University of Zurich[UZH]

Extending Rdfbox with Distributed RDF Management. Efficient RDF Indexing and Loading.

Master Thesis April 1, 2013

Daniel Spicar
of Zürich, Switzerland

Student-ID: 06-702-542
daniel.spicar@uzh.ch

Advisor: **Cosmin Basca**

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

Acknowledgements

First of all I would like to express my gratitude to Professor Abraham Bernstein, Ph.D., for giving me the opportunity to pursue my master thesis at the Dynamic and Distributed Information Systems Group (DDIS).

I would also like to thank my supervisor, Cosmin Basca, for helping me throughout this thesis with his valuable advice.

Last but not least, I am grateful for the support of my family and especially to Věra, for accompanying me during this challenging time.

Zusammenfassung

Das Semantische Web (engl. *Semantic Web*) wächst ständig. RDF Datensätze erreichen grössen von über einer Milliarde Tripel. Das Verwalten und Bereitstellen dieser Datensätze kann einzelne Computer an ihre Leistungsgrenzen bringen. Diese Arbeit erweitert Rdfbox, ein auf Hexastore basierendes semantisches Datenbankmanagementsystem, um die Unterstützung von verteilten Indexspeichern. Zusätzlich wurde eine parallele Anfrageauswertung und ein verteiltes System zum Einlesen von RDF-Graphen implementiert. Auswertungen zeigen, dass verteiltes Laden den vorherigen Lösungen weit überlegen ist und dass die parallele Anfrageauswertung für eine hohe Performanz der verteilten Indexstruckturen notwendig ist. Aber es wird ebenso deutlich, dass verteilte Indizes in Rdfbox nicht besonders effizient funktionieren können und es grundlegende Anpassungen in der Anfrageauswertung braucht, damit die Performanz deutlich erhöht werden kann.

Abstract

The Semantic Web is growing at a fast pace and single machines can reach the limits of their capabilities when trying to manage RDF data sets that contain billions of triples. In order to tackle this problem, this thesis extends Rdfbox, a Hexastore-based semantic database management system, with support for distributed index backends, a parallelised query execution engine and a distributed index loader. Evaluations show that the distributed index loader outperforms previous approaches and that the improved query execution engine is performance-critical with distributed and indices. But limits to the efficiency of distributed indices in Rdfbox are discovered that may require fundamental changes in the approach to query resolution in order to achieve a significant performance increase.

Table of Contents

1

Introduction

The Semantic Web keeps growing because an increasing amount of data is made accessible and interlinked every day. The driving force behind this effort is a vision of a World Wide Web (WWW) for data. The original WWW links documents together but the web of data interlinks all imaginable types of data. The WWW was meant to be used by humans, but linked data can be processed by machines. Because the links between data express semantics [10], machines can interpret it. Like in the WWW, this growth is not controlled by a central entity. The World Wide Web Consortium (W3C) proposes open standards to keep data accessible in a well-defined manner, such that it can be shared across application, enterprise or community boundaries and evolve into a global data space [27].

The Resource Description Framework (RDF) is the W3C's data format for the Semantic Web [30]. It specifies a graph data structure that can be decomposed into a set of statements about resources. These resources can be anything, as long as it can be referenced by a unique identifier. Because each statement consists of three elements, they are called *triples*.

Perhaps the most simple form of storing RDF, is serialised in a static file. But static files do not scale to large amounts of data because they can not be efficiently modified and queried by applications [27]. The reason for this is the scanning of large files required when accessing the data. Therefore RDF is typically stored using modern database management systems (DBMS). A DBMS optimised for storing RDF data is called a *triple store* [40]. Maintaining an index is a common approach to improve access to data in triple stores. Indices are data structures that can be used to traverse search spaces efficiently. This work focuses on a specific RDF data indexing scheme, called *Hexastore* [45]. *Hexastore* proposes a multi-index scheme for efficient support of general purpose queries at the cost of increased storage space requirements. Other approaches often prefer specific – perhaps more common – types of queries. Such approaches are suitable for specific use cases, but offer no general solution.

Rdfbox is an extension of the Hexastore indexing scheme. Prior to this work it has been limited to locally available B+-Tree index structures. This thesis extends Rdfbox with support for two selected distributed indices based on Hyperdex[20] and Apache Accumulo [22] and creates a distributed loading mechanism based on MapReduce [17].

Outline The structure of this thesis is as follows. Chapter 2 introduces the reader to the central concepts and core technologies of the Semantic Web and provides an introduction to different RDF storage solutions. Further it explains the Hexastore scheme and its implementation in Rdfbox, two technologies this work directly depends on. Then the two key-value stores used to provide distributed indices for Rdfbox are introduced. Chapter 3 presents the motivation for this work and describes the necessary implementation tasks. In chapter 4 the design and implementation of the distributed index extensions for Rdfbox are described in detail, thereafter the solution is evaluated in chapter 5. Chapter 6 compares the presented work to related distributed triple stores. Chapter 7 concludes this thesis with the discussion of results and future work.

2

Background

This chapter introduces the central Semantic Web technologies and discusses Semantic Web data storage approaches. Then follows a detailed explanation of a specific storage scheme called *Hexastore* and its implementation in Rdfbox. Finally the design of the Hyperdex and Apache Accumulo distributed key-value stores is introduced.

2.1 Semantic Web Basics

This section provides a brief introduction to the most important aspects of the Resource Description Framework and the SPARQL query language.

2.1.1 Resource Description Framework (RDF)

RDF is a core element of the World Wide Web Consortium's (W3C) Semantic Web stack. It is a language and data structure for representing information about resources on the World Wide Web (WWW) [30, 34] and able to represent information about anything that can be identified on the web. Uniform Resource Identifiers (URI) are used for identification. They are a generalisation of the well known URLs (e.g. *http://www.uzh.ch*). URLs are only used to identify network locations but URIs can identify people, concepts, ideas and other things that are not network retrievable [9].

In RDF resources are described by statements and each statement consist of three elements: *subject*, *predicate* and *object*. Because each statement forms a 3-tuple $(subject, predicate, object)$, they are called *triples*. Triples are ordered. A statement (a, b, c) with subject a, predicate b and object c is different from a statement (c, b, a). In the latter case c is the subject and a is the object. A collection of statements forms an RDF data set or *graph*. The graph interpretation arises when subject and object are viewed as nodes and the predicate as a typed and directed edge from subject to object. The subject is always a node. Nodes are either identified by an URI or they are *blank nodes*. Blank nodes are a special node type used primarily for structuring statements (into ordered lists for example). In their abstract interpretation they have no inherent identification. They can only be identified through their context. For practical purposes, applications do assign special identifiers to them however. Predicates are identified by their type's URI. Objects can be either nodes or literals. Literals represent values and

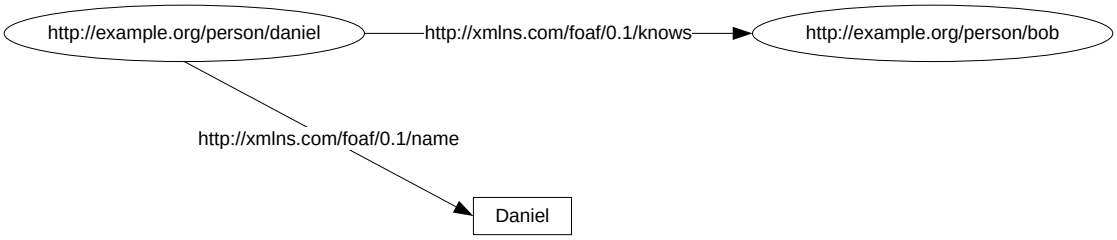

Figure 2.1: A small RDF graph consisting of two statements. One statement is "the resource daniel knows the resource bob". This is a statement connecting two nodes. The other statement is "the resource daniel has a name with value 'Daniel' ". This is a statement connecting a node to a literal.

can be plain text strings with an optional language identifier or they can be typed literals with an XML schema datatype URI. Figure 2.1 depicts a simple RDF graph.

The semantics in RDF are primarily expressed through the type of the predicates. Predicates are properties of a subject that either point to *values* or *relate* the subjects to other nodes. Examples of *value* properties are predicates which specify the name or the favourite colour. On the other hand a *relational* property specifies that a person knows another person. In these examples, rules and semantics are encountered for the first time. Although not explicitly stated by RDF, a sensible interpretation would be:

- A *value* property points from a node to a literal.
- A *relational* property points from one node to another node.
- A node that has a favourite colour is a person or an animal – but not all things can have favourite colours! Presumably geographical locations have no concept of favourite colours.
- When a subject knows an object it does not imply that the object knows the subject as well.

Such rules can be expressed with the help of vocabularies and ontologies, which are themselves expressed in RDF and accompanied by documents defining their meaning. The W3C specified the RDF-Schema (RDFS) vocabulary [25] and the Web Ontology Language OWL [42] as tools for the definition of semantics. With carefully constructed ontologies, logical reasoning on RDF data can be enabled. Many other organisations specified general-purpose and domain-specific vocabularies and ontologies based upon RDFS and OWL.

The graph form is the conceptional model of RDF. In practice RDF is often translated into a text representation. This process is called serialisation. A simple approach is to write a single line for each triple. Some formats allow to define abbreviations for recurring prefixes in order to make the notation more human friendly. Listing 2.2 is an example of the N3 notation [8]. The graph shown in figure 2.1 is fully contained in listing 2.2 although it does contain additional triples as well. In this work N3 and SPARQL syntax, which is closely related to N3, will be used.

person	name
<http://example.org/person/bob>	"Bob"
<http://example.org/person/michael>	"Michael"

Table 2.1: Results of the simple SPARQL query in listing 2.1.

2.1.2 SPARQL Query Language

This section provides an introduction to a selection of concepts in the SPARQL query language [37]. SPARQL is not only a query, but since version 1.1 a data manipulation language as well. Many powerful concepts are not covered here.

Listing 2.1: A Simple SPARQL Query

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX ex: <http://example.org/person/>

SELECT ?person ?name
WHERE {
        ?person foaf:knows ex:daniel .
        ?person foaf:name ?name .
}
```

Basic queries in SPARQL are formulated as *graph patterns* which are sets of *triple patterns*. A triple pattern is an RDF triple that can contain a variable in place of any of its three elements. Variables are denoted by either a question mark (?) or a dollar sign ($) followed by a label. Listing 2.1 presents a very simple query that consists of a single graph pattern with two triple patterns and two variables (*?name* and *?person*). The results set for this query will contain references to persons that are in a *knows* relationship to another person referenced by the URI *http://example.org/person/daniel*. Or simply (but imprecisely) put: People that know *daniel*. For each URI reference, the person's name is retrieved as well. Table 2.1 lists the results of this query applied to the graph in Listing 2.2. If the SELECT clause contained only the *?name* variable, the result would contain only the *name* column.

Listing 2.2: A Sample RDF Graph in N3 Notation

```
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix ex: <http://example.org/person/> .

