additional data from outside sources to aggregate the process with extra information and for inference of new events [WLLB06]. Other features that can be found in CEP systems regulate the *Quality of Service* (QoS), which enforces restrictions on the acceptable response time and throughput for answers. This becomes a major problem when a CEP system needs to query outside sources for additional information, which can help the system in generating meaningful complex events, but also has to provide said events to a consumer in a given time frame.

2.3.1 Esper

Esper is an open source streaming data processing engine that provides event processing services and is being developed by *EsperTech*[1] since 2006. *Esper* is available in Java and C# and offers several ways to represent events: either through getter methods of an object, by objects that implement a *hashMap* interface or through an XML representation.

Esper uses subscriptions by offering to attach listeners, which are invoked when an *Esper* statement match is found in the current data stream. Such statements are written in *Esper's* own *Event Processing Language* (EPL), which bears resemblances to the classic *Structured Query Language* (SQL): *SELECT* clauses in *Esper* describe the event properties to return, *from* clauses define the named event stream which is to use, while *where* clauses specify constraints. Additionally, *Esper* offers aggregation, grouping and ordering functions.

Esper is also able to combine several event streams, either by using *insert* clauses or by joining patterns. These *joins* are also able to encompass SQL based databases with relational data.

Another powerful feature of *Esper* is the possibility to define different views on the underlying data event stream. This is done by using *windows* which can define temporal and sequential relations between the events and are able to significantly alter the processing engine's behaviour: Examples for this are the sliding or tumbling event windows that *Esper* offers. While the sliding windows keep either a certain number of events or all events in a certain time frame in the window and advance the window with each new event (e.g. queries that return the highest temperature that has been measured within the last 30 minutes), tumbling windows store all Events until the window meets an external constraint and releases all stored events to the engine for processing. An example for this would be the implementation of a query that returns the highest measured temperature once in an hour.

Furthermore, *Esper* offers support for SQL-like subqueries, the setup of various event output frequencies and event patterns, that are used to describe relations between subsequent events. Another interesting feature of *Esper* is the possibility to run *Esper* in simulated time. This can be very useful for testing and benchmarking, as historical data can be used for this purpose. The use of simulated time demands an external clock that

---

[1] EsperTech Inc., PO Box 3129, Wayne, NJ 07474-3129, USA

advances the internal *Esper* time, but which can also be used to speed up the processing of lengthy historical data. A more detailed description of *Esper* can be found in the *Esper* documentation [Esp14].

2.4 Triple store and (C-)SPARQL

A triple store is a database with the purpose of storing and retrieving triples of information composed of subject, predicate and object (e.g. "Adam knows Eve"). These stores can be purpose built from scratch or on top of existing relational data bases; while both variants allow for the querying of the underlying RDF-graph, native triple stores are mostly responding faster to queries against them [BS09a].

Queries against a RDF-graph are mostly formulated in the *SPARQL Protocol and RDF Query Language* (SPARQL). SPARQL is the *W3C* recommendation for a RDF query language as well as protocol and supports querying multiple graphs [CCeaD13]. While SPARQL is still in development, it has already been implemented in several programming languages. One of the biggest advantages of SPARQL is that it allows unambiguous queries by using prefixes in the query to describe different ontologies.

An extension of SPARQL is C-SPARQL, which was built to support continuous queries against RDF data streams. The main reason for the proposal of C-SPARQL lies in the static - or only very slowly changing - nature of the underlying RDF graphs that are targeted by SPARQL queries; the extension made by C-SPARQL aims at combining the nearly static data from RDF triple stores with fast changing data from RDF data streams. C-SPARQL, just like EPL, uses logical windows to aggregate data events over a certain time, in order to allow for inference between this data [Bar09].

2.5 Cassandra

Apache Cassandra is an open-source nested key-value store originally developed by Facebook,[2] which has been implemented as a highly scalable distributed database. Generally *Cassandra* is counted to the so called "NoSql" data stores. In contrast to traditional DBMS these data stores are built to serve a high number of concurrent users while also allowing the insertion of new data at the same time. This is achieved by scaling-out *Cassandra* over the necessary number of servers, which will replicate and distribute the stored data according to the *Cassandra* configuration. Thereby, the data store performs like a peer-to-peer network in which data is distributed in a way that no single point of failure exists. *Cassandra* can hold all or only partial amounts of data in memory.

A downside of this approach is the "eventual consistency" across nodes. While a traditional DBMS will adhere to the ACID principles and prevent inconsistent states, this is

---

[2] Facebook Inc., 1601 S. California Ave, Palo Alto, CA 94304

not easily possible on distributed systems given the inherent latency of communication between them. This is also explained by Brewer's CAP theorem [Bre00] that states that it is impossible for a distributed computer system to simultaneously provide consistency, availability and partition tolerance.

Consistency in this regard is understood to be the guarantee that all nodes have the same data available at the same time, while availability concerns the guarantee of an assured response to a query. Lastly, partition tolerance denotes the resilience of the system against arbitrary data loss due to communication or hardware failures in the distributed system [GL02].

Cassandra employs no locking mechanism, so it abandones the first point of the CAP theorem, but offers automatic discovery for new nodes and detection of dead nodes and spreads the data over various nodes, guaranteeing the second and last point of the theorem. A kind of weak consistency assurance can be achieved by reading the same value from different nodes and majority voting, but this is unsatisfactory, as it reduces the performance and does not guarantee consistency.

Several benchmarks hint at *Cassandra* having a higher throughput than comparable NoSQL stores like HBase, Redis or VoltDB, especially in environments with mixed read-/write queries [TB11, RSMM12]. This is also partly due to its elasticity, which allows *Cassandra* to spread an increasing number of read/write queries linearly across a large number of nodes [RSMM12].

Furthermore, *Cassandra* employs its own query language CQL (*Cassandra* Query Language) which offers a SQL-like syntax and similar keywords for setting up keyspaces (databases) and tables in *Cassandra* itself. It should be noted, that this is an additional level of abstraction, that does not correspond to the actual underlying internal implementation in *Cassandra*.

2.6 Cassandra-Esper

Cassandra-Esper is a Java library that is being developed by Dr. Scharrenbach at the Department of Informatics at the University of Zurich. It acts as a middleware between *Esper* and *Cassandra* and provides an implementation of the *Esper Virtual Data Window* (VDW) that can be used to perform EPL queries against the *Cassandra* data store. With this library the external *Cassandra* store can be transparently connected to *Cassandra* which means that this library can be used to easily implement an *inverse cache* for existing *Esper* projects.

As of yet, it is unknown what kind of performance implications the use of this library and the *Esper* VDWs might have.

2.7 Inverse cache

Caching systems are widely known and employed in computer science and information technology. They all rest on the assumption that storing a certain, often used datum in a rapidly accessible place can speed up processing, due to the reduced time to fetch said datum. The higher the frequency of accesses to this datum, the higher the amount of stored data in the cache, the longer the validity of the datum in the cache, and the lower the response time, the more efficient the cache will be in accelerating the main process.

This work proposes the *inverse cache* which aims at speeding up CEP on streaming data - that is being enriched by external data - by providing a fast answering store for queries to external data sources. As most data in external stores is considered slowly changing in comparison to the information in the data stream, a cache should be able to hold a significant number of valid entries.

Another point in favour of a caching system is that many external queries to stores like DBpedia are not only slow, but might also limit the amount of external queries they answer in a given time frame to reduce the strain on their own infrastructure. A caching system can ensure that these external stores are not queried unnecessarily by holding a local copy.

Another problem concerning *Complex Event Processing* is the unbound nature of responses. A query for all cities near a certain geographical landmark might yield an unpredictable number of possible results and these might also take an unforeseeable time to be returned. CEP systems that need to fulfil certain QoS criteria, like a maximum time to respond to an incoming event, might not receive all the applicable data from a background store in time. This problem can be alleviated by the *inverse cache*, as it can still accept external data while the processing engine already moved on to the next data events. This way, the next query for this particular information can be answered by the cache without the drawbacks that the external store exhibits.

On the other hand a cache might also slow down processing in certain circumstances - namely if the main process is trying to look up a datum in the cache which does not exist. Such a *cache miss* will force the CEP engine to query the external store nevertheless. A caching system can aim at minimising the amount of cache misses, respectively at maximising the amount of cache hits by keeping the relevant data as long as it might be needed.

2.8 Benchmarking basics

A benchmark assesses the relative performance of an entity (be it an object or process), often by comparing it to different entities. There are five major factors, that all benchmarks should exhibit:

1. Relevancy
2. Repeatability
3. Fairness
4. Verification
5. Economical

These traits are supposed to make sure, that the benchmark focusses on an important aspect (relevancy), that it will produce similar results again (repeatability), that all participants can participate equally (fairness), that the results can be checked (verification) and that the test can be run without high investment of resources (economical). A more detailed explanation and many examples can be found in [Hup09].

Benchmarks often focus on certain *Key Performance Indicators* (KPIs) to meet the relevancy criterion, as these indicators should represent a simplified yet accurate representation of how the system in question is performing. Examples for such KPIs that are often used for DBMS and DSMS/CEP systems are throughput (number of records during a certain time) and latency (time from query to response) [RSMM12]. While these *are* major KPIs, these are often generated as an average value for the whole system. This can be deceiving, as a system with an unreliably spiking latency might get a better average latency than a system without such extreme behaviour. In regards to a possible application in the QoS domain such an averaged benchmark would be misleading. Thus, one should strive to report the best, worst and average for such KPIs.

A popular example for a streaming data benchmark, the *Linear Road Benchmark* [ACG04], simulates a toll system on a motorway with variable tolling based on accident detection, alerts and traffic congestion. It employs a fixed set of continuous and historical queries against a stream of events, that is being generated in advance. Furthermore the *Linear Road Benchmark* is able to run in simulated time, which allows for faster or slower than real time processing. A downside to the *Linear Road Benchmark* lies in its reliance on the relational data model, which makes it unfit for assessing the performance of RDF-graph based streaming engines.

Another recent streaming data benchmark is *SRBench* [ZDCC12] which describes in itself a general purpose benchmark for streaming RDF/SPARQL engines. *SRBench* defines 17 queries in natural language that try to measure the abilities of the tested streaming engines to perform over a broad range of use cases that cover different aspects of streaming data. As synthetic data does generally not accurately predict the performance of streaming systems [DKSU11], *SRBench* uses real world data: Mainly the "Linked Sensor Data" data set from *Kno.e.sis*,[3] but also the "Geonames" and "DBPedia" data set from the Linked Open Data cloud.

---

[3] http://knoesis.org/

Chapter 3

Implementation

The following chapter illustrates the details of the implementation by outlining the basic system components and their functions. This is followed by a more detailed look on the *Esper* set up, the a priori data manipulation of the SR-Bench datasets, the set up of the background data server and the implementation of the *InverseCache* with *Apache Cassandra*. The last part selects certain aspects of the *InverseCache* and clarifies the reasoning behind these.

If not indicated otherwise the programming language used is Java.[1] *Esper* is integrated into the project through a Java jar file.

3.1 System overview

The proposed implementation consists of five main parts that will be described in the following. All programs use the following external libraries as dependencies:

- httpclient-4.2.3.jar
- esper-4.10.0.jar
- esperio-csv-4.10.0.jar
- log4j-1.2.16.jar
- commons-logging-1.1.1.jar
- cglib-nodep-2.2.jar
- antlr-runtime-3.2.jar
- jena-sdb-1.4.0.jar
- jena-tdb-1.0.0.jar
- jena-arq-2.11.0.jar
- jena-core-2.11.0.jar
- jena-iri-1.0.0.jar
- jcl-over-slf4j-1.6.4.jar
- commons-codec-1.6.jar
- httpcore-4.2.2.jar
- xml-apis-1.4.01.jar
- xercesImpl-2.11.0.jar
- log4j-1.2.16.jar
- slf4j-log4j12-1.6.4.jar

---

[1] JDK 7

- slf4j-api-1.6.4.jar
- cassandra-esper-0.3.0.jar
- json-simple-1.1.jar
- cassandra-driver-core-2.0.0-rc2.jar
- guava-15.0.jar

3.1.1 Esper

The *Complex Event Processing* engine *Esper* (see 2.3 for more information) uses a single incoming data stream as information source. This incoming data stream is parsed into events and then forwarded to the processing engine itself. Delays between the events are mapped from the real time span using a delay factor (between [0.0001;1]) to the simulated time of the *Esper* engine. This was done to ensure that the engine does not receive a constant flow of events at the maximum processing capacity of the engine, but rather that the incoming event flow dwindles and swells in a more realistic way. *Esper* is capable of processing data at a very fast rate, but with this setting it is also possible to simulate how *Esper* would perform in a real-world real-time setting.

Esper uses EPL [Esp14], a proprietary SQL-like description language for configuration, the creation of data windows - these describe the current view of the engine on the events - and for adding statements. Statements are evaluated against the stream of events as limited by the data window. In the event of a match *Esper* calls a listener, which then in turn cascades further lookups or processes. In this case the listener is programmed to look up additional data from the background store and the cache. The *Esper* engine will only continue after receiving the requested external data or a time-out has occurred.

3.1.2 Data input reader

The input reader is necessary due to constraints of *Espers* CSV file reader. Although *Esper* can handle multiple source data streams, *Esper* is currently unable to accept multiple data streams with time stamps in an interleaving fashion. At the moment, data streams will only be sorted by the first time stamp. Due to this restriction the Knoesis *Linked Sensor Observation* data set, which is split into files for each sensor id, was merged a priori into a single file. Furthermore, this file was supplemented with extra data to indicate delays between the events in question. Additionally, these input files were checked to exhibit a monotone increase in time stamp value, because the *Esper* engine is unable to handle a non-chronological flow of events.

This input reader is also the basis for the advancement of the simulated time on which *Esper's* statements are evaluated and is also responsible for introducing the delays between these events where appropriate.

3.1.3 Background store

The external store in this project is place holder for any possible external triple store. For this thesis *Apache Fuseki* had been selected, as the performance of this particular store has already been studied in multiple papers [DS12, VMS12] and should make the interpretation of the evaluation results clearer. Furthermore, *Fuseki* allows for an easy exchange with another store at a later time. The *Fuseki* store is run locally to minimize any influence of network latency.

Esper has been configured to use a *virtual data window* for queries against the *Fuseki* store; the queries return location data for the respective Knoesis *Linked Observation Data* sensor id. The cache operates under the assumption, that if the external storage returns a result, it is correct. There are no checks in place to detect fraudulent or damaged data.

3.1.4 Inverse cache

The cache is based on *Apache Cassandra*[2] and uses the standard configuration running on a single node. While an in memory hash map would yield faster results, this approach enables the scaling of the *inverse cache* across a multitude of machines in the future. All results from the background store are also written to the *inverse cache*, so that they will be available faster during the next query. Furthermore, the cache will also accept and store data from the external store after the *Esper* engine received a time-out or incomplete data from the background data source.

Esper queries the cache through another virtual data window, so that queries to the cache and the background storage are following the same program paths. This should also lead to a similar overhead for the *Esper* virtual data windows. The queries are handled by *Cassandra-Esper* 0.3[3] which is based on the DataStax[4] drivers. Due to small API changes in the newest version of *Cassandra* and the DataStax drivers some alterations needed to be made to *Cassandra-Esper*.

3.1.5 Socket logger

The socket logger is a helper process that runs independently. It offers a socket which the other program parts can send their logging data to, so that it can be written into the central log. Currently the logger keeps track of

- the overall configuration of the engine
- the current *Esper* statement
- the total count of events sent to the *Esper* engine

---

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

[3] https://bitbucket.org/scharrenbach/cassandra-esper/wiki/Home

[4] http://www.datastax.com/

- the total amount of events matching the current *Esper* statement
- the current delay factor used
- the size of the cache to be tested
- start, stop and duration time stamps for the respective queries to *Fuseki* and *Cassandra*

The logger was designed to be non-blocking and will accept and queue all external log messages, so that both *Esper* and the cache will not be influenced. The logger will also record the time stamps for each query to the cache or background store in CSV format.

3.2 Datasets

3.2.1 EsperCSV data set

In order to provide *Esper* with a stream of events, the need for a known and widely available dataset, which would satisfy the outlined criteria for testing from section 2.8, arose. The selection fell on the Kno.e.sis *Linked Observation Data* data set, which provides observations of weather stations from north America during several hurricanes (and a single blizzard) from 2003-2009. The data is subdivided temporally into the different storm periods and available at kno.e.sis.[5] Each subset contains a number of files that describe the observations for a single weather station over a given time. Note that their might be multiple files for different time ranges for the same sensor.

These data sets are based on the works from MesoWest [PHS10], a subdivision in the Department of Meteorology at the University of Utah, which provided sensor observations as comma separated values. This information was then manipulated in order to create an RDF representation of the observations and measurements [PHS10].

This base data set was deemed a fitting choice, as it is relevant to the investigation, tests are easily repeatable as the data is static, the data is fair and it provides an easy way for verification. A slight compromise is being made in regards to running the tests economically, as complex queries with large subsets of the *Linked Observation Data* might take more than 12 hours to complete. The sizes and number of triples respectivly sensor observations are noted in table 3.1 on page 16.

To make this dataset accessible to *Esper*, it needed to be transformed. All sensor observations and their current time were extracted and parsed into a single CSV file - which was sorted according to the time stamps of the sensor observations - to accommodate *Esper's* requirements: While *Esper* is able to read multiple CSV files simultaneously and can even coordinate start times between them, it falls short in cases when the data time

---

[5] http://wiki.knoesis.org/index.php/LinkedSensorData

stamps are not monotone increasing from the end of one sensor file to the start of another. This problem was mitigated by providing *Esper* with the sensor observations in a pre-sorted single CSV file. Additionally, a delay column was added to the CSV data file, which allows *Esper* to simulate a variable flow of incoming information (e.g. bursts of data that arrive roughly every 5 minutes) while negating the need to calculate said delays during the runtime.

Furthermore, the *Linked Sensor Data* data set[6] from kno.e.sis was used, as this set provides additional information for 20000 weather stations in the US. This data set was used unaltered and imported into the background store.

3.2.2 Twitter data set

Additionally, some of the tests will also be run based on a *Twitter* dataset. This data set consists of a simple comma separated values (CSV) file, which includes a time stamp, a message id, a user id and a URI. The URIs in this dataset are DBPedia resources (i.e. <http://dbpedia.org/resource/Hamster>), which offer an easy way to query the DBPedia for background information. The data provided is not unique in so far as there are duplicate URIs.

This data set is simulating a torrent of messages (tweets) that offer a convenient way for semantic annotation through their URIs. In total, this data set contains 308991 messages with up to 1923 occurrences of the same URI.

This simple set up and the variance in the amount of URIs and messages make this data set a promising candidate for tests concerning the impact of the *inverse cache*. The amount of messages in a very short time frame and the possible very high amount of data, which can be returned from Querying said URIs offers some challenges though.

This data set was not transformed (although it was sorted) before passing the events to *Esper*. As this data is static, tests are very easily repeatable and the possibility to only parse parts of this data file should also ensure that tests can be performed economically. More data on this data set can be found on the tables 5.1, 5.2, 5.3 on page 34.

The data set can also be found on the accompanying optical disc.

---

[6] Available for download at http://wiki.knoesis.org/index.php/LinkedSensorData

[8] Note that no official name exists for this storm, as it did not fit the international naming scheme of the World Meteorological Organization for tropical storms and hurricanes.

[8] http://wiki.knoesis.org/index.php/LinkedSensorData

Name	Storm type	Date	Number of triples	Number of observations	Data size
All			1'730'284'735	159'460'500	~111 GB
Bill	Hurricane	17-22.08.2009	231'021'108	21'272'790	~15 GB
Ike	Hurricane	01-13.09.2008	374'094'660	34'430'964	~34 GB
Gustav	Hurricane	25-31.08.2008	258'378'511	23'792'818	~17 GB
Bertha	Hurricane	06-17.06.2008	278'235'734	25'762'568	~13 GB
Wilma	Hurricane	17-23.10.2005	171'854'686	15'797'852	~10 GB
Kathrina	Hurricane	23-30.08.2005	203'386'049	18'832'041	~12 GB
Charley	Hurricane	09-15.08.2004	101'956'760	9'333'676	~7 GB
"Nevada Storm"[7]	Blizzard	01-06.04.2003	111'357'227	10'237'791	~2 GB

Table 3.1: Overview of the *Linked Observation Data* data sets from kno.e.sis[8]

3.3 EsperCSV solution in detail

The following section will describe the proposed solution in detail, lay out the necessary configurations, describe the queries used and explain the data and program flow.

The *inverse cache* was programmed using the IDE Eclipse[9] and the tests were run using the Java OpenJDK 7.[10]

The *Inverse Cache* Project consists of the follwing files, which will be discussed below in so far as to aid the understanding of the project. Commonly known or expected programming routines will not be elaborated upon.

- Main.java
- ReadCsvDataFiles.java
- CreateEvents.java
- StormEvent.java
- FusekiVirtualDataWindow.java
- FusekiVirtualDataWindowFactory.java
- FusekiVirtualDataWindowLookup.java
- FusekiEvent.java
- QueryFuseki.java
- QueryCassandra.java
- CEPListener.java

---

[9] Version: 3.8.1

[10] Version "1.7.0_25". OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25_2.3.10_1_ubuntu0.13.04.2) OpenJDK 64 Bit Server VM (build 23.7_b01, mixed mode)

- CEPListenerMultiThread.java
- SocketLoggerClient.java

3.3.1 Main

The main method of the *Inverse Cache* Project accepts outside parameters as well as configuration files and configures the Log4J logger, as well as the *Fuseki*, *Cassandra* and storm event *Esper* engines. Furthermore, the internal timer on the storm event engine is disabled, as this engine will be run in simulated time, while the other two run normally.

The *Fuseki* and *Cassandra* engines are configured to use the respective Virtual Data Window (VDW), which enables *Esper* to query these external sources in the same way as the primary data stream. The *Cassandra* VDW uses *Cassandra-Esper* for this access.

Additionally, the main class sets the starting time for the storm event *Esper* engine back to 01.01.1990 - this allows for the advancement of the simulated time for events that take place after this date.

Next, a continuous query in the form of an EPStatement for the storm event engine is declared and a listener (CEPListener) is attached to this statement. The EPStatement is extracted from the outside configuration file and the listener will get called if *Esper* can match the incoming event stream to the Statement.

After this set up, the main routine starts the parsing of the external events from the csv file through ReadCsvDataFiles.

3.3.2 ReadCsvDataFiles

This helper class reads the external CSV file storing the *Linked Sensor Observation* events (see part 3.2 on page 14 for an explanation) and then passes the extracted data to the *CreateEvents* class, which creates an *Esper* event and passes this event - together with a possible delay to simulate bursts of data - to the storm event *Esper* engine.

Additionally, this class keeps track of the overall number of events that were passed to *Esper*, as well as the overall timing and total delay time used while running the test. All these items are passed through a socket connection to the SocketLogger (see chapter 3.3.9).

3.3.3 CreateEvents

In this class the more specific constructor of the StormEvent class is used to build an EsperEvent. The next step checks if the Event has a valid time stamp and advances the *Esper* simulated time if necessary - errors in the time continuum will result in a critical error and the termination of the program. If a delay was specified for the current event, this class will also wait for the time delay (stretched by the global delay factor) between two events to be reached and then pass the EsperEvent to the *Esper* engine for processing.

3.3.4 CEPListener

The *Complex Event Processing* Listener (CEPListener) waits for the *Esper* engine to match the current EPStatement to the events it receives. On a positive match, this listener update method receives two EventBeans for the old and new data in the window (a view specified through the EPStatement, which holds all current matches to that statement) that fit the search criteria.

For these matching events, we extract the name of the weather station (sensorId) and create two new queries: One for the background data store based on *Fuseki* and one for the *Cassandra* Cache. Both of these queries are simple EPL statements, as these two stores will also be queries through the VDW that were created during the configuration phase in the main class. The preparation of these queries is additionally timed, to check if querying the external stores through *Esper* incurs a non-negligible performance loss.

In the next step the *Cassandra* cache is queried for supplemental data on the weather station. If the cache holds the relevant data, it is returned to the listener and a log entry with the start and end times of the query, an event number, the ID of the station that was matched and the source of the match (in this case the *Cassandra* cache) is produced and sent to the SocketLogger. If the Cache returns an empty result, the *Fuseki* result is evaluated. Both accesses are also timed and can be used for the creation of a performance profile.

A result with the initial *Esper* data as well as the supplemental data is created - although said result is not used anywhere else in the program, as this falls outside the scope of this work.

3.3.5 CEPListenerMultiThread

This class works in the same way as the CEPListener, with the exception that it uses an ExecutorService to concurrently query both stores at the same time. If the *Cassandra* result is empty, then it waits for the *Fuseki* store to finish its query and evaluates the result. While this class performs slightly faster than CEPListener, the *Future*-constructs used pose difficulties while debugging. Any error due to incorrect queries will result in a highly unspecific Java error that points to this class, even though the error actually occurred in another subclass. If changes are to be made, the author recommends using CEPListener until a working solution has been reached and then changing over to the multi thread CEPListener. It is also advisable to use the CEPListener, if external background stores that enforce a quota on queries are to be used, as the CEPListenerMultiThread will put additional strain on the external store by creating more queries.

3.3.6 FusekiVirtualDataWindow classes

The FusekiVirtualDataWindow class is part of the three classes that are defining the actions of the *Esper* VDW for *Fuseki*, which allows for the use of EPL queries against this named window. The benefit of this approach lies in the possibility to exchange the background store with another solution, while keeping the rest of the code as it stands. This loose-coupling allows furthermore to use a unified query language (EPL) for all queries against VDW without the need to know the actual set up behind.

This class is an implementation of the *Esper* VirtualDataWindow class and handles basic methods for this window (getLookup, update, destroy, handleEvent and iterator).

The factory class FusekiVirtualDataWindowFactory is used for creating FusekiVirtualDataWindows and implements the *Esper* VirtualDataWindowFactory with the basic methods initialize, create, destroyAllContextPartitions and getUniqueKeyPropertyNames.

The focus should lie on the FusekiVirtualDataWindowLookup class, which surprisingly implements the *Esper* VirtualDataWindowLookup, as this class handles the actual communication with the outside *Fuseki* store. As the external store will only be queried for data, this class only implements the lookup method through a call of the QueryFuseki class. The returned values are then stored into a HashMap from which an *Esper* EventBean is created (through an EventFactory and a data wrapper) and passed back to the engine.

3.3.7 QueryFuseki

This part of the program deals with building and executing the SPARQL query against the *Fuseki* store. Currently it expects the *Fuseki* store to answer SPARQL queries directed at the endpoint *http://localhost:3030/ds/query* or *http://localhost:3030/ds/sparql*, but this can also be moved to a different endpoint. The actual queries transport is handled through the *Jena* libraries, while error handling and extraction of data from the query result happens in the QueryFuseki class.