ex:daniel foaf:name "Daniel" .
ex:daniel foaf:knows ex:bob .
ex:bob foaf:name "Bob" .
ex:bob foaf:knows ex:daniel .
ex:bob foaf:knows ex:michael .
ex:michael foaf:name "Michael" .
ex:michael foaf:knows ex:daniel .
```

Returning to listing 2.1, the query describes a graph where some triple elements are variables. We say they are not *bound*. An engine resolving this query has to produce graphs from the graph pattern by binding each variable to a value. A single graph patterns can have many sets of valid bindings. A bound graph pattern is a valid solution if the underlying RDF data set contains an equivalent sub-graph. In table 2.1 each row represents a valid binding for the variables of the query.

2.2 RDF Data Management

When dealing with small quantities of data very simple approaches can be used to publish RDF. RDF can be served as static or dynamically generated files or embedded into XML formats through microformats such as RDFa [2, 18, 27]. Adobe XMP [29] provides a framework for embedding RDF metadata into many binary file formats.

In order to manage large quantities of data, the underlying data must be managed by a database management system (DBMS). The following sections provide an overview of such solutions.

2.2.1 Mapping Relational Data to RDF

Because of their maturity and historical dominance, many data sets are stored and structured according to the needs of relational DBMS. It is possible to perform a mapping between the structured relational data and semi-structured RDF data. This can be beneficial in order not to disrupt legacy systems [27] or in order to expose relational databases to the web of data. The W3C maintains a working group specifying standards for relational to RDF mappings [4, 15]. They are primarily applied on data stored in a relational domain model. This is different from using relational databases for the storage of RDF graphs. In the former case data is exposed via an RDF view of the relational data structure. In the latter case the data is not structured for relational databases but relational databases are used as storage layers.

Some systems that employ relational to RDF mapping strategies are D2RQ [11], Virtuoso RDF Views [19], or Triplify [5]. Hert et al. [28] offer an extensive list and comparison of languages and frameworks.

2.2.2 Triple Stores

Most RDF storage systems decompose RDF graphs into triples for storage. These solutions are called *triple stores* [40]. Some approaches are introduced here, in order to facilitate discussion of the Hexastore approach introduced in Section 2.3.

Early triple stores used relational databases (3store [26]) or Berkley DB (Redland [7]) for persistent storage. These systems store triples directly into a large table. They are suited to resolve statement-based queries, where a triple pattern has one or two of its elements unbound. But they do not support complex queries efficiently [35]. Jena TDB [46] creates *property tables* that attempt to group information about subjects and

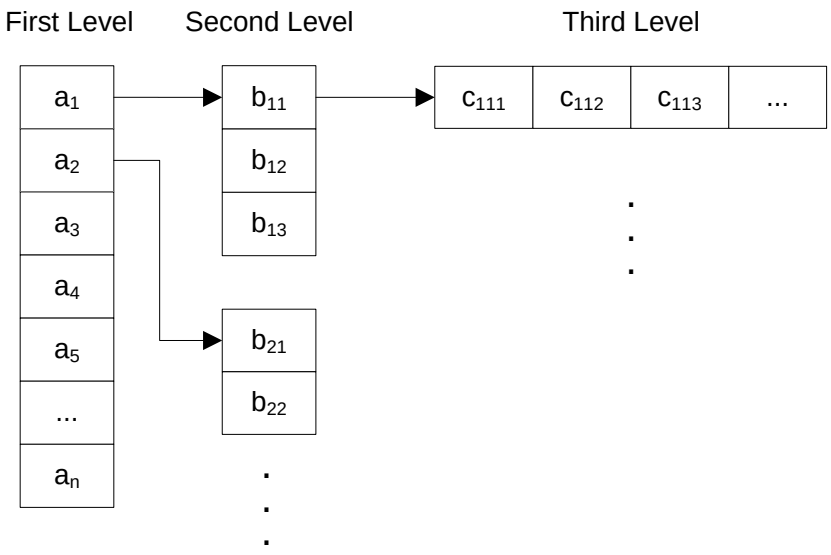

Figure 2.2: An abstract view of a Hexastore Index. The index is structured into three levels (in [44] they are referred to as *Type 1-3*). The depicted entries (a_1, b_{11}, c_{111}, etc.) represent identifiers of triple elements. Each first-level entry points to an ordered set of second-level entries and each second-level entry points to an ordered list of third-level entries.

their properties together. But it has problems answering queries that require multiple *property tables* to be combined [1]. The vertical partitioning scheme proposed by [1] is based on relational databases. It uses a dedicated table for each unique property. Every table has two columns, one for subjects and one for objects. Tables are ordered by subject. Wilkinson [46] implemented a very similar extension for Jena, called *multi-valued property tables*. Both of these approaches assume that most queries will have a bound property. These approaches do not perform well for queries with unspecified property because all property tables must be accessed. Weiss et al. [45] argue convincingly against assumptions about the type of queries and propose a solution in Hexastore.

2.3 Hexastore

Weiss et al. [45, 44] proposed a multi-index RDF storage scheme that provides efficient querying of RDF data without favouring any type of queries. Many approaches make assumptions about the type of queries or the data that is stored in them (Section 2.2.2). But RDF and SPARQL do not impose any such assumption and the Hexastore approach honours that.

2.3.1 Index Structure

Hexastore can be seen as an extension of the vertical partitioning approach by Abadi et al. [1]. Vertical partitioning creates ordered two-column tables for each unique property. This is closely related to what is called a PSO index in Hexastore. PSO stands for the initials of the elements of an RDF triple (predicate, subject and object) and the order in which these elements are indexed. Figure 2.2 depicts a Hexastore index. The PSO index has three levels. The first level contains all unique predicates. Each predicate points to a sorted vector of associated subjects on the second level. Associated means that the first-level predicate and second-level subjects appear together in triples. Each element in the subject vector points to a sorted list of associated objects on the third level. The third-level elements are associated with the second-level elements and transitively with the first-level elements as well. Therefore the third-level list contains only objects that appear together with the predicate *and* subject along the association chain. The major difference between [1] and Hexastore is that Hexastore creates six indices. One for each of the $3! = 6$ permutations of the elements of a triple: SPO, SOP, PSO, POS, OSP, and OPS. This is where its name derives from. Each index can be used for different queries. SPO can be used to resolve triple patterns with bound subject or with bound subject and predicate. But not for queries that do not have a bound subject. For such queries either O- or P-headed indices can be used. Refer to Table 2.2 for a full listing of triple pattern to index mappings.

Query	**Indices**	**Level**
s p o	SPO (any)	third
s p *?o*	SPO (or PSO)	second
s *?p* o	SOP (or OSP)	second
?s p o	POS (or OPS)	second
s *?p ?o*	SPO or SOP	first
?s ?p o	OSP or OPS	first
?s p *?o*	PSO or POS	first
?s ?p ?o	SPO (any)	(full-index-scan)

Table 2.2: There are 8 possible triple patterns. For each pattern at least two indices exist that contain the result set. The level column indicates the index level or type that needs to be queried. Index alternatives in parentheses indicate that the other alternative may be preferable due to element order. The preference between alternatives on the first level depends on the desired order of the result set.

This index structure naturally fits semi structured RDF data. Many relational structures are affected by sparsely populated multi-column tables when dealing with semi structured data. In Hexastore, no *NULL* values have to be inserted. An indexing scheme like [1] has to access all property tables to resolve queries with unbound predicate – even though many tables will not contain relevant data. Hexastore possesses an appropriate

string	**id**
<http://example.org/person/peter>	100
<http://example.org/person/franz>	101
<http://example.org/person/lukas>	102
<http://xmlns.com/foaf/0.1/knows>	200
<http://xmlns.com/foaf/0.1/name>	201
<http://xmlns.com/foaf/0.1/mbox>	206
“Franz”	416
“Lukas”	596
<mailto://franz@example.org>	731
<mailto://lukas@example.org>	744
...	...

Table 2.3: A partial view of a mapping from *string-space* to *id-space*. Strings are encoded to fixed size identifiers (the size is three bytes in this example).

index for any query type and does not have to access irrelevant data. Most importantly all lookups can be done in constant time and all joins between triple patterns can be merge-joins because there are sorted lists for any access scheme. Consider that every triple pattern maps to at least two indices (Table 2.2). If only individual triple patterns are considered, some of the indices are obsolete. But when joining multiple triple patterns, the order of the result set matters for the ability to perform merge-joins. Merge-joins are fast and can be done in linear time. Many other approaches must revert to more expensive joins because they do not contain result sets in the correct order for every combination of triple patterns.

2.3.2 Storage Space Consumption

The six indices maintained by Hexastore come at a price. The storage space requirement is high and inserts and updates are slow. Weiss et al. [45] argue that storage space is not a major concern because it is cheap and easily extensible. Further they observed that the storage requirement is less than six-fold that of an approach that stores each triple only once. In worst case the storage requirement is five-fold because some third-level element lists can be shared between pairs of indices. For example the SPO and PSO indices have identical third level O-list. The same is true for SOP and OSP which share the same P’s and POS and OPS which share the same S’. Consequently each triple element has to be stored only 5 times in worst case. Consider the indices a subject has to be stored in: once in each of SPO, SOP, PSO, OSP – but only once for the two indices POS and OPS, because they share the terminal list of subjects. In practice the real storage requirements are usually below a five-fold increase because most resources do not appear only once in the same place of a triple and there are no duplicate entries in the indices.

Furthermore storage requirements are reduced, because RDF triple elements are dictionary encoded before they are indexed. All triple elements receive a unique identifier

(id) which is stored in the indices instead of the string representation. The identifiers are usually more concise than the string representation and can be stored more efficiently. Table 2.3 contains an example of a string to identifier mapping. The dictionary encoding requires Hexastore to maintain an additional two-way mapping between the identifiers and string representations. Nevertheless storage space is saved, because most triple elements appear more than once in a given data set and the identifiers are usually shorter than their string representation.

2.4 Rdfbox – A Hexastore Extension

Rdfbox is a Hexastore extension written in Python and Cython. It uses third party key-value stores for persistent storage. A Python API for triple loading and SPARQL query execution is provided. A web frontend for query execution is included as well. The query execution engine is a custom implementation and currently does not support full SPARQL syntax.

This section describes the index operations and how Rdfbox uses sorted key-value stores to implement a Hexastore indexing scheme. Then a brief description of the Rdfbox architecture follows. It concludes with a description of the query execution engine and index loading process.

2.4.1 Index Operations

In [44], Weiss and Bernstein define two required operations for a Hexastore implementation:

getIndex Given a first-level item a there should be an operation that efficiently retrieves the second-level index associated to a.

getSet Given a second-level item b there should be an operation that efficiently retrieves the third-level ordered set associated to b.

These operations are sufficient to resolve queries. Query resolution requires binding values to variables in triple patterns. A triple pattern can have all three elements unbound, in which case every triple matches and the resolution amounts to a full index scan. But in all other cases there is at least one element bound and therefore there are two indices that contain this element on the first level (Table 2.2). In this case we choose one index and obtain the second-level index using the *getIndex* operation. If another element is bound, we use the *getSet* operation to retrieve all results for this triple pattern. If only a single element was bound, we have to use the *getSet* operation on each entry in the second-level index to obtain all results. This covers all cases for a single triple pattern. More advanced query resolution will be discussed in Section 2.4.4.

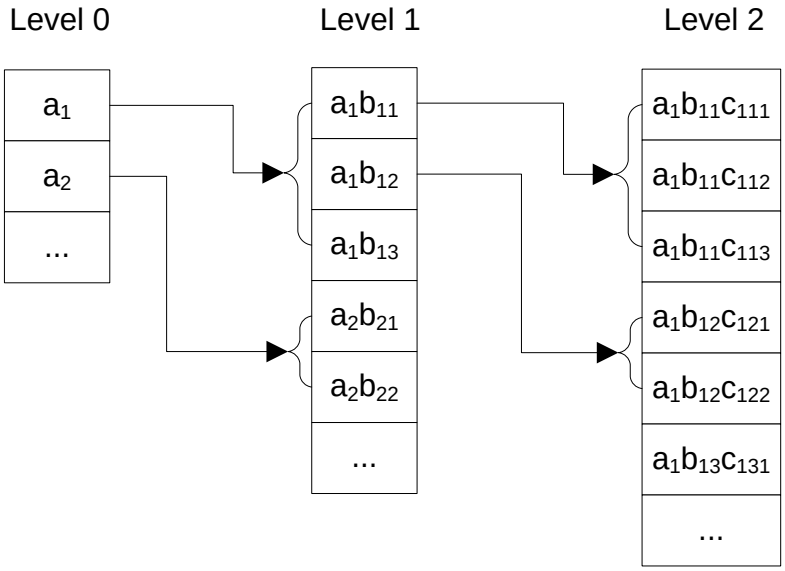

Figure 2.3: An abstract view of the Rdfbox index structure. The brackets indicate how the individual sets and lists in Figure 2.2 are mapped into a single ordered set on each level.

2.4.2 Implementing Hexastore with Ordered Key-Value Stores

Hexastore [45] does not specify how indices are persisted. A later paper by Weiss and Bernstein proposes a novel solution called *vector storage* [44] but Rdfbox delegates this task to third party key-value stores that are ordered by key. Originally only a Tokyo Cabinet [31] B+Tree database has been supported. Ritter added support for other databases in [39]. These include leveldb [16] which is based on the BigTable [14] but not distributed.

Rdfbox implements each index-level as a standalone ordered key-value database. Therefore $3 * 6 = 18$ databases are maintained. They are named after the index permutation and a zero-based level number (spo0, spo1, spo2, pso0, pso1, etc.). Optionally three *level 2* indices can be omitted because they contain the same entries as other indices (Section 2.3.2). Each database entry consists of a key and a value. The entries are ordered by key. The exact nature of the ordering is unimportant. The only requirement is that entries sharing a common prefix are grouped together. Each index level offers a cursor interface. Cursors are moveable pointers to the entries in an index. A cursor is required to implement methods to move to the beginning of the index, to advance to the next entry and to jump to an entry specified by a key-prefix. At any time the cursor allows access to the entry it is currently pointing at. Multiple cursors can be opened on a single index at the same time.

Figure 2.3 depicts and abstract view of all three index levels. The entry keys are fixed-length byte arrays. In *level 0*, the key is the encoded first-level element (the encoded subject in case of an SPO index). In *level 1* the key is the encoded first-level element concatenated with the encoded second-level element (subject+predicate in SPO). In *level 2* the key consists of all encoded triple elements concatenated together (subject+predicate+object in SPO). For a concrete example, Listing 2.3 contains a triple T1 to be indexed:

Listing 2.3: Unencoded T1

```
<http://example.org/person/franz> <http://xmlns.com/foaf/0.1/name> "Franz".
```

Assuming Table 2.3 contains the string-to-id mapping, Listing 2.4 shows the encoded version of T1 and the keys of the index entries for each index and level.

Listing 2.4: Encoded T1 and index entry keys

```
T1: 101 201 416

spo0: 101                    sop0: 101
spo1: 101201                 sop1: 101416
spo2: 101201416              sop2: 101416201

pso0: 201                    pos0: 201
pso1: 201101                 pos1: 201416
pso2: 201101416              pos2: 201416101

osp0: 416                    ops0: 416
osp1: 416101                 ops1: 416201
osp2: 416101201              osp2: 416201101
```

The $getIndex(a)$ operation is implemented by opening a cursor on a *level 1* index and jumping to the key-prefix specified by the argument a. This moves the cursor to the first entry that start with the specified prefix. Calling *next* on the cursor until the key prefix changes or the cursor reaches the end of the index, retrieves the *level 1* index for a. This range is indicated by brackets on the *level 1* index in Figure 2.3. The *level 1* element can be obtained by splitting the key at index $len(a)$ because all element identifiers have the same length.

The $getSet(b)$ operation is implemented in the same way by opening a cursor on a *level 2* index and jumping to the prefix specified by ab. The *level 2* element can be obtained by splitting the full key at index $len(a) + len(b)$. The ranges that cover a set for given values of a and b are indicated in Figure 2.3 by brackets on the *level 2* index.

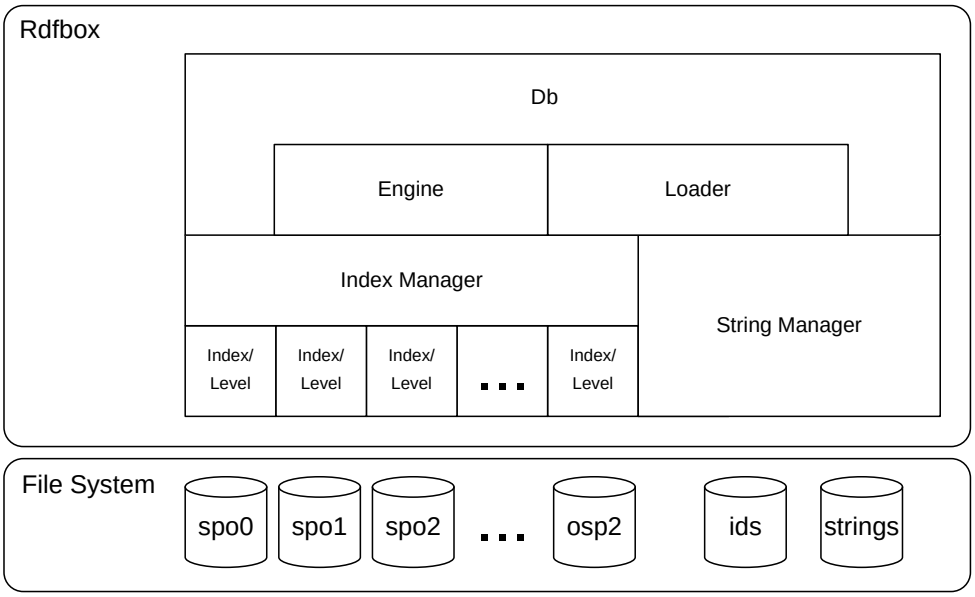

Figure 2.4: A layered view of the most important Rdfbox components. Not all Index/Level components and data stores are shown. In reality 15 to 18 data stores are used, three levels for each of the six index permutations. As described in Section 2.3.2, some levels can be shared – resulting in only 15 data stores for indices.

2.4.3 Architecture Overview

Figure 2.4 provides simplified layered view of the most important Rdfbox modules.

Db The Db module offers a Python API for interacting with Rdfbox. The most important methods are opening and closing the triple store, loading triples from a specified file and executing SPARQL queries supplied as strings.

Engine The Engine executes and resolves SPARQL queries. This module is explained in Section 2.4.4.

Loader The Loader module is responsible for loading triples into Rdfbox. Section 2.4.5 provides more details on the loading mechanism.

Index Manager Rdfbox uses 15 or 18 separate indices. One for each index level. The number varies depending on whether index sharing is enabled or not. The Index Manager abstracts the complexity of handling all these separate indices from its clients. Another important task is the joining of partial results coming from the indices.

Index/Level The Index and Level modules are mainly abstract representations of the key-value stores. Each concrete storage module has to implement the Index inter-

face in order to plug into Rdfbox. The Cursor API of Rdfbox is implemented here too.

String Manager The String Manager handles the translation of triple elements between their identifiers and the string representations and vice-versa.

2.4.4 Query Execution Optimisation with Cardinalities

Rdfbox accepts SPARQL 1.0 queries as strings an delegates query parsing to Rasqual [6]. The query execution engine is a custom implementation and currently does not support full SPARQL syntax.

So far the values of index entries were not mentioned. They are used for optimisation of the query execution plans. Rdfbox maintains detailed statistics about the cardinalities of triple elements as values in the index databases. For each key the corresponding value is a count of index entries on the next level. An example: For each encoded subject s, the spo0 index entry with key s contains the number of spo1 entries starting with prefix s. This number is the cardinality of predicates associated with s because the second-level elements in SPO are p-elements. For each s and p, the spo1 index maintains the cardinality of objects associated with s and p. The value of the spo2 index is not used.

Listing 2.5: SPARQL Query with exemplary variable cardinalities

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/> .
@prefix ex:  <http://example.org/person/> .

SELECT ?friend ?mail
WHERE {
        ex:peter foaf:knows ?friend.     #1: ?friend: 10
        ?friend foaf:name ?name.         #2: ?friend: 1000, ?name:1000
        ?friend foaf:mbox ?mail.         #3: ?friend: 100, ?mbox:100
}
```

Listing 2.5 contains a simple SPARQL query with one graph pattern. The control flow of the query resolution engine can be followed in Figure 2.5. When resolving a graph pattern, the first step is to fetch the cardinalities for all variables. Then the most selective variable is chosen to be resolved. We can see in Listing 2.5 that *?friend* has the cardinalities 10, 100, 1000, *?name* has 1000 and *?mbox* has 100. The best (smallest) cardinality is held by *?friend* with a minimum of 10. Joining all three triple patterns on *?friend* will yield at most 10 results. This is the optimal strategy based on the available information. A join returns all valid identifiers that can be bound to *?friend*. The next step starts to iterate over each result of the join. Each iteration creates a context in which *?friend* is bound to a different result and recursively calls the resolution algorithm again.

Branching out the resolution tree too much early on has to be avoided. Let each *person* in the underlying data set have only one foaf:name and one foaf:mbox. If *?name* is resolved first, then the join returns 1000 identifiers, the resolution tree creates 1000

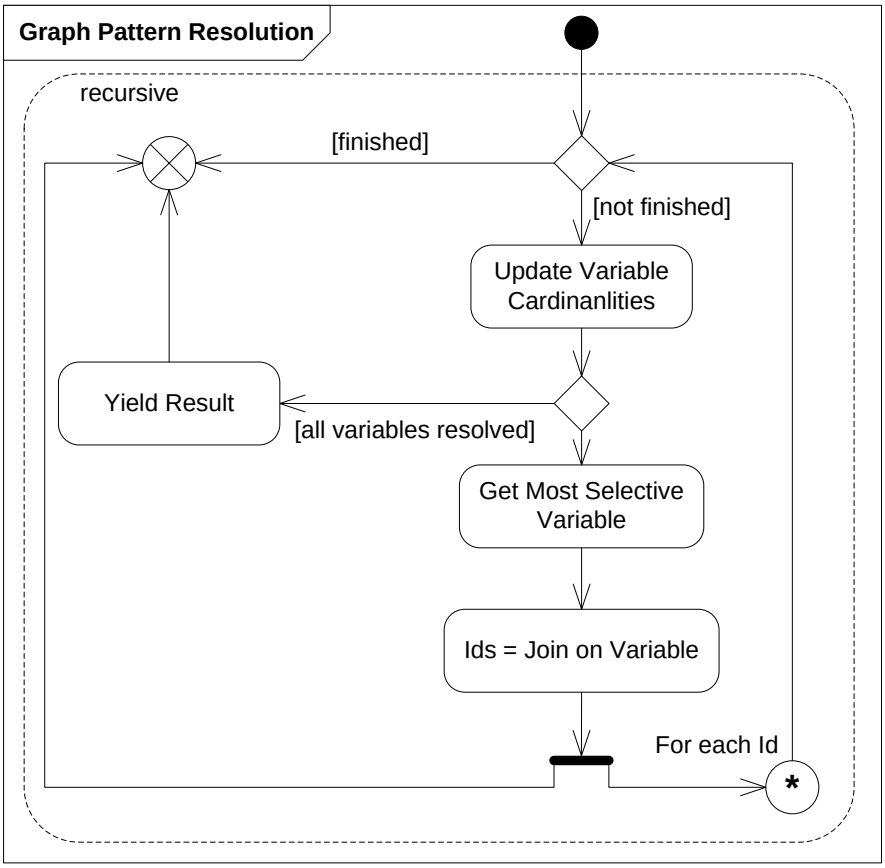

Figure 2.5: A UML activity diagram of the query execution engine's control flow. It splits into several control flows at the fork after *Join on Variable*. All control flows are resolved recursively in a single thread and each control flow can create even more recursive control flows until all variables are resolved.

recursions and each recursion performs a join on *?friend*. If *?friend* is joined first, only one join on *?friend* has to be performed and for each of the maximally 10 results, a single *?name* (and *?mbox*) has to be resolved.

A join on *?friend* creates three cursors, one for each occurrence of the variable. The index manager handles the choice of a suitable index to open the cursor on. The cursor for triple pattern #1 will be opened on the spo2 or pso2 index. It returns an ordered list of o's, corresponding to a *getSet* operation. The other cursors are opened on a pos1 index which corresponds to a *getIndex* operation. Crucially it returns an ordered list of o's as well. Therefore three ordered lists of o's can be merge-joined. The query could be resolved as well with cursors on pso1 instead of the pos1 index. But *getIndex* on pso1 returns a list ordered by s. A more expensive sort-merge join or hash-join would be necessary. This makes a key feature of Hexastore apparent. At each level a suitably

ordered list can be retrieved.

Because cursors provide efficient sequential access, the engine does not perform joins pair-wise and instead joins on all cursors at the same time. As soon as one of the participating cursors is exhausted, the merge can stop and no more unnecessary values are fetched. Therefore there is no preferred order of joins when joining on a single variable.

2.4.5 Loading RDF graphs into Rdfbox

Rdfbox has a dedicated module to load serialised RDF graphs into the indices. The loading process consists of four steps. First the input is dictionary encoded. This step has to be synchronised because the dictionary encoding must assign unique identifiers to triple elements. It must be avoided that different elements are assigned the same identifier. Second all six permutations are created for each triple. Third all permutations are sorted and finally, the data is loaded into the indices. Facilities that perform all steps on input larger than memory are provided and [39] implemented two variants that perform most steps in parallel. One variant does not sort the data before loading. Sorted input is beneficial for B+Trees but for other index structures it can be unsuitable or unnecessary.

2.5 Selected Distibuted Key-Value Stores

This section introduces the Hyperdex [20] and Apache Accumulo [22] distributed key-value stores, because they have been selected for the implementation of distributed Rdfbox indices as part of the task definition.

2.5.1 Hyperdex

Hyperdex [20] is a distributed key-value store that offers a search API that allows to retrieve entries not only by their primary key but also by secondary attributes. It supports the search operation efficiently with an approach called *hyperspace hashing*. It requires the specification of a fixed schema at the creation time of a table (or space as it is called in Hyperdex). The schema of each space consists of a primary key and a set of attributes. The key and attribute values are typed. Primitive types, strings and lists, sets, and maps constructed with those types are supported.

Hyperspace offers strong consistency guarantees and fault tolerance. In YCSB benchmarks conducted by the authors [20] it outperform Cassandra and MongoDB. Unlike Accumulo (Section 2.5.2) the Hyperdex coordinator and daemon software have very low requirements in terms of infrastructure and external dependencies and are very easy to set up. It comes with Python client bindings which makes interfacing with Rdfbox easy.

Hyperspace Hashing and the Search Operation

A multidimensional hyperspace is created from the attributes of a space, making each attribute an axis. The hyperspace is divided into regions that are assigned to servers. Hyperspace hashing is employed to map each entry to coordinates in this hyperspace by hashing each of its attributes to a location on the corresponding axis. These coordinates can then be used by a client to identify the servers, that store the desired entries. When all attributes are fully specified, the coordinates will always map to exactly one server. A search operation is specified by values or value ranges for each attribute. Unspecified attributes have an unlimited value range. Conceptually a search operation constructs a hyperplane for each attribute. It intersect the corresponding axis in one point and all other axes in every point. Figure 2.6 illustrates this on a three dimensional example. The more attributes are specified, the more planes are constructed. Search results are found where all hyperplanes intersect. A fully specified search intersects only in one point. The more attributes are specified the less servers need to be contacted to perform a search.

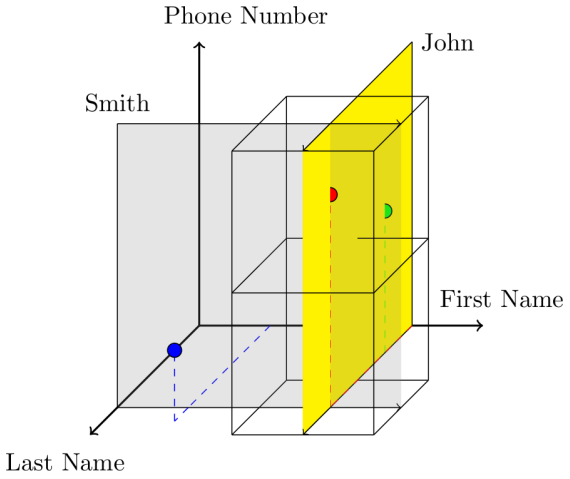

Figure 2.6: A search operation in a three dimensional hyperspace. The axes represent the attributes *first name*, *last name* and *phone number*. A search with specified *first name* = *"John"* and *last name* = *"Smith"* has been issued. Each specified search attribute forms a plane that intersect its axis in one point. Both planes intersect on a line on which all search results (phone numbers of John Smiths) can be found. The boxes indicate the regions that the intersection falls into. A region maps to a server. Graphic by [20].

In order to prevent an explosion of dimensions due to too many attributes, high dimensional hyperspaces can be partitioned into a set of lower dimensional subspaces. One such special subspace is the key-space. It is a one dimensional space for the primary key to support efficient key based operations.

2.5.2 Accumulo

Apache Accumulo [22] is a distributed and ordered key-value store that has been modelled after Google's BigTable [14] design. It is designed to scale to very large clusters of commodity hardware. Accumulo is implemented in Java. It depends on Apache Hadoop [24] for its underlying distributed file system (HDFS) and it can plug into Hadoop's MapReduce framework as a data source or sink. Apache Zookeeper [21] provides configuration management and distributed synchronization and Apache Thrift [43] is used for communication across the cluster.

BigTable Data Model

BigTable [14] has been designed for wide applicability. It supports batch-oriented processing and latency-sensitive data provision to end user applications. It offers proven scalability to very large data, high performance and high availability. The data model is simple with limited schema options. Each table is a distributed sorted map. Each entry consists of a key and a value. The value is always an uninterpreted string. It is up to the application to serialize complex data structures accordingly if needed. The key consists of three dimensions: a row, a column and a timestamp. Row and column are strings. The column has two elements. A column *family* and a column *qualifier* separated by a ":" character. The timestamp value is a 64 bit integer.

$$(\text{row, columnFamily:qualifier, timestamp}) \rightarrow \text{value}$$

Table entries are maintained in lexicographic order by row first and in decrementing timestamp order second. Furthermore tables are split into tablets at row boundaries. A tablet is assigned to at most one server at a time. This data model has some interesting implications:

- Because entries are sorted by row and tablets are guaranteed to contain all entries with the same row value, applications can reason about the locality of data by designing the elements of keys accordingly.
- Sequential access to to short ranges requires communication with only few servers because similar row keys are likely on the same server.
- Data can be versioned. When inserting different values with same row and column values, the timestamp key element naturally sorts data such that more recent versions appear first. A garbage collector can be configured to only keep the n most recent entries.

BigTable does not offer full transaction support but row operations are always serialisable. Access control is performed on column family level. Applications can be configured not have access to all column families.

Accumulo Data Model

Accumulo extends the described BigTable data model. The column model is extended by another field called *visibility*. This is used for cell-level access control to key-value pairs. Furthermore BigTable requires the explicit specification of column families before they can be used. In Accumulo this is not necessary. Because Accumulo is implemented in Java, it uses byte arrays to store key elements and value. The timestamp is stored as a long integer type.

User Defined Code Execution on Accumulo Tablet Servers

User applications can provide custom Java code that is executed on the servers when entries are being processed (when data is being read and when servers dump in-memory data onto the the file system). This code is implemented as Iterators that can be arranged in hierarchies. An iterator receives entries emitted by a previous iterator or directly from the table as input and can manipulate or consume the data before emitting it. Common tasks for iterators are filtering entries based on various criteria or aggregation of values. With iterators data can be processed on the servers where the data is saved.

3

Motivation

The Semantic Web continues to develop into a gigantic global data space as is demonstrated by the ever growing linking open data (LOD) cloud (Figure 3.1). With some data sets reaching sizes in the billions of triples new SDBMS (Semantic Database Management System) need to be developed. While centralized solutions and traditional B+Tree index structures may work for some use cases, they are often overwhelmed and limited to the resources provided by a single machine. We believe that distributed index structures offer a general and efficient approach to handling the largest of data sets with commodity hardware.