Additionally, each returned *Fuseki* query - be it empty or containing results - is forwarded and inserted into the *Cassandra* cache. This means that the *Cassandra* cache would always receive all the data that is returned from the *Fuseki* store.

3.3.8 QueryCassandra

The QueryCassandra class is the counterpart to the QueryFuseki class, as it offers access to the *Cassandra* cache via the *Datastax 2.0 Cassandra driver*. Furthermore, this class makes sure that *INSERT* queries into the *Cassandra* cache respect the size of the cache, as defined during start up. If the cache is already full, then this class will ensure to delete the oldest entries before inserting new ones. See section 3.4 for more information on the cache.

3.3.9 SocketLogger

The SocketLogger is split into a client part and a server component which writes the actual log file. The logging server is run independently and provides a socket. The client component connects to this socket and offers a simple method of writing messages through the socket to the log file on the server. The socket server is also able to handle multiple incoming connections, which will all write into the same log file.

3.4 Implementation of the inverse cache

One core component of this project is the *inverse cache*, which was realized through the implementation of a *Apache Cassandra*[11] database. The purpose of the *inverse cache* is to store results from the external background store and respond to queries faster than the background store. The cache also includes results that arrive from the external store after the primary query was aborted due to time constraints (i.e. query time-outs or QoS requirements), so that these "slow" results can be kept "at the ready" in the *inverse cache*. Another possible benefit of the cache is the possibility to store complete results, which in turn would allow the *Esper* engine to forego the need to query the background store at all.

In this implementation and in all experiments the *inverse cache* was run on the same host as the *Esper* engine and queried through a local connection, in order to minimize potential latency impact. It should be noted, that running *Cassandra* in a local network should only have a minimal performance impact and that a single cache can supply results to multiple partners at the same time. For multiple machines or clusters of streaming engines that are separated through high-latency networks (100+ ms RTT[12]) it would likely be advisable to deploy a single *Cassandra* instance per network and possibly even to interconnect these cache instances, so that *Cassandra* is able to synchronize the cache across the different locations. In case that a single *Cassandra* instance would not be able to handle all the traffic, *Cassandra* also offers linear scalability over multiple instances which facilitates the increase of possible transactions.

In this case, *Cassandra* was configured to not use any form of authentication, as only queries from the trusted local network are possible. The cache uses the keyspace cassandra with the table cassandrawindow and accepts text as primary key and coordinate values with an index on the coordinates. The script used to set up the *Cassandra* store with additional settings can be seen in listing 3.1 on page 22. The actual connection to the *Cassandra inverse cache* is handled by the Datastax Java driver in version 2.0.0-rc2 which is provided by Datastax.[13]

The maximum size of the cache is managed through the Java code: Before new items are added to the cache, a query determines the current size and compares it with the

---

[11] Version 2.0.4

[12] Round Trip Time

[13] https://github.com/datastax/java-driver

maximum allowed cache size. In case the new elements would not fit into the cache without violating the maximum size, some elements would be removed from the cache. At the moment, this is implemented through a first-in-first-out (FIFO) scheme which is based on the internal *Cassandra* time stamp for each cache entry, that can be explicitly queried but is not reported back in normal *SELECT* or *COUNT* queries. As *Cassandra* returns results in an unsorted fashion[14] it is necessary to sort the cache in order to select the oldest queries that can be removed. The selection query and sorting are shown in listing 3.2 on page 22.

---

[14] as long as not explicitly specified otherwise

```
CREATE KEYSPACE cassandra WITH replication = {
  'class': 'SimpleStrategy',
  'replication_factor': '3'
};

USE cassandra;

CREATE TABLE cassandrawindow (
  cassandraid text,
  coord decimal,
  PRIMARY KEY (cassandraid)
) WITH
  bloom_filter_fp_chance=0.010000 AND
  caching='KEYS_ONLY' AND
  comment='' AND
  dclocal_read_repair_chance=0.000000 AND
  gc_grace_seconds=864000 AND
  index_interval=128 AND
  read_repair_chance=0.100000 AND
  replicate_on_write='true' AND
  populate_io_cache_on_flush='false' AND
  default_time_to_live=0 AND
  speculative_retry='99.0PERCENTILE' AND
  memtable_flush_period_in_ms=0 AND
  compaction={'class': 'SizeTieredCompactionStrategy'} AND
  compression={'sstable_compression': 'LZ4Compressor'};

CREATE INDEX cassandrawindow_coord_idx ON cassandrawindow (coord);
```

Listing 3.1: Creation script for the Cassandra cache

```
public void reduceCacheSize(int maxSize) {

        // Code to set up session and connection to Cassandra

        // create a set, that is sorted by the value and not the key
        SortedSet<Map.Entry<String, Long>> sortedset = new TreeSet<Map.
            Entry<String, Long>>(
               new Comparator<Map.Entry<String, Long>>() {
                  @Override
                  public int compare(Map.Entry<String, Long> e1,
                        Map.Entry<String, Long> e2) {
                     return e1.getValue().compareTo(e2.getValue());
                  }
               });
        SortedMap<String, Long> rowMap = new TreeMap<String, Long>();

        // Cassandra query string with explicit time stamp
        ResultSet cassResultSet = session.execute("select cassandraid,
            writetime(coord) from cassandraWindow;");

        // extract and add relevant information to the rowMap
        while (!cassResultSet.isExhausted()) {
            Row myRow = cassResultSet.one();
            String sensorId = myRow.getString(0);
            long timestamp = myRow.getLong(1);
            rowMap.put(sensorId, timestamp);
            }

        // add rowMap to the sorted set
        sortedset.addAll(rowMap.entrySet());

        // determine cache size
        int resultSize = sortedset.size();

        //removing items by age
        if (resultSize > maxSize) {
            while (resultSize>maxSize) {
                // remove first item from cache
                session.execute(String.format("Delete from cassandraWindow
                    where cassandraid='%s'", sortedset.first().getKey()));
                // remove first item from sortedSet
                sortedset.remove(sortedset.first());
                // decrease resultSize by 1
                resultSize--;
                }
        }
    // terminating connections
    }
```

Listing 3.2: Extract from QueryCassandra.java showing the removal of old elements

One trade-off the cache faces is the decision whether to check if an item exists in the cache and then query the background store, or to start queries to both stores at the same time. The first approach is advisable in order to reduce strain on the external store, even though it might negatively influence the overall return time for background data for a query. This impact would be more pronounced for smaller cache sizes or bigger variances in queries.

The second variant of querying both data sources at the same time creates unnecessary traffic and consumes resources needlessly, if the cache already stores the complete results for the query and returns these faster than the external store. Nevertheless, this approach will theoretically ensure the fastest response time possible. This thesis implements both variants in the CEPListener and CEPListenerMultiThread classes.

3.5 Twitter inverse cache

The *Twitter inverse cache* program was created in order to focus on the investigation of the impact of the *inverse cache* for bigger amounts of data, especially in cases with QoS contracts as a limiting factor. In order to do so, the *Twitter inverse cache* uses many of the features that were already developed for the EsperCsv client. Namely the ReadCsvDataFiles class, the logging components and many of the settings and configurations that were used for the EsperCsv client.

For this reason only the particularities of this additional testing program will be described. Compared to EsperCsv the *Twitter inverse cache* offers the possibility to enforce time-outs for the queries to the background store and the *inverse cache*. Additionally, and in order to guarantee the fairness of the evaluation, the maximum number of events that should be parsed by the ReadCsvDataFiles class can be limited.

In order to generate a larger stress on the *inverse cache*, the *Esper* engine's EPL statement was configured to match every incoming event, which in turn also requires the program to query the background store (in this case DBPedia) and *inverse cache* for every event. Additionally, the queries to the background store were implemented as *DESCRIBE* queries, which return all known information for the given resource. Such a result might encompass tens of thousands of result triples.

The QoS time-out aspect was designed in a way that incoming information from the background store will only be added to the collection of results until the time-out is reached. After this cut-off time, all additional events will be forwarded to the *inverse cache*. That way the cache will exclusively hold data that has at least returned late once during the runtime of the program.

A problem that needed solving was the burst of data, that might block the whole network stack; simply forwarding all elements as soon as they arrived after the cut-off time resulted in a very high count of concurrent threads that all tried to execute *INSERT* queries simultaneously against the *Cassandra* store. This lead to very high memory and CPU requirements during the runtime, which also impacted the performance of the main program, which in turn distorted the time measurements. Additionally, *Cassandra INSERT* queries might time-out, which lead to an undesirable state of the cache.

This problem was tackled by providing a *ConcurrentLinkedQueue* which buffers all data that has to be inserted to *Cassandra*. An additional thread then queries this buffer repeatedly and inserts data one after another into *Cassandra*. This way there is only one entity per query that needs to communicate with *Cassandra*, thus greatly reducing the load on both the client and on *Cassandra* as well. The implementaion of this scheme can be seen in the two listings 3.3 and 3.4.

Unfortunately, *Cassandra* does not accept *INSERT* queries with multiple value sets. A future improvement for large amounts of data would be the use of *sstables* instead of single *INSERT* queries, which should result in a much faster rate of data insertion.

```java
final ConcurrentLinkedQueue<Triple> myQueue = new
   ConcurrentLinkedQueue<Triple>();
Thread thr = new Thread(new Runnable() {

   public void run() {
      while (true) {
         Triple qt = myQueue.poll(); // getting new triples from
            the queue
         if (qt != null) {
            String queryPreHash = String.format("%s%s%s",
                  qt.getSubject(), qt.getPredicate(),
                  qt.getMatchObject());
            HashCode hash = Hashing.goodFastHash(64).hashString(
                  queryPreHash, Charsets.UTF_8);
            String cassandraQuery = String
                  .format("INSERT INTO twittercache (hash, res, s
                     , p, o) VALUES ('%s', '%s', '%s', '%s', '%s
                     ')",
                        queryPreHash, res, qt.getSubject(),
                        qt.getPredicate(), qt.getMatchObject());
            try {
               session.executeAsync(cassandraQuery); // Executing
                   the asynchronous insert query
            } catch (SyntaxError e) {
               // Ignore errors because of EOF in triples
            }
         } else {
            try {
               Thread.sleep(100);
            } catch (InterruptedException e) {
               // Interrupts are used to terminate this thread
            }
         }
      }

   }
});
thr.start(); // Starting the worker thread
```

Listing 3.3: Extract from QueryDBPediaDescribe.java showing the asynchronous communication with Cassandra

```
if (elapsedTime > timeout) {
//      long start = System.nanoTime();

                  myQueue.add(qt); // adding late comers to the queue
                  long end = System.nanoTime();
                  queueDuration=queueDuration+(end-start);
                                   ⋮
                   }
```

Listing 3.4: Snipplet from QueryDBPediaDescribe.java showing the adding of elements to the queue and the compensation of the total reported time through the queueDuration variable

Chapter 4

Testing

This chapter outlines the motivation and reasoning behind the evaluation. First, the setup of our fundamental evaluation framework and the Key Performance Indicators (KPIs) in our scope will be explained. The KPIs are based on the PCKS [SUM^{+}13] approach in order to ensure a meaningful link between the measured items and the underlying challenges and properties. This is followed by a description of the test environment, the queries used and finally the discussion as well as the explanation of the test cases employed.

4.1 Evaluation framework

A test framework encompasses the systematic accounting of criteria observed and methodologies applied in order to determine the impact of a change. In this case, the focus lies on observing changes in properties that allow for conclusions about the impact of a cache system on recall and response times of background queries from a streaming engine. The following sections will outline the respective indications under consideration and explain how these values are raised. For this purpose, we define the following performance indicators:

1. Response time for an external query from a streaming engine for varying sizes of returned information, which can be used as a baseline.
2. Recall for such queries from a streaming engine given response time restrictions (QoS).
3. Recall for such a time restricted query with the use of the unbound *inverse cache*, which will accept information after the engine's time-out.
4. Response time of background queries to the cache.
5. Overall time to completion for the streaming engine in regard to cache size.

4.1.1 Response time baseline

The response time baseline will be tested by requesting background information from the DBPedia SPARQL endpoint[1], as DBPedia offers an extensive amount of semantic real world data. These tests should return varying amounts of data in order to evaluate the responsiveness of this endpoint. In order to prevent slow response times due to complex joins and lookups, the DBPedia queries should only consist of basic lookups.

4.1.2 Recall with time-out

As streaming engines may operate under a QoS agreement, it might not be able to wait for the complete result from an external query. This measures the completeness of the result of a query after a certain amount of time has passed. The recall is defined as the amount of received data items out of the total available amount of matching data items for a certain query. A higher recall is favourable, as this will expose more relevant data to the streaming engine, thus yielding improved results.

4.1.3 Recall with time-out and inverse cache

The *inverse cache* will accept data after the streaming engine's time-out, which should result in a collection of data items in the cache. Operating under the assumption that an additional query to the cache will respond faster to a request for data, the cache should be able to offer additional data items to the streaming engine. This should result in a higher recall for the background query, which can improve the correctness of the streaming engine's composite result.

4.1.4 Response time from the cache

Analogical to part 4.1.1 the response time for different queries and varying amounts of returned data from the *inverse cache* is also taken into account. This should also allow for the specification of the lowest boundary for a time-out for given queries.

4.1.5 Overall time to completion

Lastly, one of the most basic indicators is the total runtime, which might be impacted by the faster responses from the cache. This indicator represents the amount of real time spent for processing all stream events and the associated background lookups.

4.2 Testing system

All tests were run on the so called *Kraken Cluster* at the *Department for Informatics* at the *University of Zurich*. This cluster consists of 12 distinct nodes that are managed through

---

[1] http://dbpedia.org/sparql

the *Terascale Open-Source Resource and QUEue Manager*[2] (Torque) that resides on another dedicated management server. All nodes are equipped with 24 physical processors and around 63 GB of RAM which makes the execution of parallel processes highly efficient and offers enough heap space for the execution of the test programs.

The used operating system on these nodes is Debian[3] and the java version is 1.7.0_25.[4] A detailed overview of the properties of a node can be found in the appendix on page 59.

4.3 Test cases

The following section describes the test cases that were used to gather data on the performance indicators mentioned beforehand. Each subsection outlines how the respective test was run and what data was gathered. Furthermore, any variation in the test setting will be pointed out (i.e. changing the cache size or varying the time-outs).

4.3.1 Test 1

A data set containing data from Twitter[5] (see section 3.2 on page 14 for details) was parsed and converted to *Esper* events. For each entry in this dataset a query to the SPARQL endpoint of DBPedia was initialized and the times for a complete result were noted. In order to receive a substantial amount of data without straining DBPedia unnecessarily, the query used was a *DESCRIBE* query. In order to receive a good baseline, this test should be run multiple times and the results should be condensed into an average as well as percentiles.

4.3.2 Test 2

This test also uses the *Twitter* data set, but introduces the *inverse cache* and a limitation on the timeout for each query. The timeout for the *inverse cache* and the DBPedia query are set to the same values and the same queries as are used in test 1. Furthermore, the amount of *Twitter* events that are parsed is limited to see if the results show a meaningful difference due to different test sizes. Both test parameters can be varied through start up arguments of the TwitterInverseCache.jar.

This test is supposed to yield results for the average recall for the possible queries from DBPedia and the *inverse cache* under time constraints. These values should also allow for the derivation of the combined recall. The used time-outs and the amount of events parsed can be seen in table A.1. All 100 permutations should be run several times in order to establish reliable values.

---

[2] http://www.adaptivecomputing.com/products/open-source/torque/

[3] Linux hector 3.2.0-4-amd64 #1 SMP Debian 3.2.51-1 x86_64 GNU/Linux

[4] OpenJDK Runtime Environment (IcedTea 2.3.10) (7u25-2.3.10-1 deb7u1) OpenJDK 64-Bit Server VM (build 23.7-b01, mixed mode)

[5] https://twitter.com/

4.3.3 Test 3

This test employs the EsperCSV classes and the storm events presented in chapter 2 and 3.3. The provided sensor events are parsed into storm events, which *Esper* handles. *Esper* then tries to match one of the queries from table 4.1 to the events in the current window. In the case of a successful match, *Esper* initiates a location lookup query for the first matching station. For the first result this is expected to be exclusively being returned by the *Fuseki* store, while the *Cassandra* store might also respond to subsequent queries. The used Esper EPL queries are based on the widely known SR-Bench queries, which are presented in natural language.[6]

This test further offers the possibility to alternate the maximum allowed cache size; in the experiments that were conducted the sizes 10, 100 and 1000 were used. Each query execution time is being logged through the SocketLogger in order to infer the impact on the total runtime of EsperCsv.

---

[6] http://www.w3.org/wiki/SRBench

Query name	EPL query
SRB1	SELECT DISTINCT sensorId, value FROM StormEvent.win:time_batch(60 minutes) WHERE (propertyName='Precipitation' and value>1)
SRB2	SELECT DISTINCT sensorId, sum(value) FROM StormEvent.win:time_batch(60 minutes) WHERE ((propertyName='Precipitation' OR propertyName='PrecipitationAccumulated' OR propertyName='PrecipitationSmoothed') AND value>1)
SRB3	SELECT sensorId FROM StormEvent.win:time(60 minutes) WHERE (propertyName='PeakWindSpeed' and value>73)
SRB4	SELECT sensorId, AVG(value) from StormEvent.win:time_batch(10 minutes) WHERE (propertyName='WindSpeed' and sensorId IN (SELECT sensorId from StormEvent.win:time_batch(10 minutes) where (propertyName='AirTemperature' and value>32)))
SRB5	SELECT DISTINCT sensorId from StormEvent.win:time(3 hours) WHERE (propertyName='Precipitation' AND value>10 AND sensorId IN (SELECT DISTINCT sensorId from StormEvent.win:time(3 hours) where (propertyName='AirTemperature' AND value<32 sensorId IN (SELECT DISTINCT sensorId from StormEvent.win:time(3 hours) where (propertyName='WindSpeed' AND value>40)))))
SRB6	SELECT DISTINCT sensorId FROM StormEvent.win:time(60 minutes) WHERE ((propertyName='Visibility' and value<10) OR (propertyName='Precipitation' AND value>30))
SRB8	SELECT sensorId, min(value) as minTemp, max(value) as maxTemp FROM StormEvent.win:time_batch(30 minutes) WHERE (sensorId=(SELECT sensorId from StormEvent.win:time_batch(30 minutes) TOP 1) AND propertyName='AirTemperature') GROUP BY sensorId
SRB TEST	SELECT * FROM StormEvent.win:time(1 min) where sensorId='AP035'

Table 4.1: Overview of the EPL queries for EsperCsv

Chapter 5

Experimental results

The following chapter provides an overview concerning the run experiments and their outcomes. For data too numerous to be displayed in full, excerpts are shown. The interpretation of this data will take place in chapter 6 on page 42ff.

5.1 First experiment

The first experiment was conducted with the TwitterInverseCache tool. The tables 5.1 and 5.2 on the following page offer an overview of the structure of the data file which was used to feed the *Esper* engine with events. In total this data set contains 308991 lines which are translated by the ReadCsvDataFiles class into 308990 *Esper* events. Each event contains a DBPedia resource, which will be queried using a *DESCRIBE* query through *Jena*.

Additionally, a test was run in order to determine if there is a major difference for the arrival timings of single data elements from DBPedia. This was measured through a *Jena* Iterator for each result. The findings for *DESCRIBE* and *SELECT* queries can be seen in table 5.3.

Resource (http://dbpedia.org/resource)	**Number of elements returned**	**Occurrences**
/Up_All_Night_Tour	73	1923
/Up_All_Night_(One_Direction_album)	288	447
/UAN	50	353
/One_Direction	350	177
/United_Kingdom	92202	145
/Tour_de_France	279	124
/Hug	104	120
/Australia	49584	77
/Justin_Bieber	603	74
/Indonesia	9102	69
/United_States	431232	57
/France	84594	56

Table 5.1: Number of returned triples for queries with more than 50 occurrences

Resource (http://dbpedia.org/resource)	**Number of elements returned**	**Occurrences**
/United_States	431232	57
/England	135266	6
/United_Kingdom	92202	145
/France	84594	56
/Poland	73135	2
Average	2900	7.25

Table 5.2: The five DBPedia resources that return the most triples

Query type	**Select**	**Describe**
Average time between results in ms	0.0397	0.0413
Percentile of results >=2ms	0.0011	0.0002
Percentile of results <1ms	0.9778	0.9762

Table 5.3: Overview of the response times and spread for *SELECT* and *DESCRIBE* queries for SPARQL queries that return more than 10000 elements

5.2 Second experiment

The second experiment uses the *Linked Sensor Observation* data sets (see table 3.1 on page 16 for details) as event inputs for the *Esper* engine. Each data set was run for 7 different EPL statements and for each combination of three different cache sizes (10;100;1000). The resulting 147 combinations were run two times in order to ensure that the displayed results are valid. The overall average recall for the whole run is shown - it should be noted, that due to the fact that the cache is empty to begin with, the average recall will never be able to reach 1. The recall was calculated as the total number of events that were matched by the cache divided by the number of elements that matched the EPL statement.

Unfortunately, it became obvious that even using the fast *Kraken* Cluster, some of the queries (SRB 5 and SRB 8) were too complex to allow for fast processing. These tests were aborted after each of them ran for more than 24 hours, as a such demanding and long-running test violates the premise of an economical test. These aborted tests are marked with *timeout*.

Another surprising point is that several data sets did not yield any results for certain queries. This is based on the composition of sensor observations for these tests: While SRB 1 and SRB 2 require precipitation information in order to match the EPL query to the generated data stream, these data sets do not include such information. Thus as result it was impossible to calculate recall. As for the query SRB 3 and the missing results for the data sets *Charley* and *Nevada Storm* - while these data sets include data concerning peak wind speeds, the boundary set by the query is so high, that it does not match a beneficial amount of events. For this reason, they were excluded from further discussion and are not represented in these tables.

5.3 Third experiment

In order to paint a more accurate picture of the impact the cache size has on the recall, this test uses the three simple queries (SRB 1-3) with finer granularity for the cache size, after the most interesting range was narrowed down previously. Additionally, recalculating the old values (10;100;1000) offers an opportunity to check the results of the previous tests.

The tables A.4, A.5, A.6 outlining these results can be found in the appendix. Additionally, the figures 5.1-5.8 on the next pages visualize the recall for each data set.

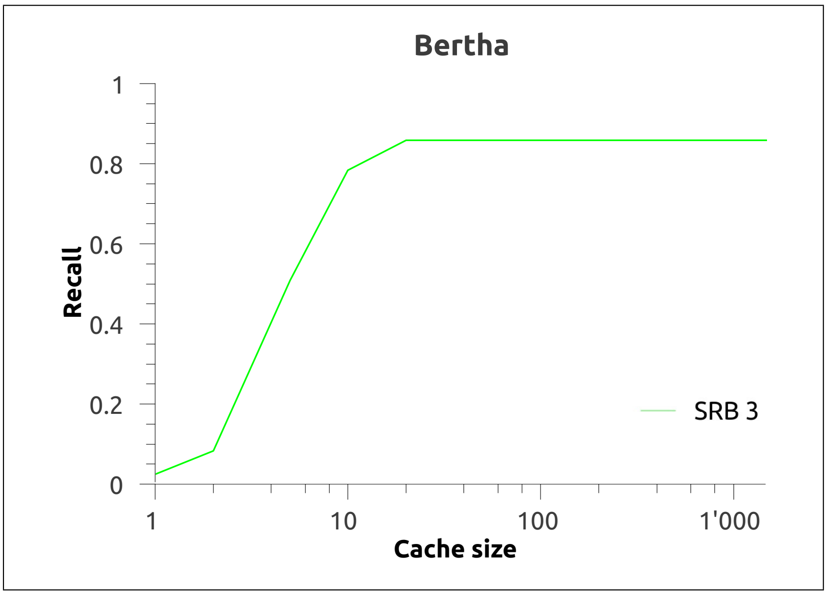

Figure 5.1: Recall for for various cache sizes for data set 'Bertha'

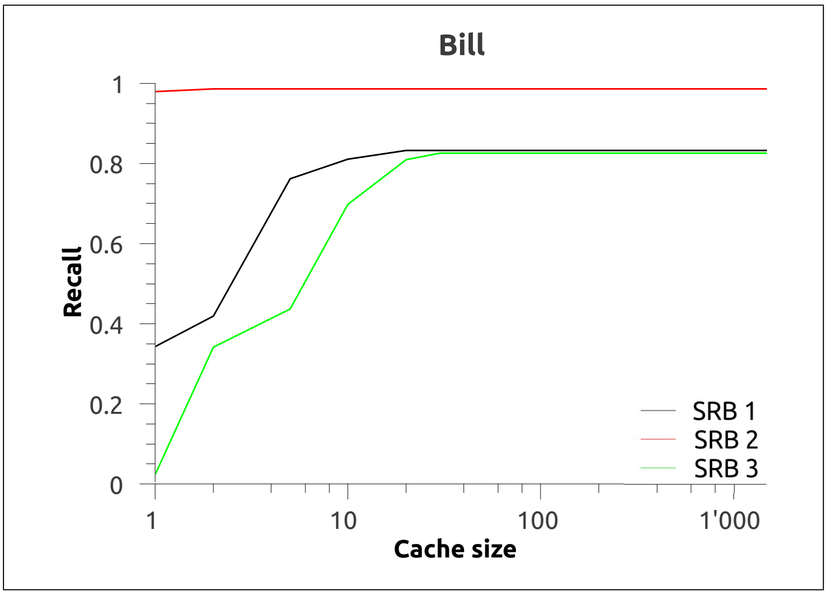

Figure 5.2: Recall for for various cache sizes for data set 'Bill'

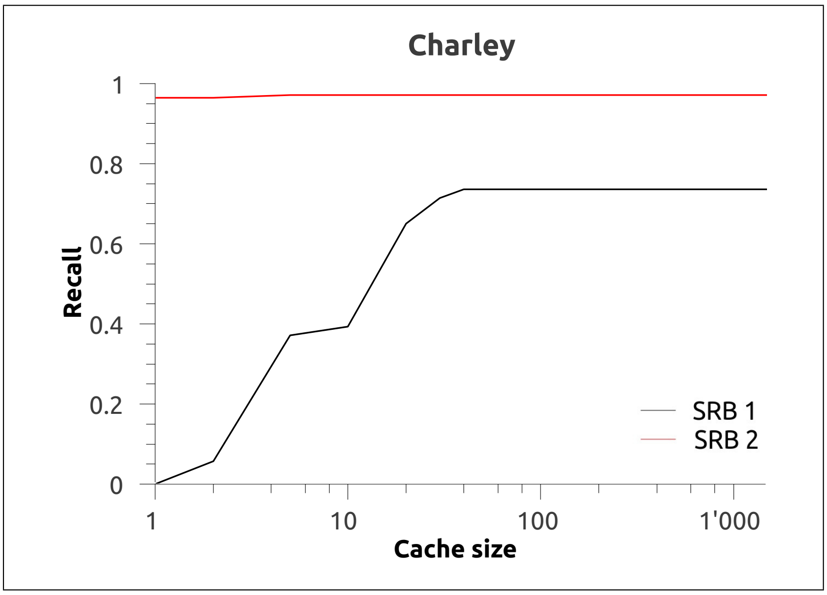

Figure 5.3: Recall for for various cache sizes for data set 'Charley'