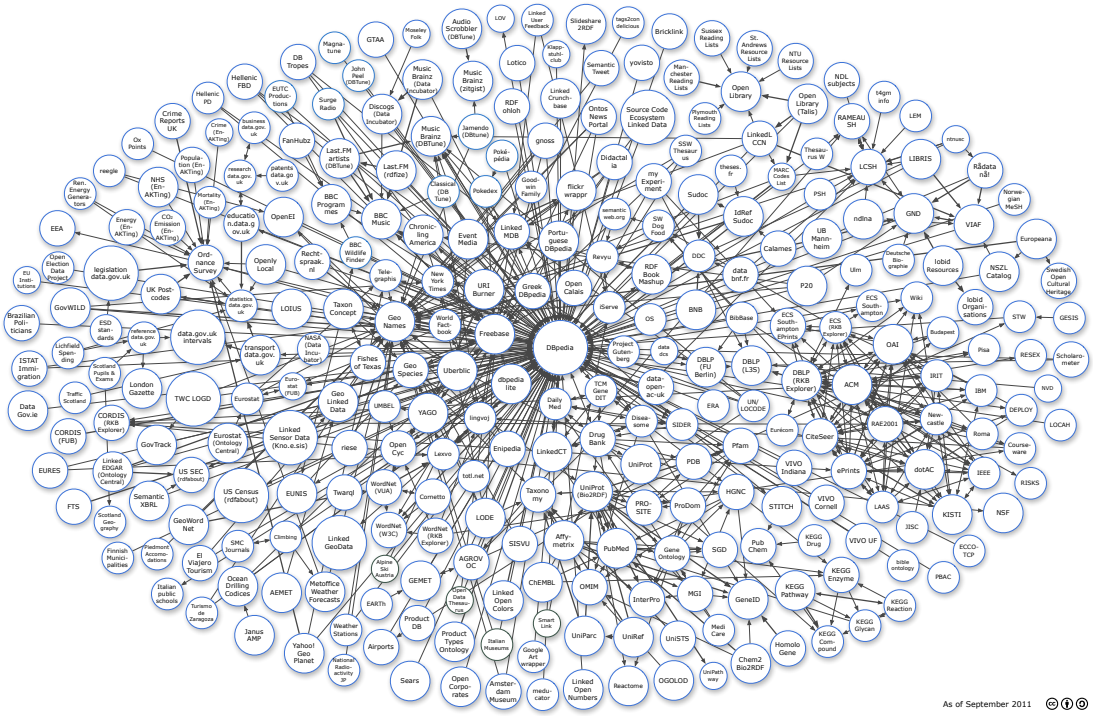

Figure 3.1: Linking Open Data cloud diagram as of September 2011, by Richard Cyganiak and Anja Jentzsch. *http://lod-cloud.net/*

With Rdfbox, a SDBMS developed at the University of Zurich and based on the Hexastore [45] indexing scheme, we have an high-performance centralized solution. In this work we aim to develop efficient distributed index plugins for the support of very large data sets.

3.1 Task Description

This master thesis' tasks are to develop efficient implementations of:

- A distributed index plugin based on the Hyperdex [20] project.
- A distributed index plugin based on the Accumulo [22] project.
- A parallel, high-performance and tunable MapReduce [17] based RDF loader.

All implementations are to be materialized in the scope of the Rdfbox SDBMS and tested. The source code has to be documented.

The goal is to support larger quantities of data than can be supported with centralized indices while maintaining the efficiency inherited from the Hexastore indexing scheme.

4

Implementation

This chapter describes the design and implementation of the Hyperdex and Accumulo distributed index extensions and the MapReduce based loader. The subject of the following discussions are the reasons for different implementation choices and the identification of optimisations based on preliminary evaluations. Special attention is given to the implementation of a multi-threaded query execution engine.

4.1 Hyperdex Index

Section 2.5.1 introduced Hyperdex, a distributed key-value store with a special search API [20]. It implements uses *hyperspace hashing* to support efficient searches of entries by secondary attributes. Hyperspace hashing creates interesting options for structuring and looking up entries. Low infrastructure requirements and easy setup are additional advantages of Hyperdex.

At the beginning of this work the latest Hyperdex release was version 0.4.0, which did not offer ordered access to keys nor any method to retrieve entries ordered by attributes. Ordered access is required for Rdfbox' cursor API and the query execution engine is built around the concept of efficient ordered sequential access. Hyperdex' schema options allow for values to be lists, but values are always associated to one coordinate in the hyperspace and served by a single server. They can be replicated on other servers but this is only a fault tolerance mechanism and cannot be used for partitioning. To access elements in lists, the entire list has to be retrieved first. Therefore creating large lists of triple element identifiers would not scale. But ordered access is not strictly necessary to resolve queries. The search API is a powerful tool to retrieve the required entries – but in unpredictable order. Fast merge joins are not possible but other joins can be used. But the development version of Hyperdex introduced a sorted_search method which would enable ordered access starting with the next release.

The remainder of this section covers a first approach at implementing the Rdfbox extension using a single hyperspace for all entries. A discussion of the findings of a preliminary evaluation follows. It discovered serious deficiencies and which required a different approach. Then the implementation of this new approach is examined but because the desired results are not achieved. An analysis of the limitation of using Hyperdex as an Rdfbox backend concludes this chapter.

4.1.1 Single Hyperspace Index

Hyperdex is designed for cluster environments and large quantities of data [20]. For this reason a starting assumption has been made that a single hyperspace would be able to hold all data without significant performance degradation. Rdfbox uses a very simple data schema. An identifier-key is mapped to a cardinality-value. Depending on index level, the identifier-key (*idkey*) is a triple element (encoded) or a concatenation of encoded triple elements. This schema can be directly mapped into a two-dimensional space consisting of a string-typed key and an integer-typed value. Because Rdfbox creates six permutations from each triple, the supplied *idkey* is not guaranteed to be unique when all entries are written into the same space. The actual key entered into the space is made unique by prepending the index permutation and level (e.g. spo0201 for a spo0 entry with *idk* 201). For efficient searching additional s, p and o secondary attributes have been considered. A natural schema for RDF data that would allow to resolve each triple pattern in a SPARQL query with a single search in Hyperdex. Consider the triple patterns in Table 2.2, each bound triple element can be mapped to a specified secondary attribute. Each variable is an unspecified attribute. The search would return all valid bindings for all variables in the triple pattern. But this approach has two problems.

1. Because all three index levels are stored in the same space, the results also contain entries from different index levels.

2. Resolving results per triple pattern is problematic. A single triple pattern could return many results (potentially the entire index). And with unordered results, expensive hash joins are required, which keep many partial results in memory – alternatively memory requirements can be mitigated by performing more searches in Hyperdex.

The first problem is trivial to solve. Additional *index* and *level* attributes allow the search to be specified to return only results from the required index.

The second problem can not be entirely solved without ordered results. Rdfbox avoids the memory problem because the cursor API accesses only one entry from each involved index at a time and merge joins on sorted lists require very little memory. A full solution of this issue has been postponed because a new Hyperdex release with a sorted_search method has been anticipated. Furthermore it would be very ill-advised to invest a lot of time into a solution requiring expensive joins because one of the key advantages of the Hyperdex index schema is not to have to perform expensive joins at this level. Temporarily a simple hash-join extension has been implemented in the query engine – under the assumption that all partial results fit into memory.

Schema Definition

Awaiting the next Hyperdex release, experiments with different schemas have been made. There are a multitude of options but the more attributes are specified, the more data has to be transferred over the network when writing and reading entries. Assuming the

network might become a bottleneck but still wanting to exploit the ability to search for entries by secondary attributes, the s, p and o attributes have been dropped and the following schema has been implemented:

$$key{:}string \rightarrow idk{:}string,\ index{:}string,\ level{:}int,\ count{:}int$$

idk is the identifier of the entry as supplied by Rdfbox, *index* is the permutation of the index (e.g. spo), *level* is the index level and *count* is the cardinality.

Implementation of Index Operations

The Rdfbox operations *getIndex* and *getSet* are mapped to searches in Hyperdex. Sections 2.3 and 2.4.2 introduce these operations in an abstract manner. The attributes to search for, can be specified as a value and results must match this value exactly. Alternatively ranges can be specified in which case attribute values must fall into this range to match. For this index plugin, the s, p and o attributes have been replaced with the construction of an appropriate range for *idk*.

getIndex is mapped to a search operation as follows:

$$getIndex(a) \rightarrow search(idk = (a, end),\ index = permutation,\ level = 1)$$

getSet is mapped to a search operation as follows:

$$getSet(b) \rightarrow search(idk = (ab, end),\ index = permutation,\ level = 2)$$

The parameter *permutation* is the permutation string of the index on which the operation is applied (e.g. spo) and (a, end) is the range starting with a (inclusive) and ending with *end* (inclusive). Similarly ab is the concatenation of a (obtained from the preceding *getIndex* operation) with b. *end* represents the largest possible *idk*. This means, that each search is effectively open ended. The range could be limited to the largest possible *idk* starting with a but the query execution engine is relying on the range of results to go beyond the specified prefix or it does not produce correct results.

Sorted Search

With the release of Hyperdex 1.0.rc1, the index extension has been adapted to the new API. The most notable improvement has been the use of sorted_search. This variant of the search method returns results sorted by a specified attribute. While sorted_search method calls are somewhat slower, the ability to use merge joins instead of hash joins outweighs the drawbacks.

Preliminary Evaluation and Optimizations

With a fully functional index implementation, preliminary evaluations with larger data sets have been performed. As expected loading performance was significantly worse than with centralized indices. Without special optimisations, this had to be expected. An evaluation of loading times can be found in Section 5.2.1. More interesting was the evaluation of the query execution times. For these tests a dataset with 50,000 triples was loaded. All queries required minutes to resolve despite the data set being very small in relation to the goals of this thesis. The exact results were not recorded but *PQ1* required about ten minutes to resolve and *PQ2*-type queries about two minutes. Refer to Listings A.1 and A.2 in Appendix A.1 for the queries.

An investigation revealed that a single sorted_search operation takes almost two seconds with 50,000 triples loaded. But great number of searches had to be performed when hundreds of intermediate results had to be joined.

Optimisations The investigation helped to identify some optimisations that help to reduce the number and size of searches.

The sorted_search method requires the specification of the number of results that should be returned at most. Because Rdfbox stores cardinalities, this number can be retrieved. Initially it was assumed that this additional request would be too expensive and a constant number has been used instead. In case more results were required, the requested range has been shifted to the end of the last returned search result and another search was issued. Obtaining the correct cardinality made searches more efficient and provided an overall improvement. Two things could be achieved by this measure.

- Searches that were triggered because the initially requested number was too low, could be avoided. In general fetching more data is more efficient than searching for it multiple times.
- Many searches fetched too much data because the actual cardinality is very low (e.g. 1). With the optimisation a search never retrieves more results than it maximally needs.

This change introduced a new problem too. Sometimes the cardinality of a search is very large. It may be better to fetch only a part of the entries of the search. If two sets of search results are joined, one with a single result and another with 100,000 results, they can join at most on one single entry (because that is the size of the smallest participating set). If the result appears early in the large set, then most of the results have been needlessly retrieved. A configurable upper limit for searches has been introduced. A search never fetches more elements than this limit. This also helps to keep memory requirements lower. If more results are needed, they can be fetched in another search.

A final optimisation allows to avoid searches all together, when the search range intersects with a previous search range. A limited amount of searches is cached and before a new search is issued, the beginning of its range is compared to the cached search ranges. If it is contained, the cached results are returned. With searches being so expensive, this check is beneficial to do even if it does not avoid searches in many cases.

Subjecting the optimised solution to the same evaluations again, revealed a significant performance increase. The execution time has been reduced by a factor of 5-8. The exact results can be found in Section 4.1.2. Nevertheless query execution still did not perform sufficiently. Especially some execution times approaching one hour for the one million triples data set show that the solution does not scale. More investigation into the causes triggered a different approach which is described in the next section.

4.1.2 Multi-Space Index

The single hyperspace index suffered from slow searches. Even several optimisations that reduced the number and size of searches did not achieve sufficient performance. Most potential for better performance was believed to lie in execution time optimisations of the search operation itself. Therefore the factors which make searches slow have to be investigated. It was found that the total number of dimension did not have any measurable impact on search performance. But the total amount of entries did. Search execution times became longer as more data was loaded into a space. Consequently, the number of entries in a space had to be kept low in order to have fast searches.

The Hexastore approach produces many entries and their number can not be lowered without loss of information. The benchmark_1m data set contains 1 million triples (Section 5.1). But consider that each triple has six permutations and each permutation creates entries in up to three index levels. Effectively it produces 7.8 million entries when indexed. The exact number of entries for 1 million triples can vary a lot and depends entirely on the distribution of the triple elements in the data set.

But there is a simple approach to reducing the number of entries a space has. Instead of writing all entries into a single space, a dedicated space can be used for each index level. It is easy to implement and it would distribute the entries to 18 (or 15 if indices are shared) spaces – although the distribution is not even.

Schema Changes and Index Operations

The new version of Hyperdex (1.0.rc1) enabled searches for the primary key (and not only for secondary attributes any more). This was used to create a very simple schema for the multi-space solution. While it could be applied to simplify the single-space schema as well, it would not improve performance significantly.

Because each index level is now contained in its own space, the *index* and *level* attributes are not needed any more. The schema becomes very simple:

$$idk{:}string \rightarrow count{:}int$$

idk is the identifier as supplied by Rdfbox and *count* is the cardinality.

Only minor changes had to be made to the implementation of the index operations. Instead of using search, sorted_search is used. The optimisations as discussed in Section 4.1.1 are used as well. Instead of using the *index* and *level* parameters as attributes in the search, they are used to identify the space on which the sorted_search needs to be applied.

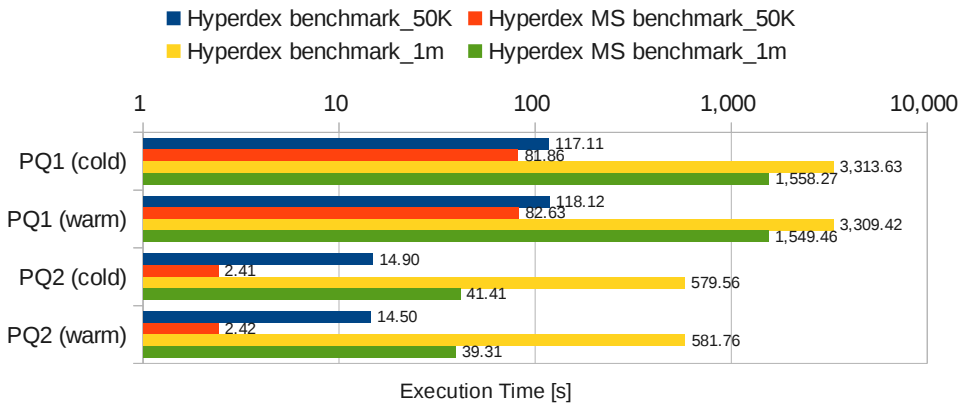

Figure 4.1: Comparison of query execution times for Hyperdex backends. *Hyperdex* refers to the optimized single space index and *Hyperdex MS* refers to the multi space index. *PQ1* and *PQ2* are names of the queries. The queries and the setup are described in Appendix A.1 and the data sets in Section 5.1. The multi-space version performs significantly better than the optimized single-space version. Mind the logarithmic scale.