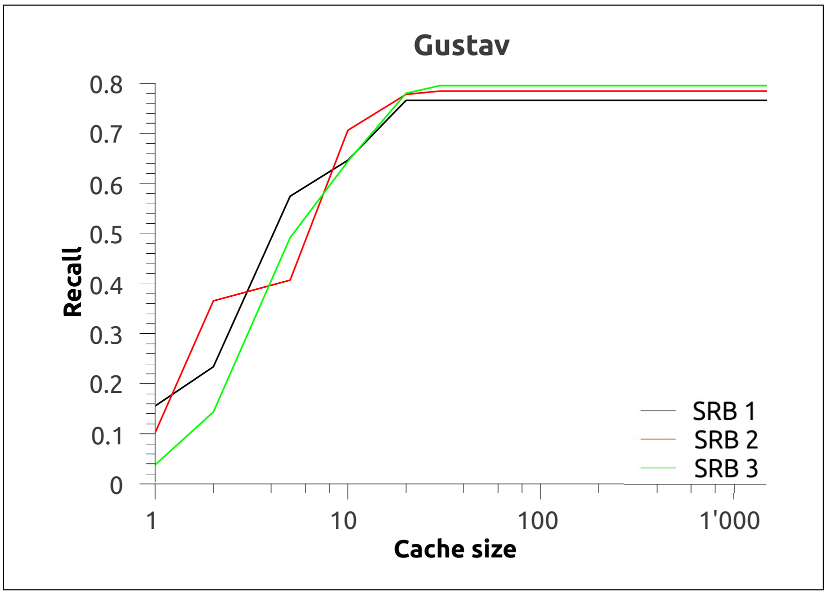

Figure 5.4: Recall for for various cache sizes for data set 'Gustav'

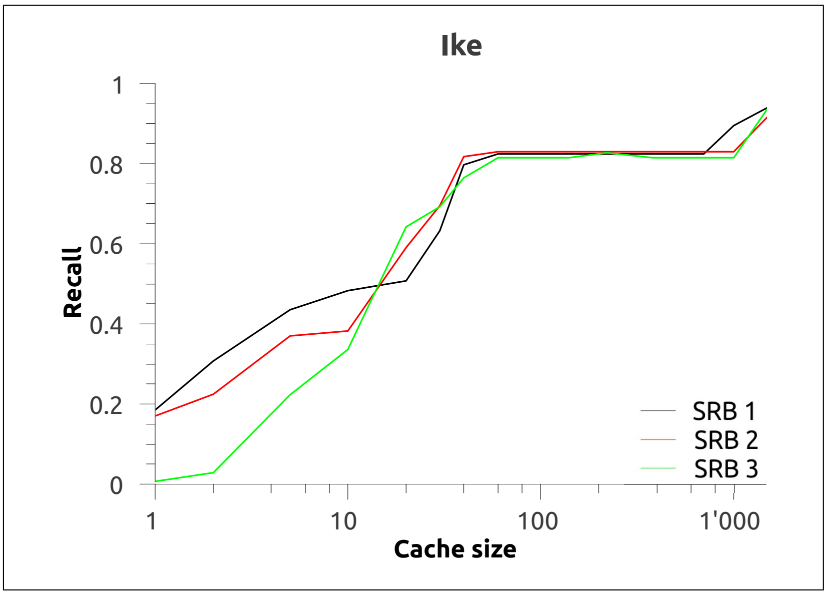

Figure 5.5: Recall for for various cache sizes for data set 'Ike'

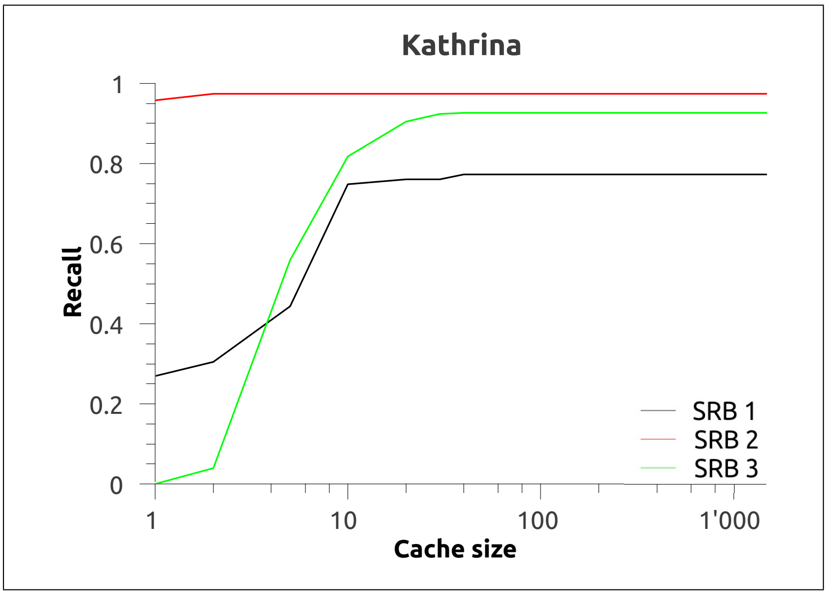

Figure 5.6: Recall for for various cache sizes for data set 'Kathrina'

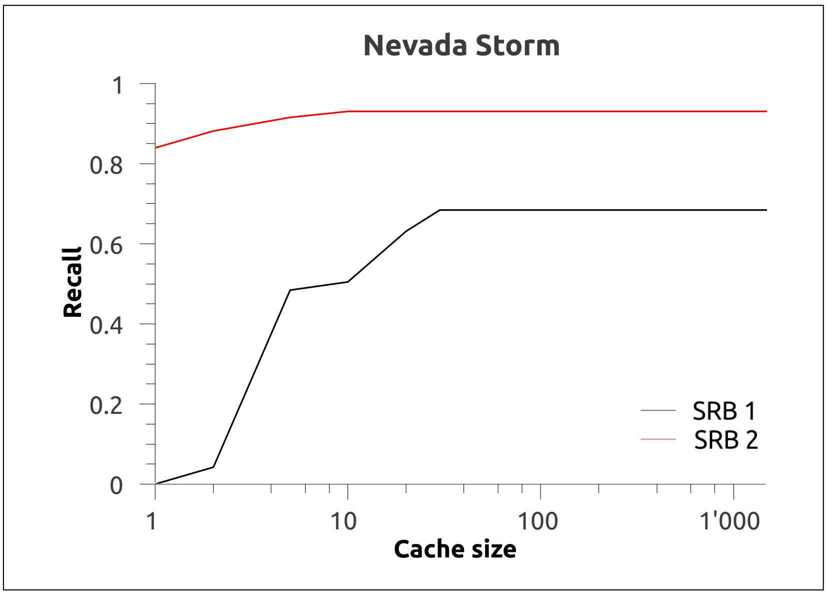

Figure 5.7: Recall for for various cache sizes for data set 'Nevada Storm'

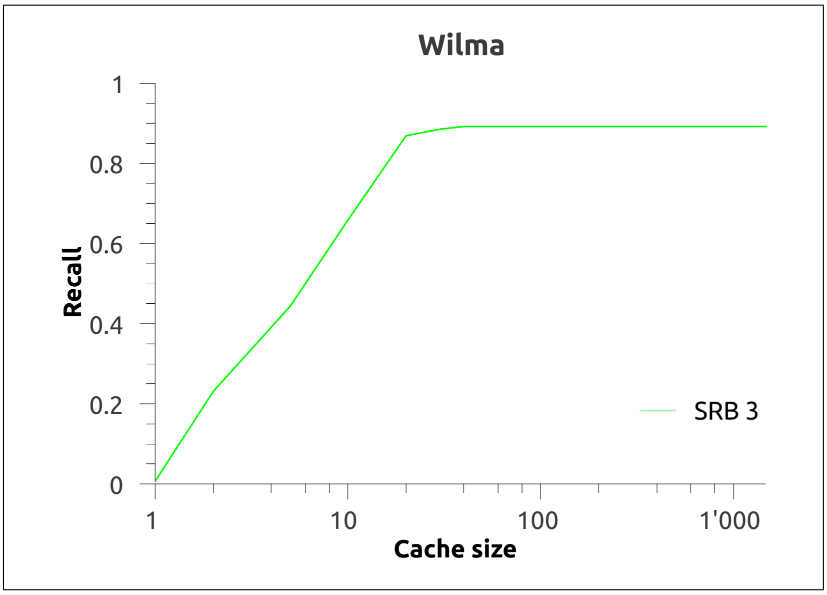

Figure 5.8: Recall for for various cache sizes for data set 'Wilma'

5.4 Fourth experiment

This experiment simulates the influence of the *inverse cache* on queries that are bounded through time-outs. This test used the *Twitter inverse cache* program to calculate the recall for different time-outs. The observation time-outs were chosen in order to cover both favourable and unfavourable circumstances for the recall of the *inverse cache*. As some quick tests determined that the route to the DBPedia servers is quite fast (~24 ms RTT), it was decided to use roughly 1.5x of the RTT as the lowest time-out value. From this starting point each subsequent measurement uses roughly double the time-out value of the previous run. The last time-out was chosen to be very high, so that these results can be used as a baseline. In order to ensure economical processing, a batch size of 1600 events, which will result in roughly 3200 total queries per run, was used.

The detailed results can be found in table 5.4. The respective time-out is listed - no results that violate this cut-off were taken into account - as well as the number of queries that could be answered (partly or in full). The next columns break this number down into the sources of these results. In this context, answer-parts are defined as sub-elements of an answer; this distinction is necessary as queries might return such a high number of results, that the time-out may cut off the transmission, leaving the *Esper* engine with parts of the answer from the background query.

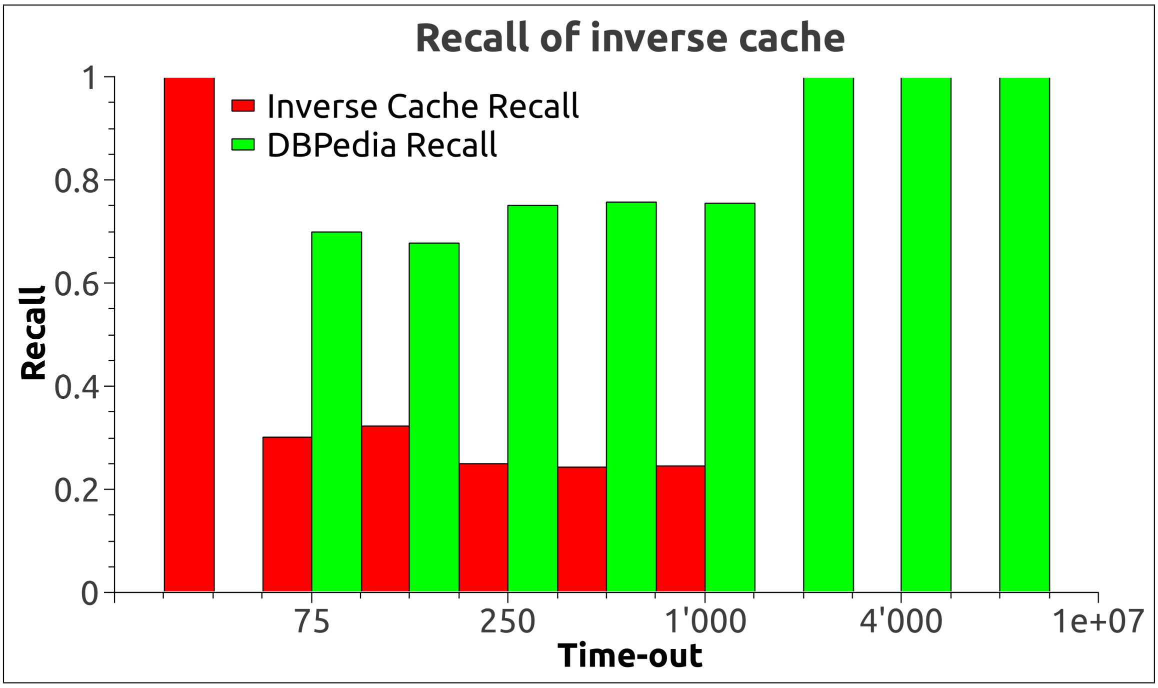

Figure 5.9: Recall based on the absolute number of results

Time-out in ms	**Answered queries before time-out**	**Answers from Inverse Cache**	**Answers from DBPedia**	**Sub-Answers from inverse cache**	**Sub-Answers from DBPedia**
37	26	26	0	1478	0
75	1448	435	1013	26971	114748
125	2025	655	1370	45619	336585
250	1919	479	1440	34197	577536
500	1960	478	1482	33938	999989
1000	1974	482	1492	47552	1212338
2000	1517	1	1516	15493	2384874
4000	1575	0	1575	0	7438619
8000	1573	0	1573	0	7303188
9999999	1590	0	1590	0	13907099