Comparison to the Single Space Index

Figure 4.1 compares query execution times of the single-space and the multi-space index variants. The multi-space index performance is faster by a factor of 1.5-2 for *PQ1* and a factor of 6-15 for *PQ2*. This is a dramatic improvement. But with one million triples loaded most benchmark queries still need over a minute to execute. *PQ1* even about 25 minutes. Struggling with long execution times on small data sets pointed to fundamental problems in the way Hyperdex is used in Rdfbox.

4.1.3 Limitations of the Hyperdex-based Indices

The preliminary evaluations discovered that the search operation is a major bottleneck. Various optimisations that reduce the number, size and execution time of searches have been implemented. Performance did increase significantly but not to satisfactory levels and more untapped potential for significant optimisations could not be found any more. The problem is, that in this use case, search operations are principally too slow. Hyperdex advertises the search operation as an efficient method which is not untrue. A single search performs well when considered in isolation. But in the Rdfbox use case too many searches are required. I do not think that this is a problem of Hyperdex. Rather it lies in the way Hyperdex has to be used in Rdfbox. Query resolution in Rdfbox (see section 2.4.4) uses cursors and not searches. In the Hyperdex-based indices the *jump* operation, that moves a cursor to a specified position, is mapped to a Hyperdex search. But moving a cursor is assumed to be a very efficient operation and therefore its use is not restricted. Hyperdex searches are a much more complex operation and not designed

with such a usage pattern in mind.

Hyperdex is a distributed system, searching operations are delegated to remote machines. While a search is running, Rdfbox is mostly idle. From the operating system's point of view, a search is an I/O operation and while it is running, the system is waiting. Different programming models that allow to create systems that can do other work while some operation performs I/O do exist. Unfortunately Rdfbox' query execution engine has not been implemented with such a programming model. It only performs one I/O operation at a time and waits for its completion before any other work is done. Hyperdex is not the only index extension suffering from this limitation. Any index that spends significant time on I/O operations is affected. It can be a centralized index waiting disk I/O or a distributed index waiting for network I/O.

Changing Rdfbox' query execution engine to one that can do work in parallel is a complex task and the other problem of misusing the search operation for the emulation of cursors would not be solved. After careful consideration it has been decided to leave the Hyperdex plugin in the current state and focus on an Accumulo-based index that promised better overall performance because it supports a cursor-like interface natively. If the query execution engine had to be parallelised, it could still be done for Accumulo.

4.2 Accumulo Index

Apache Accumulo [22] is an open source distributed keys-value store and an implementation of Google's BigTable [14] design. Section 2.5.2 introduces Accumulo. In terms of infrastructure requirements it is very demanding. It requires a working Hadoop HDFS and MapReduce cluster and while no special purpose hardware is required, considerable amounts of memory and CPU cores are needed on each cluster machine to ensure stability of the system under load. To set up and configure such a cluster is a non-trivial task.

Accumulo stores all entries sorted by key and offers a cursor-like scanner API for data retrieval. Basic scanners retrieve data in sorted order. This makes it a perfect match for Rdfbox. Furthermore the ability of Accumulo to act as a source and sink of data for Hadoop MapReduce jobs, can be used to design a distributed MapReduce loading mechanism. During this work the latest Accumulo release (1.4.2) only offered a Java client API. In order to use Accumulo from Rdfbox' Python/Cython code, a solution had to be found.

This section covers the creation of a custom service that is used to interact with Accumulo from Python. Then it describes the design and implementation of the index extension an alternative, multi-threaded query execution engine. It ends by presenting the implementation of a distributed MapReduce [17] based loader for Accumulo.

4.2.1 Accumulo Client Proxy

In order to be able to interface with Accumulo's Java API, a custom service offering a narrow API specifically for Rdfbox has been implemented. Apache Thrift [43], an

interface description language (IDL), a software library and a code generation framework for efficient and reliable communication across programming languages has been used for this. Thrift allows the specification of services and data types in an abstract IDL and is used extensively throughout Accumulo. From the abstract service definition, a provided compiler can generate RPC (remote procedure call) server and client code in various programming languages.

Accumulo's Java client is very complex and could not be trivially replicated in Python. Instead Thrift has been used to create a Java service that implements a subset of the Rdfbox Index and Cursor API and wraps the Accumulo Java client. Because this service uses the Accumulo client on behalf of Rdfbox, it is called Accumulo client *proxy*. Using the Thrift Python library, Rdfbox can communicate with the client proxy via RPCs. Figure 4.2 explains the involved components and highlights the role of Thrift.

The client proxy process can run on a remote machine but in general it is preferable to run it on the same machine as Rdfbox. Accumulo's client uses a variety of advanced techniques to optimize data transfer. When the client proxy runs on a remote machine, the communication between the proxy and Accumulo is efficient, but the transfer between Rdfbox and the proxy is not.

4.2.2 Multi-Table Index

Mirroring the approach of the Hyperdex multi-space index (Section 4.1.2), 15 (or 18) tables are created in an Accumulo instance. This approach has been chosen because it allows a simple, direct mapping of Rdfbox indices into Accumulo tables, allowing quick verification of its suitability as an index backend. I wanted to avoid discovering severe problems late, as happened with Hyperdex.

Key Schema

In section 2.5.2 the BigTable data model has been introduced. It allows many options to structure a key into several columns. But inspired by Rya (Section 6.1) a very simple but efficient key schema has been chosen. In the row field of the key, the identifiers (*idk*) supplied by Rdfbox are used. The other key fields (column, visibility) are empty. *timestamp* is automatically set by Accumulo but it is not used. The value field (*count*) contains the cardinality value:

$$(idk,\ \text{“”},\ \text{“”},\ timestamp) \rightarrow count$$

This key format makes interaction with Accumulo simple and efficient. No filtering has to be applied and all table entries are in the order Rdfbox expects them in.

Implementing Index Operations

To retrieve data from Accumulo tables, clients use scanners. Scanners can be used as configurable iterators that return the table entries in lexicographic key-order. The *range*

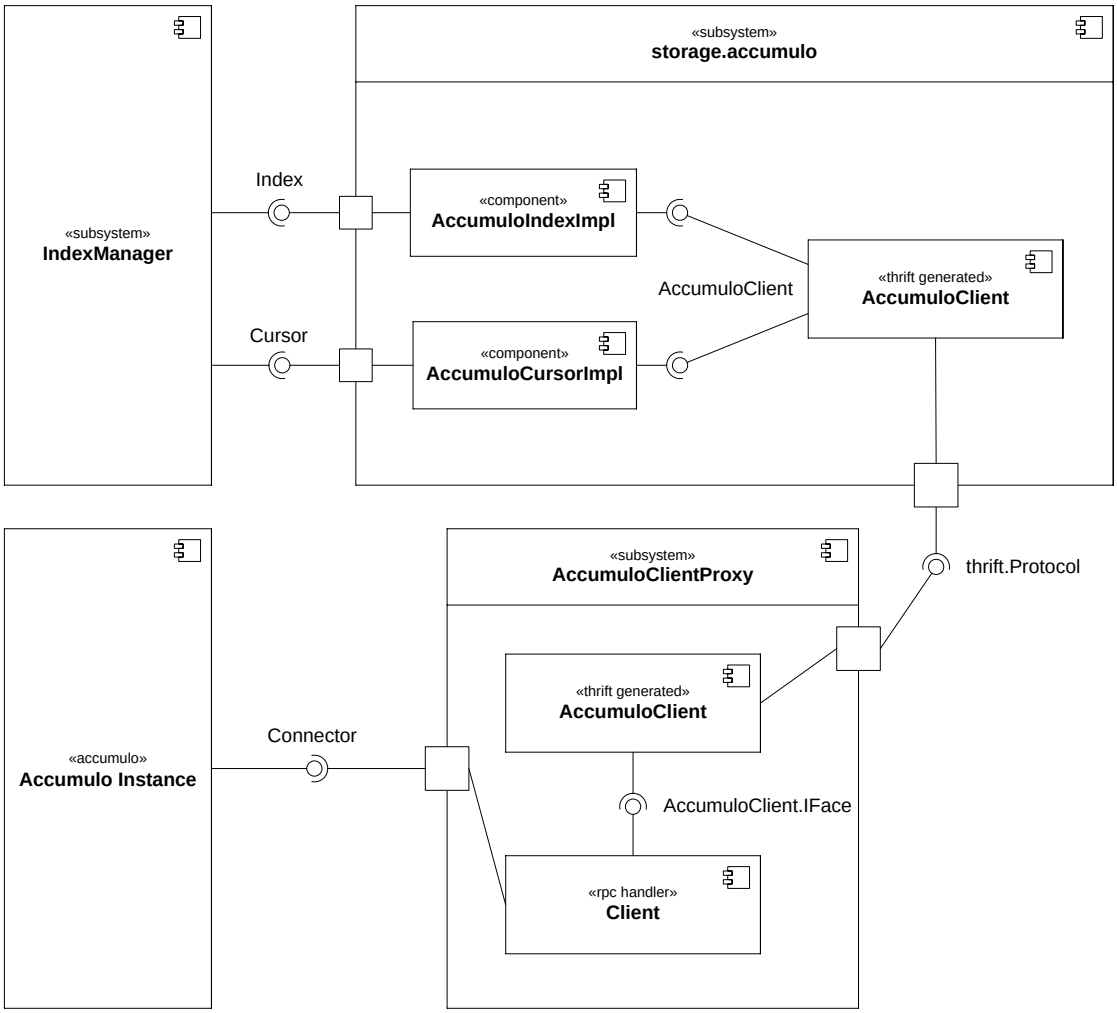

Figure 4.2: A UML component diagram of the components related to the client proxy. Thrift-generated components are marked accordingly. *AccumuloClient* is used by the index and cursor implementations to issue RPCs. It extends the service interface defined in the abstract IDL and contains additional methods to communicate via the Thrift protocol. The *thrift.Protocol* interface is in reality a protocol stack that can be configured in various manners (e.g. framed, buffered or binary format). It is the bridge between Python and Java programming languages. The communication is handled by components in the Thrift library (not shown). *AccumuloClient.IFace* is the service interface and *Client* in *AccumuloClientProxy* is the RPC handler, which implements the Accumulo client wrapper. The *Connector* interface is used by the RPC handler to communicate with the Accumulo instance.

of entries can be configured. Start and end keys of the range can be omitted, in which case the range starts at the beginning or continues until the end of the table respectively.

The Hexastore *getIndex* and *getSet* operations are implemented in Rdfbox by creating open-ended cursors starting at a specified key prefix. Then the cursors are advanced until the specified prefix changes (Sections 2.3 and 2.4.2). The same can be done with Accumulo scanners, therefore the implementation of these operations is a direct mapping of cursors onto scanners.

getIndex creates a scanner with a specified range:

$$getIndex(a) \rightarrow scanner(table = (index = permutation,\ level = 1),\ startkey = a,\ endkey = \infty)$$

getSet creates a scanner with a specified range:

$$getSet(b) \rightarrow scanner(table = (index = permutation,\ level = 2),\ startkey = ab,\ endkey = \infty)$$

$(index = permutation,\ level = levelnumber)$ represents a string created by combining the index permutation and index level (e.g. spo1). ab is the concatenation of the a with b. a can be retrieved from the preceding *getIndex* operation.

Preliminary Evaluation

Subjecting this solution to the same tests as the Hyperdex backends, revealed that it resolved queries an order of magnitude faster. A detailed comparison to Hyperdex can be found in Section 4.2.5. This meant that queries that took minutes to resolve in Hyperdex could be resolved in a matter of seconds. An analysis of where time is spent during query resolution showed that operations on the scanner were a major time sink. This is reminiscent of the problems with Hyperdex search operations (Section 4.1.1). A direct comparison is not possible because the scanner operates differently, specifically it starts to return results much faster than the sorted_search operation. The search operation has a big initial delay until the first result is sent. Therefore scanners operate very efficiently as cursors. The time spent on scanner operations can be largely attributed to the relatively slow data transfer over a network interface. Network access can not be avoided and the problem of waiting for I/O has already been observed in Section 4.1.1 with Hyperdex. The problem can be approached by creating a query execution engine that can do multiple I/O operations in parallel.

4.2.3 A Multi-Threaded Query Execution Engine

Rdfbox' default query execution engine is single-threaded and resolves queries recursively. Figure 2.5 shows an activity diagram of the resolution algorithm's control flow. Section 4.1.1 and the previous section discussed how the query resolution is often idle

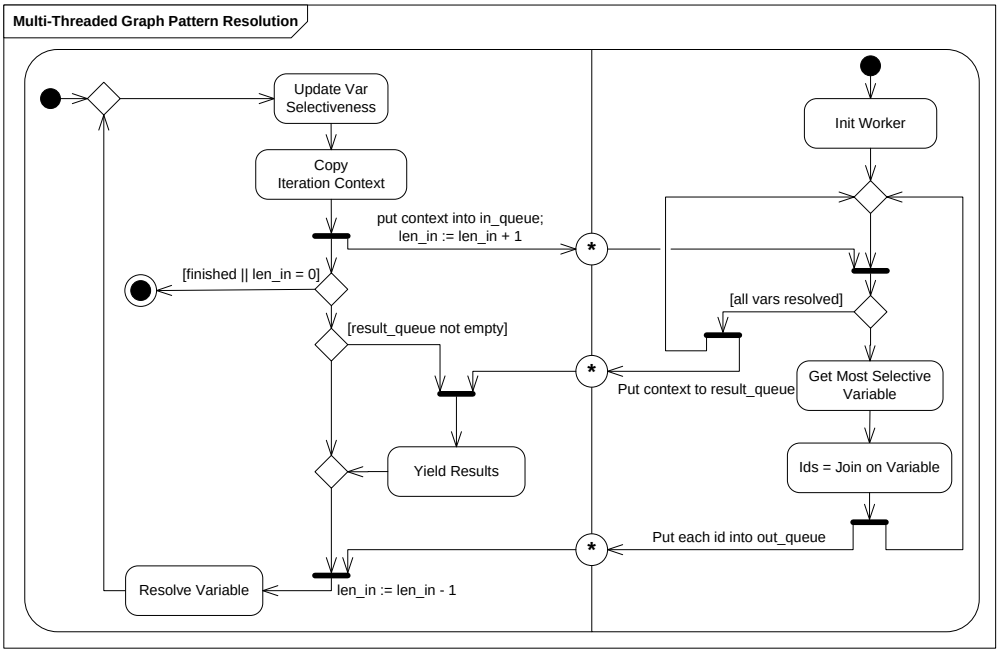

Figure 4.3: A UML activity diagram illustrating the multi-threaded query engine's control flow. A producer-consumer pattern is used to distribute work. The main thread feeds the worker threads with *contexts* to work on. A *context* primarily contains the graph pattern that needs to be resolved and a set of variable bindings. Worker threads produce two kinds of output. When a context is fully resolved (all variables have a binding), the context is put in a *result queue*. The main thread consumes contexts from this queue and translated the variable bindings to their string representation to yield a final result. The second output of the worker threads are identifiers of bindings for joined variables. The main thread uses the identifiers to create new contexts for later execution.

while it awaits I/O operations to complete. I/O waiting times can not be avoided directly. The network interface and disk access are slow operations. In the distributed case, the mere network latency makes every remote procedure call slow, even when the remote operation can work in memory. But there is a way to increase performance in such situations if the waiting time can be used to perform other work.