Table 5.4: Summary of the results for n=1600 and variable query time-outs

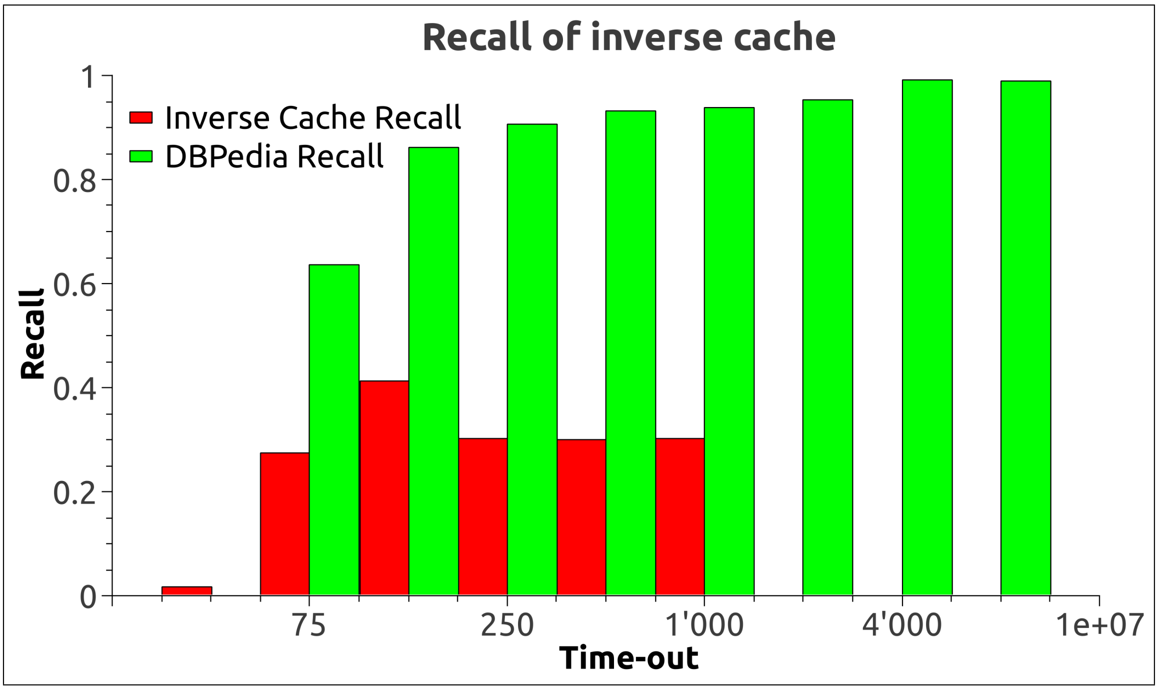

Figure 5.10: Recall based on the maximum number of possible results

Chapter 6

Discussion

This chapter will delve into the accumulated data and offer explanations for the observations where possible. Each experiment is going to be revisited in order to establish a link between the different aspects of the results and to help further the understanding of the *inverse cache*.

6.1 First experiment

The first experiment can be seen as groundwork as it underpins the decision to use *DESCRIBE* queries for parts of the experiments. The results show clearly that although there is a small difference (0.0397 ms vs. 0.0413 ms) between the more common *SELECT* query and the *DESCRIBE* query, both need approximately the same amount of time to return the next part of a result through an iterator. It should be noted though, that *SELECT* queries tend to be faster, as they need less computation to find the correct results.

Furthermore, the reliability of the DBPedia SPARQL endpoint is impressive, in so far as that more than 97% of the sub-results that arrived through the *DESCRIBE* iterator or the *SELECT* result set arrived with less than 1 ms gaps in between. The amount of sub-results that required equal or more than 2 ms until the next item was available is also minuscule (less than 0.1%). These numbers were extracted from queries which return more than 10’000 elements, so they should hold true for larger result sets.

6.2 Second experiment

The second experiment was conducted with EsperCsv in order to gather information on the recall performance of the *inverse cache* for various fixed cache sizes. The *Linked Sensor Observations* offer data on distinct storms in the US, yet they still belong to the same class of observations, which facilitates the building of a common processing path. As there is already a widely-known stream benchmark that uses the same base data set, it was decided to also implement some of the SR-Bench queries for this test. Unfortunately, it turned out that these queries were mainly constructed with pure streaming engines in mind: *Esper* is easily capable of processing several thousand events per second, but having to query background stores for additional data slows this processing down to a crawl, as such external queries take significantly longer.

Combining the SR-Bench queries with complete data sets leads to the undesirable result that some of the queries would need to run for more than 24 hours, which makes these queries unwieldy. Tests and benchmarks should be repeatable with limited resources so that they can be considered economical. Unfortunately, some of the more complex statements (SRB 5, SRB 8) fell into this category and were thus removed from further work. These queries might be interesting in regard to long term testing, though.

The remaining queries were used to determine the overall recall with various maximum cache sizes. A very obvious result from these tests is the fact that the recall seems to be mostly influenced by cache sizes of fewer than 100 results. High recall rates for cache sizes of 10 indicate that the EPL query results seem to exhibit a low amount of variability - examples can be found in the data for the SRB 2 query. Another factor that impacts the average recall is the total size of the data set used: A big data set might return more results and thus require a bigger cache to reach maximum efficiency.

A query to highlight is SRB 6, which exhibits a consistently even spread over all three caching sizes and across all data sets. Nevertheless, the influence of the raw event number is also evident when looking at the recall numbers for the biggest data set Ike. It shows that in order to reach the highest possible recall, even the biggest cache size can still be improved for this particular permutation.

Apart from these aspects, the *inverse cache* achieved high recall rates across the board, which is an indication that the amount of repetition, even in real world data, is not to be underestimated. Even a very small cache that only holds the results for 10 queries might be able to answer a surprising amount of background queries. Another possible improvement factor for this is the cache eviction strategy. During these examples a very basic "first in, first out" strategy was used to shrink the cache. A heuristic approach might be able to identify entries that are more likely to be requested again and thus increase the recall even further by keeping them in the cache whereas the FiFo algorithm would have removed them.

Again, it should also be noted, that the recall is defined as the percentage of queries that the cache was able to answer. It is impossible (barring errors) for the very first query for a certain element to be found in the cache, which means that the recall cannot rise to 1, it can only approach this border asymptotically.

6.3 Third experiment

The third experiment performed, is based on the results from the previous test, which rendered evident that the recall rate reached a high niveau very fast. In order to allow for a more detailed explanation in this respect, it was elected to generate additional data for supplemental cache sizes.

This test chose the fastest three queries (SRB 1-3) for this task and introduced 5 new sampling points between 10 and 100, as well as 3 below that number. Additionally, the range between 100 and 1000 was also subdivided and a last measuring point for a size of 1500 was introduced. The reasoning behind the introduction of the last sampling point is to check if the cache already topped-out at 1000 entries, or if there would still be room for some additional improvements. In total 16 cache sizes for each of the 8 distinct storm data sets were run for each of the three selected queries. This results in 384 variations.

Investigating whether there is a clear correlation between the size of the base data set (and thus the number of observations) and the optimal cache size did not yield clear results. While some data sets contain significantly more observations, this does not necessarily translate into demand for a bigger cache size. The cache sizes for the highest recall are visualized in the figures 6.1, 6.2, 6.3. The average optimal cache size, uniformly averaged over all data sets[1], yields an optimal cache size of 26.73, which can be used as a starting point for further optimisation.

In conclusion, this experiment has shown that an *inverse cache* with a modest size is able to reach promising recall results. It has to be noted though, that the optimal size of the *inverse cache* is highly dependant on environmental factors like the data set size (respectively the number of events per second for a streaming engine), the query itself and the make up of the data set.

---

[1] and excluding the outlier from SRB 3 / Ike

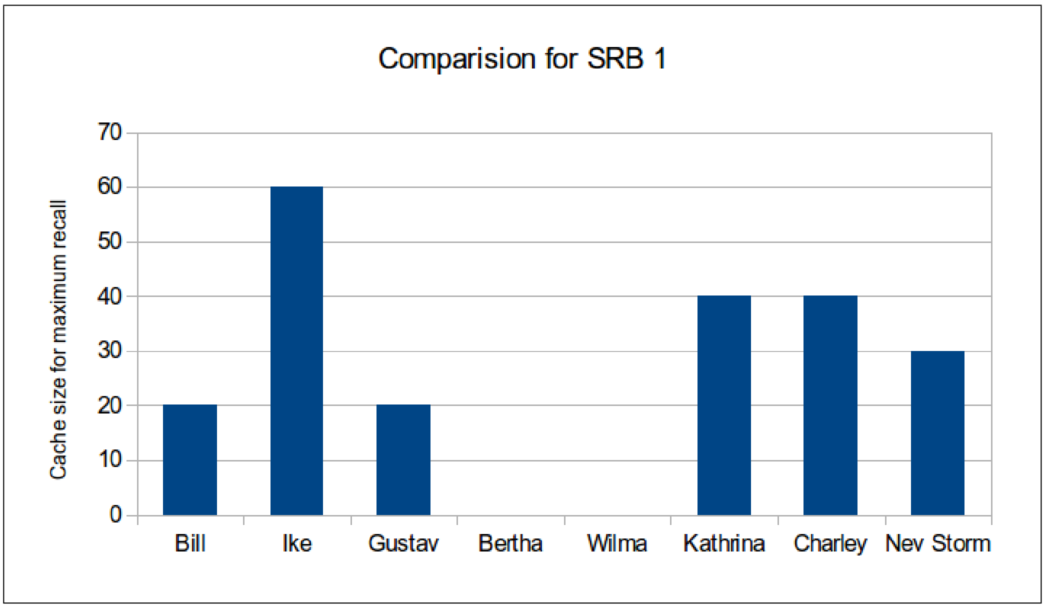

Figure 6.1: Minimum cache size from which on no further improvements of the overall recall were found for the query SRB 1

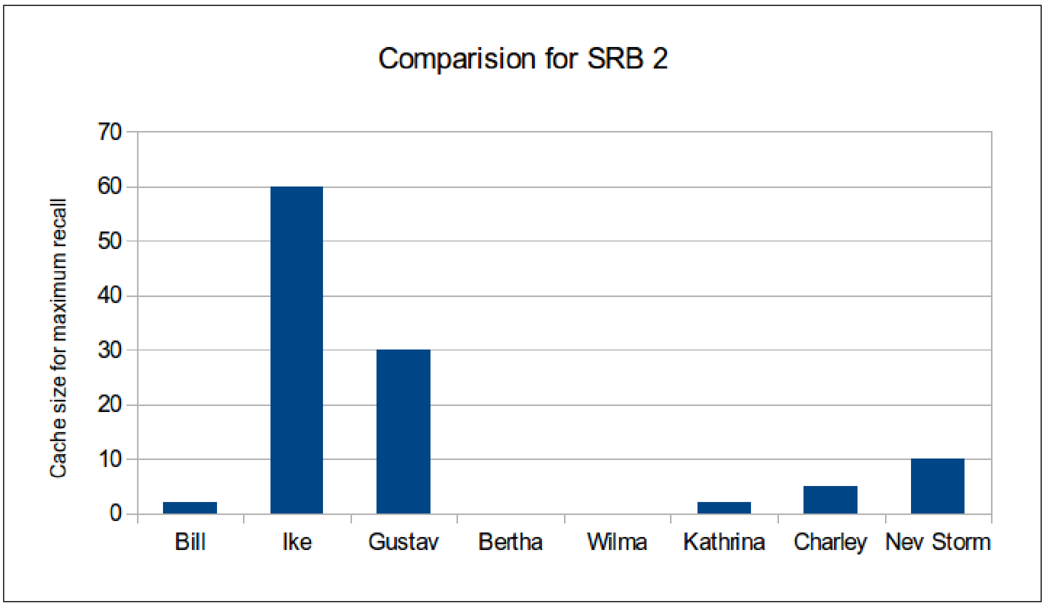

Figure 6.2: Minimum cache size from which on no further improvements of the overall recall were found for the query SRB 2

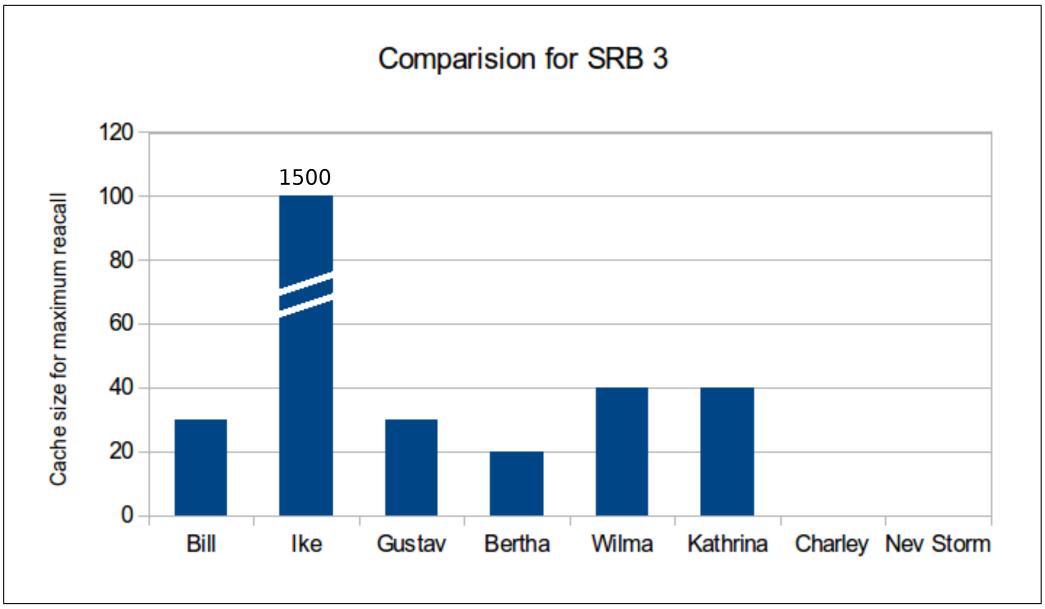

Figure 6.3: Minimum cache size from which on no further improvements of the overall recall were found for the query SRB 3

6.4 Fourth experiment

The aptly named fourth experiment investigates the relation between the recall of the *inverse cache*, the recall of an external background store and enforced time-outs through the use of the *Twitter inverse cache* Project. This test simulates the operation of the *Esper* engine under a time constricted QoS agreement for background data; *Esper* only accepts information from the background and *inverse cache* queries until a certain time-out is reached. Nevertheless, data that arrives after this cut-off is still forwarded to the *inverse cache*.

For this experiment, the range of time-out values that were considered, ranges from 37 to 8000 ms. An additional pass with a time-out of 999’999 ms was made to guarantee the existence of a baseline for the queries, so that the other results can be compared against the baseline. All received data was checked again in order to assure that only data consistent with the cut-off criterion was considered in the evaluation. One should also note, that the maximum number of results is smaller than the number of events, as there are several background queries that do not return any data. These empty results are not counted.

The results for this test, as seen in figure 5.10 and 6.1, as well as table 5.4 on page 41, show that the *inverse cache* improves the recall for time-outs in the range of 37-2000 ms. Beyond the 1000 ms time-out, the amount of recall for the *inverse cache* is steadily shrinking because there are less and less latecomers that need to be written to the cache.

In an environment with a forced time-out the *inverse cache* can fulfil two possible functions:

1. Increasing total recall: The *inverse cache* can allow for a higher number of total results, by augmenting the results from the background query with latecomers during the allowed time-out period
2. Speeding up recall: In an environment where only a part of the results is needed (i.e. a situation with a fixed recall or respectively a fixed needed result count) the *inverse cache* can alternatively speed up the response to a background query by adding its own results, allowing the response to reach the required amount of information faster and enabling the engine to proceed more rapidly.

The decision which function would be more advantageous, cannot be made in general though, as this would depend on outside factors.

The implementation of the *inverse cache* in this context has a downside too, which becomes clear when inspecting the recall values for the times-outs in the range 125-1000 ms: The combined recall of the DBPedia and the *inverse cache* queries is above 100%, which strongly suggests that there are duplicate results. These duplicates can be created easily due to the interdependency of the *inverse cache*, which will store every latecomer and the DBPedia queries which are not guaranteed to arrive in the same order or even in the same amount of time. This interdependency leads to a number of intersecting results. No further inquiries to determine the amount of duplicates were made.

Another area that can be improved is the handling of the latecomers, where a more intelligent approach might be beneficial. As of now, the *inverse cache* will receive all latecomers regardless of whether this datum already exists in the cache. As queries for checking the existence of data are fast in *Cassandra*, it might be beneficial to check for old data before overwriting it with a new *INSERT* query. This check was not implemented into the current version, as it is yet unclear if such a check would incur a significant performance penalty in comparison to making non-essential *INSERT* queries.

Additionally, the second experiment made apparent that distinct *INSERT* queries are a comfortable way of storing data in *Cassandra*, at least as long as there are not too many at the same time. The fourth experiment is storing several magnitudes more data in the *inverse cache* which leads to the *INSERT* queries turning into a bottle neck. In these cases it would be advisable to employ *Cassandra's sstables* mechanic which bulk inserts data as soon as a certain threshold is reached. Alternatively, it would also be possible to deploy several *Cassandra* instances and balance the *INSERT* queries between these machines.

It should also be noted, that the recall of the *inverse cache*, as displayed in figure 6.1 on page 45, first grows and then shrinks again. This behaviour might look counterintuitive, but can be explained by taking the background of the time-outs into account. The *inverse cache* is unable to provide results for very fast time-out values, so the recall grows with increasing time-outs. This continues up to the point when the amount of DBPedia results and responses from the *inverse cache* are in equilibrium. From there on, larger time-out values will lead to a diminishing recall for the *inverse cache*, as the overall amount of results that are considered latecomers, is shrinking.

Chapter 7

Related work

This thesis relates to the following domains of research:

1. Semantic Flow Processing (SFP)
2. Information Flow Processing (IFP)
3. Complex Event Processing (CEP)
4. RDF / SPARQL
5. Benchmarks for streaming processing

Information in the semantic web is commonly described using the Resource Description Framework (RDF) and increasingly often queried using SPARQL. The last years have borne witness to the creation of several stream processing systems. Systems like C-SPARQL [Bar09] are of major relevance for the area of semantic flow processing, which matches patterns (continuous queries) on segments of the data flow (commonly called windows). Another example is Event Processing SPARQL (EP-SPARQL) [AF11], which focusses more on the temporal relation between streamed data.

Closely related to these SFP systems are *Complex Event Processing* (CEP) systems like *Esper* [Dek07], that aggregate events with additional data in order to infer new information from these connections [SLG08]. Queries can be written in a plethora of dialects - *Esper* uses EPL, the *Esper* Processing Language.

There have been various proposed benchmarks for stream processing systems over the last years. Among the most widely known is the *SR-Bench(mark)* [ZDCC12] which uses real world data from the *Linked Open Data* cloud and describes several queries in natural language in order to be agnostic in regards to the actual implementation of these queries.

Another widely known benchmark is the the *Linear Road Benchmark* [ACG04], which simulates a toll system on a motorway with variable tolling based on accident detection, alerts and traffic congestion. It employs a fixed set of continuous and historical queries against a stream of events that is being generated in advance. A downside to the *Linear Road Benchmark* lies in its reliance on the relational data model, which makes it unfit for assessing the performance of RDF-graph based streaming engines. The Linear Road Benchmark is mainly used for IFP systems [JAA$^+$06].

Furthermore, there is the *Lehigh University Benchmark* (LUBM), which is a benchmark that is aiming at OWL Knowledge Base systems [GPH05]. This benchmark features synthetic data built over a university domain and offers a list of 14 SPARQL queries for which several metrics are computed.

In addition, there is also the *Berlin SPARQL* benchmark [BS09b] (BSBM) which uses an e-commerce setting wherein different products are offered and consumers can post reviews about these products. The queries are designed to emulate the search and navigation behaviour of a customer looking to buy a certain product. The BSBM was created in order to allow for performance comparison between different underlying systems that offer SPARQL endpoints.

In relation to the foundation of benchmarking, the work of [Hup09] is important, as it outlines basic properties and demands that need to be fulfilled by a good benchmark. Among them are demands for relevancy, repeatability, fairness as well as being economical and verifiable. These basic ideas are extended by the *Properties-Challenges-KPIs-Stress-tests* (PCKS) paradigm for benchmarking SFP systems [SUM$^+$13], which outlines seven commandments for stress testing SFPs in relation to precisely defined key performance indicators (KPI). This paradigm makes the valid point that not only throughput should be considered for performance evaluation [LpNQV13], but that response time, the maximum input throughput and the minimum time to completion or to accuracy should also be regarded as valid KPIs in most contexts.

Moreover, there is also a significant corpus investigating the impact of caches on databases [CMEF13] and RDF stores [MUA10, YW11, WW11] - but to the extent of the knowledge of the author, the proposed *inverse cache* for background queries of streaming engines is a novel idea in relation to streaming semantic data. The author was unable to find any other system that proposes caching of external information at the stream processing sink. It should be noted though, that there are some works that describe the use of a cache on a data stream source or for storing semantic data [DFJ$^+$96].

Chapter 8

Conclusion

This thesis implemented an *inverse cache* for use with *Complex Event Processing* engines that enrich events with additional data from outside sources. The *inverse cache* runs on the same machine as the CEP engine to ensure a fast connection in accordance with the spacial / temporal locality principle. In settings with *Quality of Service* requirements for the response time, the *inverse cache* will store late responses from the external data source and provide the event processing engine with these stored results for subsequent external queries.

The proposed system should fulfil two hypotheses: Firstly, allowing the stream processing engine to perform faster and secondly, providing more comprehensive results in environments with a QoS time-out constraint.

In order to check these hypotheses, the proposed system has undergone several tests which examined Key Performance Indicators (KPIs) that mirror these properties (minimum time to completion, recall in relation to response time). The tests were designed following the PCKS methodology [SUM$^+$13] and used SR-Bench data sets as well as queries.

After the evaluation through the programs *EsperCSV* and *TwitterInverseCache*, the results from chapter 5 clearly show that both of the early hypotheses can be validated for the test cases. In a scenario with a moderate amount of external queries the *inverse cache* was able to answer many of the background queries in 10% of the original query time. Furthermore, it could be shown that the *inverse cache* increased the overall recall for queries with a maximum response time between 37 and 1000 ms.

The significance of these results needs to be limited though, as the *inverse cache* has not been tested in different environments so far. Furthermore, the *inverse cache* employs a naive eviction strategy for tests with limited cache sizes and it has been shown that statistical heuristics might have the ability to improve this eviction approach and thus also yield higher recall results [NSB13].

Moreover, it should be investigated if *Apache Cassandra* is the best solution for the underlying storage of the *inverse cache*, or if another storage solution is superior in regards to the requirements of the *inverse cache*. Another point that could likely be improved, is the communication with *Cassandra* during *INSERT* queries. Currently the main programs are unable to insert many data items very rapidly as *Cassandra* only accepts a set number of concurrent connections. While one could certainly raise this limit, the author would rather propose to use the *Cassandra sstables* mechanic in order to allow for bulk inserts.

Future research might also surpass this work's implementation of the SR-Bench queries and create a comprehensive set of results for all available SR-Bench queries. This would improve the comparability of the *inverse cache*. Incorporating the *inverse cache* into other benchmarking systems for streaming engines might also result in a more complete view of the strengths and weaknesses of the local caching approach. Another intriguing topic that opens up through this thesis is the investigation of the impact that a distributed *inverse cache* would have in comparison to this singular implementation.

The author believes that while the current research focusses on improving RDF stores, the potential for caching external results locally has been overlooked so far. Hopefully, this thesis offers a first glimpse into the possibilities for performance improvement through systems such as the *inverse cache*.

Bibliography

[ABNS06] Asaf Adi, David Botzer, Gil Nechushtai, and Guy Sharon. Complex Event Processing for Financial Services. *2006 IEEE Services Computing Workshops*, pages 7–12, September 2006.

[ACG04] Arvind Arasu, Mitch Cherniack, and Eduardo Galvez. Linear road: a stream data management benchmark. In *VLDB '04 Proceedings of the Thirtieth international conference on Very large data bases*, pages 480–491, 2004.

[AF11] Darko Anicic and Paul Fodor. EP-SPARQL: a unified language for event processing and stream reasoning. *Proceedings of the 20th international conference on World wide web*, pages 635–644, 2011.

[Bar09] Davide Francesco Barbieri. C-SPARQL : SPARQL for Continuous Querying. *Proceedings of the 18th international conference on World wide web*, (c):1061–1062, 2009.

[BBC10] DF Barbieri, Daniele Braga, and S Ceri. Querying rdf streams with c-sparql. *ACM SIGMOD Record*, 39(1):20–26, 2010.

[BBD+02] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems - PODS '02*, page 1, 2002.

[Bre00] Eric A. Brewer. Towards robust distributed systems. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*, page 7, 2000.

[BS09a] Christian Bizer and Andreas Schultz. Benchmarking the performance of storage systems that expose SPARQL endpoints. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[BS09b] Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *International Journal on Semantic Web and Information Systems (IJSWIS)*, 5(2):1–24, 2009.

[BSW01] Shivnath Babu, L Subramanian, and Jennifer Widom. A data stream management system for network traffic management. *Workshop on Network-Related Data Management (NRDM 2001)*, pages 1–2, 2001.

[CCeaD13] Aranda Carlos, Olivier Corby, and Souripriya et al Das. SPARQL 1.1 Overview, 2013.

[CCG10] Jean-paul Calbimonte, Oscar Corcho, and Alasdair J G Gray. Enabling Ontology-based Access to Streaming Data Sources. In *9th International Semantic Web Conference (ISWC 2010)*, pages 96–111, 2010.

[CM12] G Cugola and Alessandro Margara. Processing flows of information: From data stream to complex event processing. *ACM Computing Surveys (CSUR)*, pages 359–360, 2012.

[CMEF13] P Cudré-Mauroux, Iliya Enchev, and Sever Fundatureanu. NoSQL Databases for RDF: An Empirical Evaluation. *The Semantic Web – ISWC 2013*, 8219:310–325, 2013.

[Dek07] Paul Dekkers. *Complex Event Processing*. PhD thesis, 2007.

[DFJ$^+$96] Shaul Dar, MJ Franklin, BT Jonsson, D Srivastava, and Michael Tan. Semantic data caching and replacement. *VLDB*, 96:330–341, 1996.

[DKSU11] Songyun Duan, Anastasios Kementsietsidis, Kavitha Srinivas, and Octavian Udrea. Apples and Oranges : A Comparison of RDF Benchmarks and Real RDF Datasets. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 145–156, 2011.

[DS12] Miyuru Dayarathna and T Suzumura. XGDBench: A benchmarking platform for graph stores in exascale clouds. *Cloud Computing Technology and Science (CloudCom)*, pages 3–10, 2012.

[Esp14] Esper Team. Esper Reference, 2014.

[FSB13] Lorenz Fischer, Thomas Scharrenbach, and Abraham Bernstein. Scalable Linked Data Stream Processing via Network-Aware Workload Scheduling. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems*, number 296126, page 81, 2013.