When the query execution engine performs a join on a variable, it generally retrieves multiple results. Each result gives rise to a context in which this result is bound to the variable it was joined on. Conceptually this creates a tree of contexts. At the root is a completely unresolved context. Each branch on the way towards the leaves is a new context where a variable is bound to one of the values created by the last join in the parent context. The leaves are fully resolved results. By default, each of those contexts has been resolved in series. Worse even, for each context all still unbound variables are

resolved recursively in the same manner. Consider an example: after the first join, there may be 100 results – thus 100 independent contexts for which the graph pattern has to be resolved. But 99 of these contexts need to wait until the first context and all the child contexts it creates are resolved. Only then the next context is worked on. If the system was under full load during resolution, nothing could be gained. But because the system is largely idle when resolving a context (because much of the work is performed remotely or spent transferring data), the idle time could be used to work on (some of) the 99 other contexts in parallel.

A multi-threaded query execution engine has been implemented. It resolves the described contexts in parallel. Python has a GIL (Global Interpreter Lock) that mostly inhibits the use of multiple CPUs, because most operations must synchronize on the GIL. Notably waiting for I/O tasks releases the GIL. Because distributed query resolution is not primarily a CPU, but I/O bound task, multi-threading is applicable for this case. An added benefit of multi-threading is shared memory between threads, which proved to be very useful.

The original implementation of the query resolution engine has been very ill-suited for parallelisation. It was implemented in a stateful manner. The contexts described before, did not actually exist as a separate entity. Only a single context could be active at a time because it stored its state in the engine instance. Extensive refactoring of various Rdfbox core classes has been necessary to extract the context into an independent entity that supported deep-copy operations to create duplicates of the current state.

Figure 4.3 describes the multi-threaded query execution engine. It uses a configurable fixed-size worker thread pool. The worker threads primarily perform I/O bound tasks, which causes them to yield control to other threads while waiting for results. The size of the thread pool does not depend on the number of CPU cores. In our experiments we found the number of 10 threads to be good. But this number depends on several factors. Optimally the local CPU utilisation should be high during execution. If the worker pool is too small, the CPU utilization is low, which means more work could be performed in parallel. When the worker pool is too large, it could overwhelm the local resources or lower throughput in any of the distributed nodes because too many requests are being processed in parallel. Because of Python's GIL, only one Python thread is doing work at a time. Monitoring has shown that a carefully calibrated thread pool size uses the system's CPU resources efficiently nevertheless. On one hand, the main thread became a CPU bound task because it does not wait for I/O now, especially when the worker threads can feed it with intermediate results continuously. On the other hand there are other processes involved as well (the operating system and the client proxy can schedule work on CPU cores with free resources).

The *input queue* that feeds workers with tasks, is a priority queue. Tasks that are added later, have a higher priority – effectively working like a stack. The priority queue realizes of depth-first like approach that attempts to reach leaves (results) in the context tree quickly. If the query has a specified limit, this helps to resolve it more quickly. Especially when the limit is significantly smaller than the full result set.

This engine does not replace its single-threaded variant. Which engine is used at runtime is determined by a compilation flag. Index extensions need to supply alterna-

tive cursor and merge-join algorithm implementations that operate correctly in a multi-threaded environment. Currently only Accumulo does this. For a comparison of the multi-threaded and single-threaded query execution engines, refer to Section 4.2.5.

4.2.4 Implementation of Distributed Joins

The development of a multi-threaded query execution engine has been triggered by the observation that much of the execution time is spent on data transfer over the network interface. It has been assumed that much of this data transfer could be avoided if joins could be performed remotely.

This section introduces a related solution, the Accumulo Wikipedia Search Example and then explains the implementation of distributed joins in the Accumulo index extension.

The Accumulo Wikipedia Search Example

Some use cases are explained on the Accumulo website. The Wikipedia search example [23] shares many aspects with the Rdfbox use case and inspired the implementation of distributed joins.

It is a word search over Wikipedia articles using a reverse index of words to documents. The Accumulo iterator mechanism (Section 2.5.2) is used extensively to implement much of the functionality. The index is built using an aggregator, a special kind of iterator that rewrites table entries by combining values of similar rows. Initially multiple entries for the same word are written if the word appears more than once in the indexed dataset. The aggregator combines all entries with the same word into one entry and attaches a list of articles that contain the word.

Searches for more than one required word are implemented as intersections of document lists. Consider a search for all articles with the words “wood” and “fire”. The set of articles containing “wood” has to be intersected with the set of articles containing “fire”. Intersections are implemented as iterators as well and (at least partially) performed on tablet servers in a distributed fashion. The joins in Rdfbox are intersections as well. Consider that a join on *?var* in Rdfbox is the intersection of each triple pattern’s bindings for *?var*.

Data Model and Index Operations

In order to use an intersecting iterator for the Accumulo index extension, the data model has to be changed because iterators do not work across different tables. The first step has been to map all Rdfbox indices into a single table and to find a key structure that supports distributed joins.

Each Accumulo table can be split into multiple *tablets* and each tablet is always entirely hosted by a single node. Section 2.5.2 contains more information. Iterators run in *tablet scope* and because tablets are split at row boundaries, all entries with the same row field value are in the same tablet. Therefore data locality can be considered when

designing the key-schema. All entries that need to be joined remotely, require the same row field value. Otherwise it can not be ensured that an iterator can see all of them. Further each entry's index permutation and level has to be stored in the key schema, to make it unique and to be able to retrieve entries by index or level.

Rdfbox index operations are based on *prefixes* of the entry identifiers (*idk*). The operations return triple element identifiers that are obtained by removing the *prefix* from the full *idk*. Let the remaining part be the *postfix*. Because the identifiers of triple elements are fixed-size byte arrays, the length of the *pre-* and *postfix* is always a multiple of the id size.

Consider an entry in an SPO index with an $idsize$ of 3 byte:

$$\text{spo0: } \underbrace{}_{prefix}\ \underbrace{101}_{postfix} \rightarrow count \qquad len(prefix) = \mathbf{0} * idsize;\ len(postfix) = \mathbf{1} * idsize$$

$$\text{spo1: } \underbrace{101}_{prefix}\ \underbrace{201}_{postfix} \rightarrow count \qquad len(prefix) = \mathbf{1} * idsize;\ len(postfix) = \mathbf{1} * idsize$$

$$\text{spo2: } \underbrace{101201}_{prefix}\ \underbrace{416}_{postfix} \rightarrow count \qquad len(prefix) = \mathbf{2} * idsize;\ len(postfix) = \mathbf{1} * idsize$$

A pattern emerges. The $prefix$ length is $level * idsize$ bytes and the $postfix$ length is $1 * idsize$ bytes. This separation of the *idk* is used in the new key schema that supports distributed joins:

$$(postfix,\ index\colon prefix,\ \text{“”},\ timestamp) \rightarrow count$$

Again the visibility field is empty and *timestamp* is not used. *index* is the concatenation of the index permutation and level (e.g. spo1). The row field value is $postfix$ and $index\colon prefix$ means that the column family is *index* and the column qualifier is $prefix$.

This key format guarantees that all entries with the same *postfix* are in the same tablet because *postfix* is used as the row. The column families of each row are the indices that it has entries for. The column qualifiers are the *prefixes* that the *postfix* appears with.

Rdfbox index operations are now performed by iterators. Consider that a cursor is specified by the index it is opened on and the *prefix* it jumped to. An Rdfbox join can be interpreted as resolving all *postfixes* that appear in the indices with the *prefixes* specified by the involved cursors. This information has been moved into the index key schema and therefore cursors are made obsolete. Distributed joins are specified by a set of columns (a column is specified by *index* and *prefix*). The *JoinIterator* is attached to a scanner with unlimited range. When the scanner is now used to iterate over entries, only rows (*postfixes*) that appear in *all* the specified columns are returned. Thus a join is performed.

This distributed join has a consequence. There is no ordered view of each index any more. The indices are now ordered by *postfix* which is not useful in the scope of Rdfbox. But the primary benefit of an ordered view, the ability to perform fast merge joins in the query execution engine, is made obsolete by distributed joins. Therefore this is not a concern.

Code Changes

Accumulo already provides an *IntersectingIterator*. An extension has been implemented that adapts it to work with the described key format. It is implemented in Java. A JAR file containing the iterator has to be distributed to all Accumulo nodes in order to use it. Some modifications had to be made to the client proxy because the index does not offer an ordered view of the entries any more. The *JoinIterator* is used together with a *BatchScanner*. *BatchScanners* perform parallel scans on multiple tablets but they return entries without predictable order. This variant of the client proxy is called *AccumuloUnorderedClient*. The client proxy service has to be started with the *unordered* parameter and in Python an *AccumuloUnorderedClient* instance has to be used for interaction. In Rdfbox, distributed joins can be activated with a compile flag but only Accumulo supports it.

4.2.5 Preliminary Results and Analysis

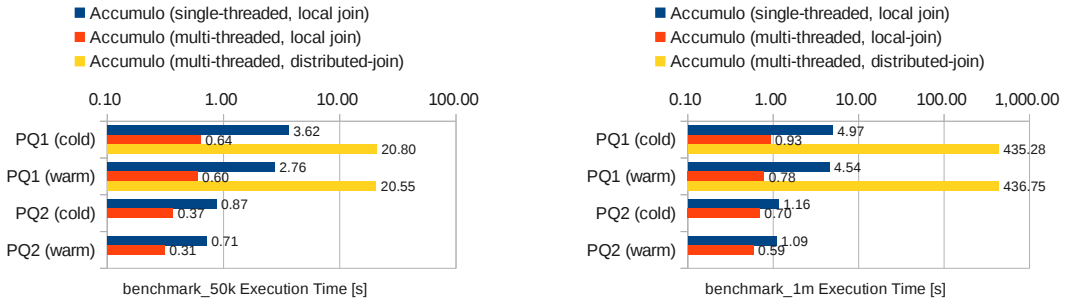

Figure 4.4: Preliminary evaluation query execution times for different Accumulo index implementations. Single-threaded and multi-threaded refer to the query execution engine variant used. The times are measured with (**left**) the benchmark_50k data set and (**right**) the benchmark_1m data set. Multi-threaded, local-join performs best, distributed joins perform very badly (*PQ2* evaluations with distributed joins were aborted after surpassing one hour of execution time). Mind the logarithmic scales.

Figure 4.4 compares the results of the preliminary evaluation setting for all Accumulo index variants. The setting is described in Appendix A.1. Comparing Accumulo to Hyperdex (Hyperdex evaluation results can be found in Section 4.1.2 and Figure 4.1) reveals that Accumulo outperforms Hyperdex by at least an order of magnitude in each case. This supports the decision to concentrate on Accumulo. But because the data sets are relatively small, it is more important that Accumulo scales dramatically better between 50,000 and one million triples. The best Hyperdex variant (MS), increases execution time between the two data sets by a factor of almost 19 and approximately 16 for *PQ1* and *PQ2* respectively. The same factors for the single-threaded Accumulo

variant (which uses the same query execution engine) are merely 1.4 and 1.5 and for the best Accumulo variant (multi-threaded, local join) 1.3 and 1.9. The data does not allow to make a final statement about how either of the indices scale but Accumulo has more potential to perform well with larger data sets.

Comparison of Single-Threaded and Multi-Threaded Query Resolution

Multi-threaded query resolution is an order of magnitude faster than single-threaded query resolution for every evaluated query and data set. Therefore the assumption that network I/O is the main bottleneck in single-threaded query resolution can be verified. The multi-threaded variant will be evaluated with larger data sets in Chapter 5.

Analysis of Distributed Joins

Tests confirmed that the distributed joins produce correct results but they are very slow. For *PQ1* it is comparable with Hyperdex backends but for *PQ2* it is much worse. The results for *PQ2* were not measured for either data set because the execution has been aborted after an hour of execution time. It is not entirely clear why these joins are so slow but some findings are:

- Even though several Accumulo examples use the approach of filtering entries by column, this seems to be significantly slower than access by row ranges.
- The index structure is effectively a reverse index, mapping all postfixes to the indices and prefixes they appear with. However the index is not accessed in that manner. All rows have to be filtered by column, resulting in each row being accessed. That this does not scale is not very surprising but it is not clear why the Wikipedia search example performs much better. Its evaluation has been run on a much stronger infrastructure however [23].

Perhaps there are some problems with the join algorithm implementation as well. But due to lack of time the decision has been made to leave the distributed joins in the current state and spend the remaining time optimizing the earlier approach that performs considerably better.

4.2.6 MapReduce-based Loading for Accumulo

The facilities provided by Rdfbox for loading are inefficient (Section 5.2.1) for distributed indices. This section describes how a more efficient distributed loader has been implemented.

The loading process as described in Section 2.4.5 consists of four steps: an input RDF graph has to be encoded, permutations of all triples have to be created, the input has to be sorted and then written to the index backend. The encoding step needs to be synchronized. But the other steps can be distributed.

MapReduce is a programming model for distributed computing designed to process vast amounts of data [17] and Apache Hadoop [24] implements an open source software

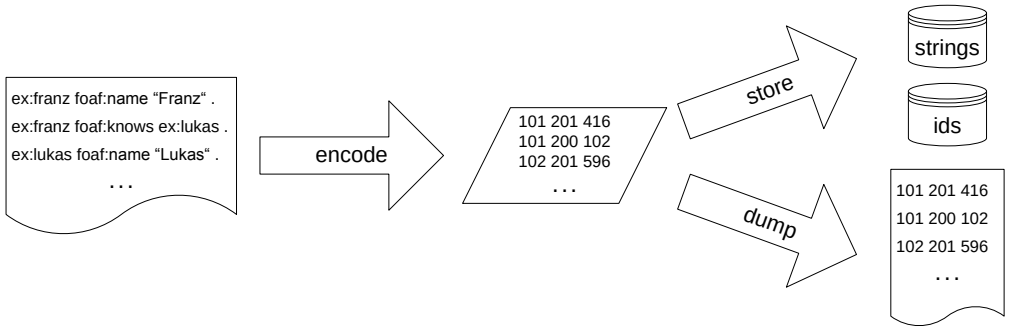

Figure 4.5: The *dumper* process. A serialised RDF graph is used as input. The graph is encoded, that means translated from string-space into identifier-space. The resulting string-to-identifier and identifier-to-string mappings are stored in the string manager of Rdfbox. The encoded triples are written (dumped) into a file.

framework for MapReduce. Accumulo runs on top of a Hadoop cluster and is designed to work as a data source and data sink of MapReduce jobs.

Separation of the Encoding Step

In order to decouple the encoding step from the other steps, a simple Python program has been created that uses Rdfbox to encode an input graph. It is called the *dumper* because it stores the encoded graph to a file instead of executing any of the other loading steps. Figure 4.5 illustrates this process. Rdfbox loader components are only used to create the string and identifier mappings in the string manager. If the input is larger than a specified size, the input is processed in chunks. This allows to encode RDF graphs that are larger than memory. Following the dumping process, the file with the encoded triples can be uploaded to a distributed file system (DFS) where MapReduce jobs can use it as input.

Distributed Loading

With the encoded graph saved in the DFS, a MapReduce job can be used to implement the remaining steps (permutation, sorting, writing to Accumulo table). Figure A.1 in Appendix A.2.2 illustrates this process. The map steps create the permutation and the reduce step sorts the permutations and aggregates the cardinality values. Because it uses both, the map and the reduce steps, it is called *Map/Reduce* loader. The output is written to Accumulo directly on DFS level, which is very efficient and makes this process considerably faster than conventional loading. But it does not distribute load very well. Because output is written sorted by index, the load is concentrated on one or two tables. The nodes hosting these tables are under heavy load while other nodes are idle. Sorting and aggregation are tasks that can be performed by Accumulo. Therefore an unsorted variant has been created. This *Map-only* loader writes the output of the map processes

directly into Accumulo. Figure A.2 in Appendix A.2.2 describes the process. Because each map process creates entries for each table, all tables and therefore all nodes are busy for the entire loading time. The load is distributed evenly. The lexicographic ordering of the entries is done automatically by Accumulo and a *SummingIterator* has been configured on the tables to aggregate the cardinality values.

Section 5.2.1 compares the distributed loaders to conventional loaders and Section 5.2.2 evaluates distributed loading in detail.

4.3 General Changes to Rdfbox

At the beginning of of this work the intention has been to work towards a release of Rdfbox. Therefore some time has been invested to make sure Rdfbox can be compiled on different platforms. The build scripts had to be repaired because some library names and locations were platform dependent and not all resources have been discovered for inclusion in distribution builds.

Further another effort to extend Rdfbox with more centralized index backends [39] has been developed in parallel on the same code base. This parallel work introduced many additional external dependencies which, in the absence of a useful dependency management solution, proved to be problematic for the development workflow and made it complicated to install Rdfbox for users. In general a user does not need all backends at the same time but uses only a single one or a small subset. Rdfbox featured a plugin configuration that primarily allowed to cut down on compilation time of Cython modules by choosing which modules to compile. But often static dependencies from plugin modules had to be introduced into core Rdfbox modules. That prohibited a successful compilation of Rdfbox without those plugins – even when they were never used at runtime. To resolve this problem, the plugin mechanism has been extended to enable exclusion of static dependencies at compile time depending on the current plugin configuration and to disable unit tests for backends that were not compiled.

5

Evaluation

This chapter starts with the presentation of the RDF data sets that are used for evaluations. Then the write and read operations with Hyperdex and Accumulo backends are evaluated before a discussion of the limitations of this evaluation concludes the chapter.

5.1 Data Sets

A mix of synthetic and real data sets of different sizes has been used for the evaluations. Table 5.1 lists the dataset sizes and their origin. Synthetic datasets are artificially generated by a computer program for evaluation purposes. Real data sets are taken from databases that are employed in real use cases.

Name	**Description**	**Number of Triples**
benchmark_50k	Synthetic product database	50,116
benchmark_1m	Synthetic product database	1,000,226
benchmark_25m	Synthetic product database	25,000,557
benchmark_100m	Synthetic product database	100,001,402
sp2bench	Synthetic bibliography	10,000,457
dbpedia-subset	Encyclopedia	43,600,000
life-science	Chemicals, Drugs	52,787,000

Table 5.1: Datasets used for the evaluations. The benchmark datasets originate from the Berlin SPARQL Benchmark [13]. The other data sets are part of the FedBench datasets [41]. The life-science data set includes the dbpedia-subset.

5.2 Writing Triples

The required steps for loading RDF graphs are described in Section 2.4.5. They can be separated into pre-processing steps (encoding, permuting triples and sorting) which are performed by the Rdfbox Loader module and the effective writing of entries into the distributed index. This sections shows that the conventional loading approach in Rdfbox is inefficient for distributed indices and evaluates a MapReduce [17] based distributed loading process.

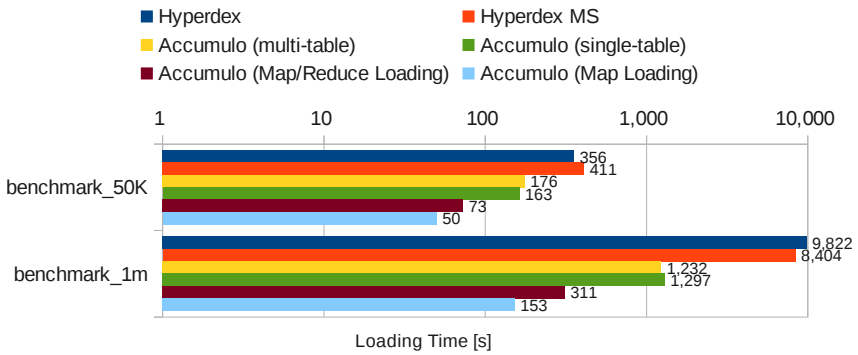

Figure 5.1: Loading times with different loader variants. *Hyperdex* refers to the single-space implementation and *Hyperdex MS* to the multi-space implementation. To be comparable with the time measurements of conventional loaders, the values for the distributed loaders (*Map/Reduce* and *Map*) consist of encoding and loading times added together. All Accumulo variants perform significantly better than Hyperdex variants and distributed variants are an order of magnitude faster than the corresponding conventional loaders. Mind the logarithmic scale.

5.2.1 Conventional Loading

Conventional loading refers the default loading process in Rdfbox as described in Section 2.4.5. Compared to distributed loading, all steps are performed on a single machine and the index entries are written via the *put* interface. Both, the Hyperdex and the Accumulo backend, can be used with conventional loaders.

Figure 5.1 contains a comparison of conventional loading times of all implemented index variants (the distributed variants are listed as well). All data is taken from the preliminary evaluations (Appendix A.1.1 and A.1.2). Conventional Accumulo variants (multi-table and single-table) generally perform much better than Hyperdex variants. Specifically this comparison reveals that the Hyperdex variants scale very badly with a scaling factor between 20 and 28 (that is the loading time for benchmark_1m divided by the loading time for benchmark_50k). Between the conventional Accumulo loaders, no significant difference can be seen and scaling factors are between 7 and 8.

The conventional loading process achieves relatively low throughput because the *put* interface is not very suitable for bulk loading of data as entries are written one at a time. With distributed indices this causes significant network overhead for each entry. This is apparent in Figure 5.1 where the distributed loaders can be seen to outperform the conventional loaders by an order of magnitude. The distributed loaders use a better interface for bulk loading.

5.2.2 Distributed Loading

A MapReduce based distributed loading process has only been implemented for the Accumulo index extension because the Hyperdex extensions performed badly, even with small data sets (Section 5.2.1).

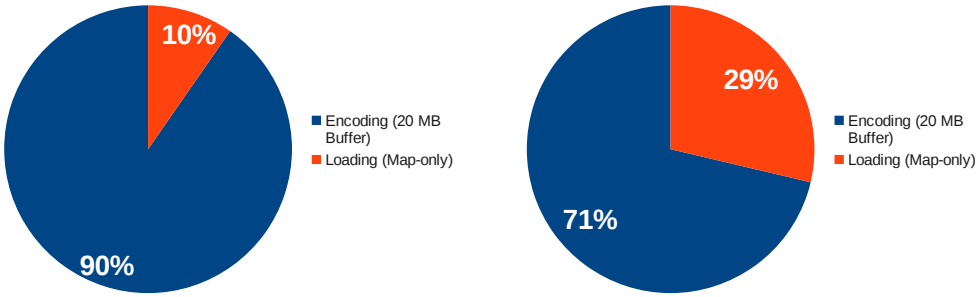

Figure 5.2: Encoding time compared to distributed loading time. (**left**) dbpedia-subset (43.6 million triples) spends a bigger fraction of time on encoding than (**right**) sp2bench dataset (10 million triples). Encoding becomes a bottleneck for large data sets.

The encoding is not distributed (Section 2.4.5). Therefore the *encoding phase* is evaluated separately from the other steps which are combined into the *loading phase*. Ritter has shown that a major bottleneck of the overall writing process chain is the encoding step [39]. When encoding approximately 150 million triples, 90% of the time has been used by the encoder process. A similar bottleneck can be identified for distributed loading (Figure 5.2). The larger the dataset is, the more the writing process is dominated by encoding time.

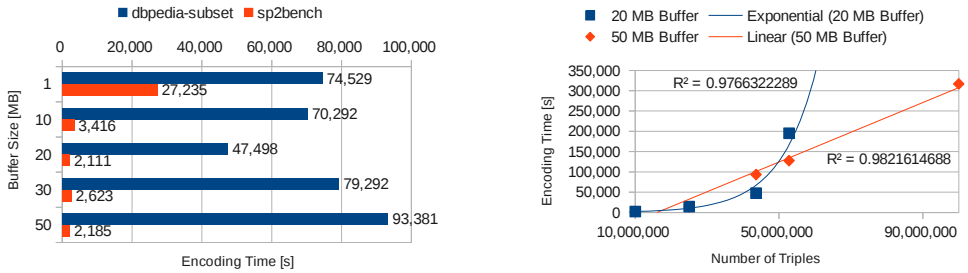

Figure 5.3: (**left**) Encoding times of the dbpedia-subset (43.6 million triples) and sp2bench (10 million triples) for different buffer sizes. Buffer size can significantly influence encoding time. (**right**) Encoding time by data set size. 20 megabyte buffer performs better until approximately 50 million triples. Then it appears to scale exponentially. From 50 to 100 million triples 50 megabyte buffer is better.

Encoding Phase

A simple program for the encoding of RDF data and the creation of the string-to-identifier mappings has been created (Section 4.2.6). The data sets (Section 5.1) were encoded on a computer with two 3.16 GHz CPU cores, 2 GB memory, running Ubuntu Server 12.04 64bit Edition and a 7200 RMP hard drive.

In order to encode data sets larger than memory, the input is split into chunks that are processed individually. The size of the chunks depends on the configuration of the buffer size for the encoder. Setting large buffer sizes (>100 megabyte) failed to encode data sets bigger than one million triples. The process ran out of memory. Furthermore the buffer size influences the encoding time dramatically (Figure 5.3). A buffer size of 20 megabytes is optimal for data sets smaller than 50 million triples. Figure 5.4 shows the relative time spent on sub-tasks of the encoding phase. The entire phase is dominated by the encoding sub-task, that is: dictionary encoding and writing of the string-to-identifier mappings. With a 20 megabyte buffer size the encoding sub-task is most efficient but this effect is less obvious for smaller data sets.

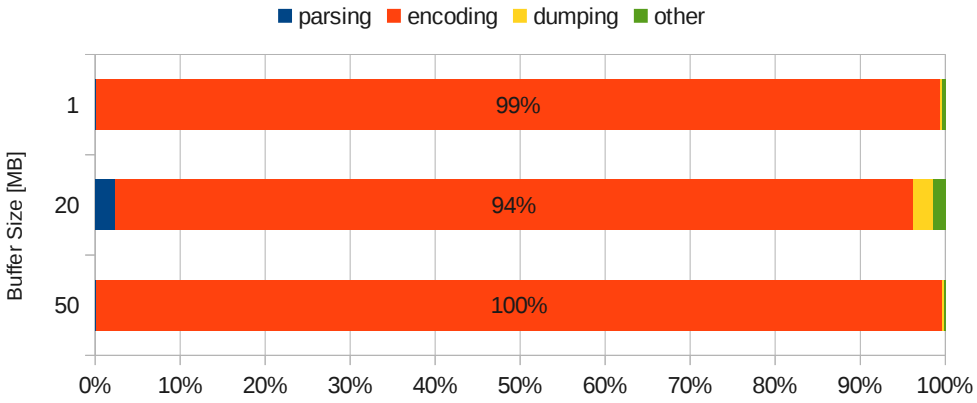

Figure 5.4: Relative time spent on encoding sub-tasks for the dbpedia-subset. Parsing is the reading and interpreting of input data. Encoding includes the dictionary encoding and the creation of the string-to-identifier mappings. Dumping refers to the writing of encoded triples into a file for later processing. Other is time spent by the algorithm on other tasks such as gathering of statistics and splitting the input into chunks.

Based on the data plotted in Figure 5.3, it is appears that the encoding time scales exponentially with data set size for a 20 megabyte buffer ($R^2 = 0.9766$). The data for the 50 megabyte buffer indicates linear growth ($R^2 = 0.9622$) but based on attempts to encode a larger data set (150 million triples – it did not finish) it has to be expected that the 50 megabyte buffer eventually reveals exponential growth as well.

The Role of Main Memory [39] used almost the same encoding process as part of the full loading process and managed to encode and load the same datasets in significantly

less time. Encoding and loading of the FedBench [41] cross-domain data set lasted approximately 34,000 seconds (9.4 hours). Out of which 90% is encoding time. Compare this to the encoding of 100 million triples in this work that took approximately 316,800 seconds (88 hours). The main difference between the two encodings is the computer they ran on. [39] had access to a machine with eight 2.933 GHz CPU cores and 72 GB main memory. Because encoding is not parallelised (Section 2.4.5), most of the difference has to be attributed to the difference in main memory. A hypothesis is that optimal buffer sizes and the efficiency of the encoding process depends strongly on the system's main memory. Better hardware infrastructure and further evaluations are necessary to verify this.