[GL02] Seth Gilbert and Nancy Lynch. Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51, June 2002.

[GPH05] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2):158–182, 2005.

[HCH$^+$99] E.N. Hanson, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J.B. Park, and a. Vernon. Scalable trigger processing. *Proceedings 15th International Conference on Data Engineering (Cat. No.99CB36337)*, pages 266–275, 1999.

[Hup09] Karl Huppler. The art of building a good benchmark. *Performance Evaluation and Benchmarking*, 2009.

[JAA$^+$06] Navendu Jain, Lisa Amini, Henrique Andrade, Richard King, Yoonho Park, Philippe Selo, and Chitra Venkatramani. Design , Implementation , and Evaluation of the Linear Road Benchmark on the Stream Processing Core. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 431–442, 2006.

[KCF12] Srdjan Komazec, Davide Cerri, and Dieter Fensel. Sparkwave: continuous schema-enhanced pattern matching over RDF data streams. *DEBS '12 Proceedings of the 6th ACM International Conference on Distributed Event-Based Systems*, pages 58–68, 2012.

[LF98] DC Luckham and Brian Frasca. Complex event processing in distributed systems. *Computer Systems Laboratory Technical Report CSL-TR-98-754*, 1998.

[LpDtPH11] Danh Le-phuoc, Minh Dao-tran, Josiane Xavier Parreira, and Manfred Hauswirth. A Native and Adaptive Approach for Unified Processing of Linked Streams and Linked Data. In *The Semantic Web – ISWC 2011: 10th International Semantic Web Conference*, volume 7031, pages 370–388, 2011.

[LpNQV13] Danh Le-phuoc, Hoan Nguyen, Mau Quoc, and Chan Le Van. Elastic and Scalable Processing of Linked Stream Data in the Cloud. *The Semantic Web – ISWC 2013*, 287305:280–297, 2013.

[Luc02] David C Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*, volume 2003. Reading: Addison-Wesley, 2002.

[MUA10] Michael Martin, J Unbehauen, and S Auer. Improving the performance of semantic web applications with SPARQL query caching. *The Semantic Web: Research and Applications Lecture Notes in Computer Science*, 6089:304–318, 2010.

[NSB13] MK Nguyen, Thomas Scharrenbach, and Abraham Bernstein. Eviction Strategies for Semantic Flow Processing. In *9th International Workshop on Scalable Semantic Web Knowledge Base Systems*, number 296126, page 66, 2013.

[PHS10] Harshal Patni, Cory Henson, and Amit Sheth. Linked sensor data. *2010 International Symposium on Collaborative Technologies and Systems*, pages 362–370, 2010.

[RSMM12] Tilmann Rabl, G Sergio, Victor Munt, and Serge Mankovskii. Solving Big Data Challenges for Enterprise Application Performance Management. *Proceedings of the VLDB Endowmen*, 5(12):1724–1735, 2012.

[SLG08] S Stoa, Morten Lindeberg, and Vera Goebel. Online analysis of myocardial ischemia from medical sensor data streams with Esper. *Applied Sciences on Biomedical and Communication Technologies*, pages 1–5, 2008.

[SUM$^+$13] Thomas Scharrenbach, Jacopo Urbani, Alessandro Margara, Emanuele Della Valle, and Abraham Bernstein. Seven Commandments for Benchmarking Semantic Flow Processing Systems. In *Proceedings of the 10th Extended Semantic Web Conference (ESWC)*, pages 305–319, 2013.

[TB11] Bogdan George Tudorica and Cristian Bucur. A comparison between several NoSQL databases with comments and notes. *2011 RoEduNet International Conference 10th Edition: Networking in Education and Research*, pages 1–5, June 2011.

[TGB05] VV Titov, FI Gonzalez, and EN Bernard. Real-time tsunami forecasting: Challenges and solutions. *Developing Tsunami-Resilient Communities*, 2005.

[VMS12] Martin Voigt, Annett Mitschick, and Jonas Schulz. Yet Another Triple Store Benchmark? Practical Experiences with Real-World Data. *SDA*, 2012.

[WLLB06] Fusheng Wang, S Liu, Peiya Liu, and Yijian Bai. Bridging physical and virtual worlds: complex event processing for RFID data streams. *Advances in Database Technology-EDBT 2006*, 2006.

[WW11] GT Williams and Jesse Weaver. Enabling fine-grained HTTP caching of SPARQL query results. *The Semantic Web–ISWC 2011*, pages 762–777, 2011.

[YW11] Mengdong Yang and Gang Wu. Caching intermediate result of SPARQL queries. *Proceedings of the 20th international conference companion on World wide web - WWW '11*, page 159, 2011.

[ZDCC12] Ying Zhang, PM Duc, Oscar Corcho, and JP Calbimonte. SRBench: a streaming RDF/SPARQL benchmark. In *The Semantic Web – ISWC 2012*, pages 641–657. 2012.

Appendix

A.1 Test cases

Time-out in ms	Number of events
37	10
75	100
125	200
250	400
500	800
1000	1600
2000	3200
4000	6400
8000	12800
999999	308991

Table A.1: Overview of permutations used for Test 2

Query Name	Data set	Cache size
SRB1	Bertha	10
SRB2	Bill	100
SRB3	Charley	1000
SRB4	Gustav	
SRB5	Ike	
SRB6	Kathrina	
SRB8	Nevada Storm	
SRB TEST	Wilma	

Table A.2: Overview of permutations used for Test 3

A.2 Overview of node properties

Each of the 12 *Kraken* nodes offers this hardware:

```
processor : 8
vendor_id : AuthenticAMD
cpu family : 16
model   : 9
model name : AMD Opteron(tm) Processor 6174
stepping : 1
microcode : 0x10000c4
cpu MHz  : 800.000
cache size : 512 KB
physical id : 0
siblings : 12
core id  : 2
cpu cores : 12
apicid  : 24
initial apicid : 8
fpu  : yes
fpu_exception : yes
cpuid level : 5
wp  : yes
flags  : fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov
   pat pse36 clflush mmx fxsr sse sse2 ht syscall nx mmxext fxsr_opt
   pdpe1gb rdtscp lm 3dnowext 3dnow constant_tsc rep_good nopl
   nonstop_tsc extd_apicid amd_dcm pni monitor cx16 popcnt lahf_lm
   cmp_legacy svm extapic cr8_legacy abm sse4a misalignsse 3
   dnowprefetch osvw ibs skinit wdt nodeid_msr hw_pstate npt lbrv
   svm_lock nrip_save pausefilter
bogomips : 4400.35
TLB size : 1024 4K pages
clflush size : 64
cache_alignment : 64
address sizes : 48 bits physical, 48 bits virtual
power management: ts ttp tm stc 100mhzsteps hwpstate
```

Listing A.1: Overview of the CPU features of a Kraken node

```
MemTotal: 66112060 kB
MemFree: 37049716 kB
Buffers: 334808 kB
Cached: 27904112 kB
SwapCached: 2664 kB
Active: 13243324 kB
Inactive: 15101836 kB
```

```
Active(anon): 77580 kB
Inactive(anon): 51212 kB
Active(file): 13165744 kB
Inactive(file): 15050624 kB
Unevictable: 9948 kB
Mlocked: 9948 kB
SwapTotal: 127406076 kB
SwapFree: 127394000 kB
Dirty: 4 kB
Writeback: 0 kB
AnonPages: 115336 kB
Mapped: 11780 kB
Shmem: 19200 kB
Slab: 509960 kB
SReclaimable: 475360 kB
SUnreclaim: 34600 kB
KernelStack: 2240 kB
PageTables: 2720 kB
NFS_Unstable: 0 kB
Bounce: 0 kB
WritebackTmp: 0 kB
CommitLimit: 160462104 kB
Committed_AS: 367468 kB
VmallocTotal: 34359738367 kB
VmallocUsed: 416764 kB
VmallocChunk: 34308991840 kB
HardwareCorrupted: 0 kB
AnonHugePages: 0 kB
HugePages_Total: 0
HugePages_Free: 0
HugePages_Rsvd: 0
HugePages_Surp: 0
Hugepagesize: 2048 kB
DirectMap4k: 248384 kB
DirectMap2M: 37498880 kB
DirectMap1G: 29360128 kB
```

Listing A.2: Overview of the memory configuration of a Kraken node

A.3 Experiments

Query	Cache	Bill	Bertha	Charley	Gustav	Ike	Kathrina	Nevada Storm
SRB 1	10	0.8112	-	0.3714	0.6467	0.5075	0.7485	0.5053
	100	0.8322	-	0.7357	0.7665	0.8239	0.7725	0.6842
	1000	0.8322	-	0.7357	0.7665	0.8239	0.7725	0.6842
SRB 2	10	0.9860	-	0.9720	0.7066	0.3821	0.9738	0.9301
	100	0.9860	-	0.9720	0.7844	0.8299	0.9738	0.9301
	1000	0.9860	-	0.9720	0.7844	0.8299	0.9738	0.9301
SRB 3	10	0.6984	0.7833	-	0.6439	0.3354	0.8178	-
	100	0.8254	0.8583	-	0.7955	0.8150	0.9262	-
	1000	0.8254	0.8583	-	0.7955	0.8150	0.9262	-
SRB 4	10	0.2204	0.3097	0.9988	0.4155	0.1743	0.0774	0.6434
	100	0.7297	0.7993	0.9988	0.6759	0.4429	0.9774	0.9605
	1000	0.9362	0.8561	0.9988	0.7793	0.8267	0.9774	0.9605
SRB 5	10	timeout	timeout	timeout	timeout	timeout	timeout	timeout
	100	timeout	timeout	timeout	timeout	timeout	timeout	timeout
	1000	timeout	timeout	timeout	timeout	timeout	timeout	timeout
SRB 6	10	0.0156	0.2009	0.0571	0.0145	0.0000	0.0124	0.0000
	100	0.1476	0.9134	0.4306	0.1112	0.1037	0.3333	0.2931
	1000	0.6246	0.9938	0.8996	0.6691	0.5846	0.8880	0.9013
SRB 8	10	timeout	timeout	timeout	timeout	timeout	timeout	timeout
	100	timeout	timeout	timeout	timeout	timeout	timeout	timeout
	1000	timeout	timeout	timeout	timeout	timeout	timeout	timeout

Table A.3: Overview of the average overall recall for a combination of data set and query with limited cache sizes

Cache Size	Bertha	Bill	Charley	Gustav	Ike	Kathrina	Nevada Storm	Wilma
1	-	0.3427	0.0000	0.1557	0.1851	0.2695	0.0000	-
2	-	0.4196	0.0571	0.2335	0.3075	0.3054	0.0421	-
5	-	0.7622	0.3929	0.5749	0.4358	0.4431	0.4842	-
10	-	0.8112	0.3714	0.6467	0.5075	0.7485	0.5053	-
20	-	0.8322	0.6500	0.7665	0.6328	0.7605	0.6316	-
30	-	0.8322	0.7143	0.7665	0.5075	0.7605	0.6842	-
40	-	0.8322	0.7357	0.7665	0.7970	0.7725	0.6842	-
60	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
80	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
100	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
140	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
220	-	0.8322	0.7357	0.7665	0.8955	0.7725	0.6842	-
380	-	0.8322	0.7357	0.7665	0.9403	0.7725	0.6842	-
700	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
1000	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-
1500	-	0.8322	0.7357	0.7665	0.8239	0.7725	0.6842	-

Table A.4: Additional results for varying cache sizes - displayed is the overall recall for query SRB1

Cache Size	Bertha	Bill	Charley	Gustav	Ike	Kathrina	Nevada Storm	Wilma
1	-	0.9790	0.9650	0.1018	0.1701	0.9581	0.8392	-
2	-	0.9860	0.9650	0.3653	0.2239	0.9738	0.8811	-
5	-	0.9860	0.9720	0.4072	0.3701	0.9738	0.9161	-
10	-	0.9860	0.9720	0.7066	0.3821	0.9738	0.9301	-
20	-	0.9860	0.9720	0.7784	0.5910	0.9738	0.9301	-
30	-	0.9860	0.9720	0.7844	0.6955	0.9738	0.9301	-
40	-	0.9860	0.9720	0.7844	0.8179	0.9738	0.9301	-
60	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
80	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
100	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
140	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
220	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
380	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
700	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
1000	-	0.9860	0.9720	0.7844	0.8299	0.9738	0.9301	-
1500	-	0.9860	0.9720	0.7844	0.9164	0.9738	0.9301	-

Table A.5: Additional results for varying cache sizes - displayed is the overall recall for query SRB2

Cache Size	Bertha	Bill	Charley	Gustav	Ike	Kathrina	Nevada Storm	Wilma
1	0.0250	0.0238	-	0.0379	0.0063	0.0000	-	0.0065
2	0.0833	0.3413	-	0.1439	0.0282	0.0390	-	0.2313
5	0.5083	0.4365	-	0.4924	0.2226	0.5597	-	0.4430
10	0.7833	0.6984	-	0.6439	0.3354	0.8178	-	0.6580
20	0.8583	0.8095	-	0.7803	0.6426	0.9046	-	0.8697
30	0.8583	0.8254	-	0.7955	0.6928	0.9241	-	0.8860
40	0.8583	0.8254	-	0.7955	0.7649	0.9262	-	0.8925
60	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
80	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
100	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
140	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
220	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
380	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
700	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
1000	0.8583	0.8254	-	0.7955	0.8150	0.9262	-	0.8925
1500	0.8583	0.8254	-	0.7955	0.9373	0.9262	-	0.8925

Table A.6: Additional results for varying cache sizes - displayed is the overall recall for query SRB3