Loading Phase

For distributed loading a relatively small MapReduce and Accumulo cluster has been deployed on four nodes. Each node had two 3.00 GHz CPU cores, 2 GB memory, a 7200 RPM hard drive and a Ubuntu Server 10.04 32bit Edition operating system. Hadoop DataNodes and TaskTrackers were deployed on each node. One node has been designated as the master node and additionally ran a NameNode and JobTracker. The other nodes have been designated slave nodes. Accumulo has been set up in the same way. All nodes ran a TServer and Logger process. The master node additionally ran a Garbage Collector, Monitor and Master process. The master node also hosted the Zookeeper server. Furthermore MapReduce was configured to run maximally one map process and one reduce process on each slave and only one reduce process (no map processes) on the master. This relieved the master of some workload and ensured one CPU core would be available for Accumulo during the execution of jobs.

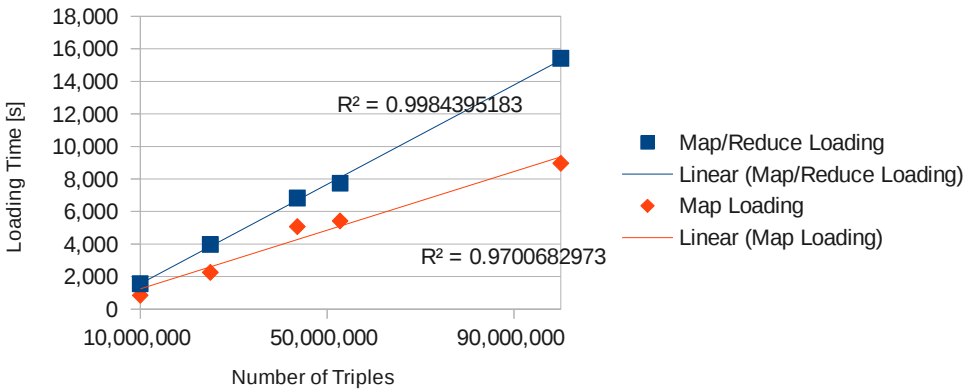

Figure 5.5: Distributed loading time by data set size for both loading variants. Loading time grows linearly with dataset size and *Map-only* loading scales better than *Map/Reduce* loading.

Two variants of distributed loading (Section 4.2.6) exist. *Map/Reduce* loading uses map and reduce steps. The reduce step sorts the input and performs cardinality aggregation. The second variant, *Map-only* loading, omits the reduce step and writes entries into Accumulo after the map step. Accumulo then performs the ordering and cardinality aggregation in a deferred manner. Figure 5.5 plots loading times for both variants by data set size. The data suggests that Map/Reduce loading scales linearly ($R^2 = 0.9985$). Linear is the best fit for the *Map-only* variant too ($R^2 = 0.9701$). But neither evaluation contains enough data points to make a final statement. In general *Map-only* loading scales better than *Map/Reduce* loading because each map task writes entries into all Accumulo tables (Section 4.2.6 and Figure A.2). Consequently all tables are ingesting new entries in parallel. Load from additional map tasks is spread evenly across all Accumulo nodes. This leads to higher throughput overall. In the *Map/Reduce* case, only one or two tables are ingesting data at a time because entries are written in sorted order (Section 4.2.6 and Figure A.1). Therefore load from additional reduce tasks is concentrated on one or two nodes that may become overwhelmed, leading to lower throughput. One aspect has not been investigated in detail. Map-only loading places more burden on Accumulo until all entries are eventually aggregated. This may have a negative effect on query performance for a limited time but all queries can be answered nevertheless.

5.3 Query Execution Times

The query performance of the Hyperdex backends has already been discussed in Section 4.1.2. Section 4.2.5 covered the comparison of the different Accumulo variants. This section evaluates the best Accumulo variant with large data sets. The best variant is multi-threaded query resolution combined with local (as opposed to distributed) joins.

The infrastructure for these evaluations is the same as for the distributed loading evaluation (Section 5.2.2). Namely a setup with four Hadoop/Accumulo nodes. One node additionally hosts master processes and the Zookeeper server. Rdfbox and the client proxy shared a dedicated machine with two 3.16 GHz CPU cores, 2 GB memory, 7200 RMP hard drive and Ubutu Linux Server 12.04 64 bit edition operating system. The data sets are described in Section 5.1 and the queries in Appendix A.1 and A.2.3. All execution times in this section are averaged over five runs, unless differently indicated.

5.3.1 Preliminary Evaluation Queries

The preliminary evaluation queries were chosen because they exercise different aspects of query resolution. *PQ1* (Listing A.1) focuses on data transfer between the remote indices and the local Rdfbox instance. The joins involved are very simple to process. The data for *PQ1* in Figure 5.6 indicates a logarithmic growth in execution time ($R^2 = 0.9186$) but the data is not sufficient to make a final statement. *PQ2* (Listing A.2) involves joins of a higher complexity and is intended to evaluate the query resolution algorithm.

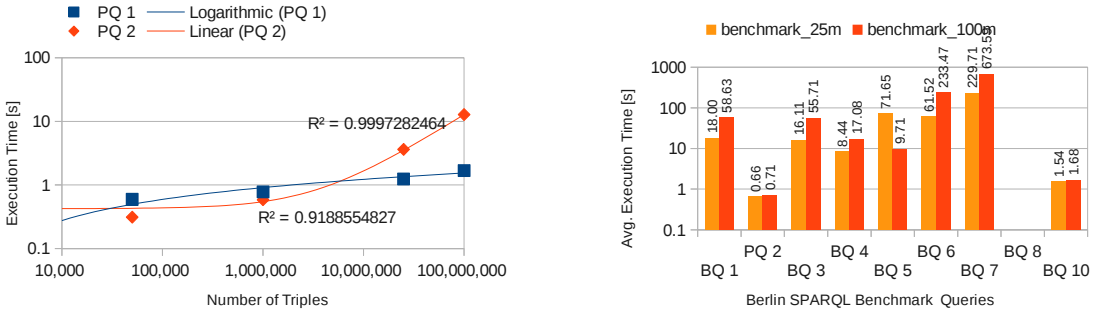

Figure 5.6: (**left**) Preliminary evaluation query performance with different data set sizes. *PQ1* and *PQ2* plotted for data set sizes from 50,000 triples to 100 million triples on a *log-log scale*. *PQ1* starts off as the slower query but appears to scale logarithmically. *PQ2* appears to scale linearly. (**right**) Average query execution times for the the benchmark_25m and benchmark_100m data sets. The time scale is logarithmic. Missing bars indicate that the query did not finish (typically they were aborted after several hours of execution).

With growing data set size, the execution time appears to grow linearly (Figure 5.6, $R^2 = 0.9997$).

5.3.2 Bernlin SPARQL Benchmark Queries

The two data sets benchmark_25m (25 million triples) and benchmark_100m (100 million triples) are used to evaluate how queries scale with data set size. The full queries are available in Listing A.5. The execution times are shown in Figure 5.6. *BQ 7* puts a lot of stress on the Accumulo infrastructure. It reports a scanning rate of over 40,000 entries/s and appears to reach a limits. Accumulo logs warnings about several nodes running low on memory. It would be interesting to test this query on a more powerful infrastructure, to determine the impact of the limited resources. *BQ 8* on the other hand exhibits an extremely low scanning rate (1000 entries/s) and failed to produce results in meaningful time. It is not known why the low scanning rate happens. *BQ 5* is an anomaly. It executes an order of magnitude faster with 100 million triples than with 25 million triples. The measurement has repeatedly verified with no explanation found.

In general it can be observed that the queries scale less than by a factor of 4, which is the factor the data set is scaled with. The very simple queries *PQ 2* and *BQ 10* scale with a much lower factor. These observations are in line with the analysis of the scaling behaviour conducted with the preliminary queries (Section 5.3.1).

5.3.3 Life Science Queries

The life-science data set contains 52.8 million triples and is therefore a medium-sized data set in this evaluation. The full queries are available in Listing A.3. Figure 5.7 compares

the execution times of all life-science queries. The performance of half of the queries is in the range where it can be used for responsive applications. *LQ 3* and *LQ 5* approach the one minute mark which is clearly too much for responsive applications but still useful for batch processing. *LQ 7* did not produce a single result after several hours of execution although it did put considerable stress on the infrastructure. Accumulo reported a scanning rate of approximately 25,000 entries per second for the entire execution time which indicates that the query execution engine did not find an efficient execution plan. Further investigations show almost all data transfer happened on the pos1 index and that the it contains many rows with identical prefix. It is likely that one of the involved joins repeatedly transferred the same, very large range of data, indicating a very *unselective* join. The time for *LQ 9* is the time to retrieve just one result. Most load was distributed on the indices pos1, pso1 and pos2 and the scanning rate fluctuated constantly between transfer 15,000 and 20,000 entries per second. It appears to suffer from similar problems as *LQ 7*.

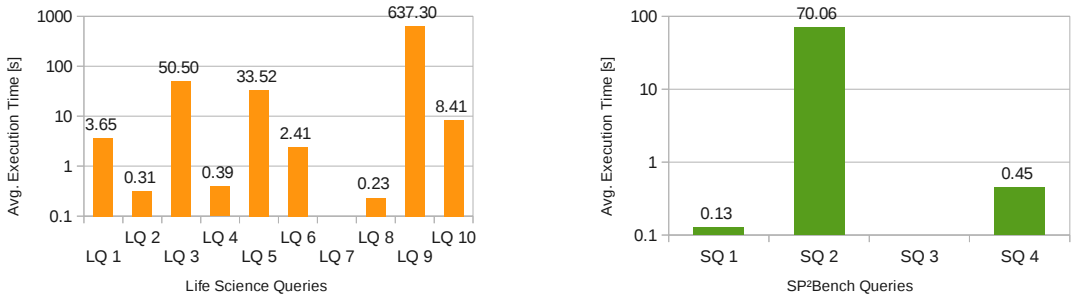

Figure 5.7: Average query execution times for the (**left**) life-science and (**right**) sp2bench data sets. The time scale is logarithmic. Queries with missing bars did not finish (typically they were aborted after several hours of execution).

5.3.4 SP^{2}Bench Queries

The sp2bench data set contains 10 million triples which makes it a relatively small data set in this evaluation. The queries are available in Listing A.3. *SQ 1* and *SQ 4* perform well but they are rather simple queries. Both complex queries perform significantly worse. *SQ 2* brings the Accumulo infrastructure to its limits with a scanning rate over 60,000 entries/s. *SQ 3* had to be aborted before it produced any results. A first analysis suggests that for some reason it exhausted the memory on the machine that runs Rdfbox which degrades the performance until it stalls completely. This may be due to a bug in the execution engine but more investigations have to be conducted to verify this.

5.4 Query Execution Time Analysis

Query execution produced mixed results. An observed effect is that individual joins need to transfer more data when operating on larger data sets. Therefore the size of intermediate results is growing as well – especially when the joins are *unselective*. Each intermediate result creates a context which typically requires joins as well and adds even more contexts to be processed – until it is fully resolved (Section 4.2.3). An assumption is that this leads to an exponential increase of the number of join operations with larger data set sizes and consequently the total size of data that has to be transferred to resolve a query may scale exponentially as well. In conclusion, queries with many *unselective* joins are expected to be very problematic with large data sets. *LQ 7*, *LQ 9* and *SQ 2* appear to be such cases.

In general it is evident that the current query resolution is not efficient enough for complex queries with many joins. Future work may investigate the precise causes of these inefficiencies and implement further optimisations to the query execution engine. Unfortunately the algorithmic complexity of SPARQL queries can be very high [36] and there is no easy solution that would avoiding this. But it has to be considered that the transfer of data on the network interface is a bottleneck that scales directly with the size of the data set. Therefore transferring the data to the algorithm will subject execution times directly to the data set size. If a distributed resolution algorithm could be created, and most of the data transfer avoided, the influence of the data set size may be reduced considerably. If the algorithm is efficient, it can eventually outperform the current centralized algorithm.

5.5 Evaluation Limits

Not enough data could be gathered to make any definitive statements about the behaviour of the presented solutions with very large data sets. Consequently one of the goals of this thesis can not be fully evaluated. Namely whether the presented solution can handle data sets larger than those that can be used with centralized index backends. This has several reasons. On one hand a more powerful infrastructure is required to be able to evaluate larger data sets. On the other hand, the encoder process and possibly parts of the multi-threaded query resolution engine can be implemented more efficiently, which would lower the infrastructure requirements in general.

Further, a different limitation is that these results can not be directly compared to other implementations even though data sets and queries from established benchmarks [13, 41] have been used. Most benchmark queries are not fully supported by the Rdfbox query execution engine. Therefore they have been modified to avoid unsupported features. Sometimes this may alter the query characteristics significantly.

6

Related Work

This chapter describes two works closely related to the Accumulo index extension. Both of these related works are based on a distributed BigTable [14] based storage layer. Table 6.1 contains a summary of the comparisons that are explained in more detail in the following sections.

6.1 Rya – Scalable Triple Store implemented on Accumulo

The Rya RDF Triple Store is an RDF storage and querying layer for Accumulo [38]. It uses three indices (SPO, OSP and POS) to save triple elements as concatenated strings in the row field of an Accumulo key. The column field is not used but the visibility field provides access control to triples. The value field is empty. This yields the following data model for the case of an SPO index. OSP and POS are modelled alike:

(spo, “”, securityFlag, timestamp) $\rightarrow$ -

When querying data, triple patterns are reordered such that a maximum prefix of bound values matches one of the three indices. Example: $(a, ?b, c)$ would have a maximum matching prefix in the OSP index when reordered to $(c, a, ?b)$. Because Accumulo sorts all entries by row, sequential access to rows sharing a common prefix is very efficient. This concept is very similar to Rdfbox' sorted indices. Rya uses a special statistics

	Rya	**CumulusRDF**	**Rdfbox**
BigTable Implementation	Accumulo	Cassandra	Accumulo
Data Distribution	DFS (HDFS)	DHT	DFS (HDFS)
Permutations	3	3	6
Permutation Structure	none	columns	levels
Cardinalities	triple elements	none	rich*
Query Support	SPARQL	triple patterns	SPARQL subset

Table 6.1: Comparison of BigTable-like distributed RDF stores. * Section 2.4.2 describes the rich cardinalities stored in Rdfbox.

table to store cardinality values for triple elements. These statistics are are created with periodic MapReduce jobs.

The Rya RDF store provided very valuable insight for the implementation in this thesis. But it deviates from Hexastore-based implementations in various points:

- Rya does not use dictionary encoding on triple elements but saves the string representations directly in Accumulo.
- It only uses three indices and not all six possible permutations. It does not use index levels like Rdfbox or any deeper structures in indices for that matter. Each entry is a full triple.
- According to [38] joins during query execution are not merge joins in Rya. A future extension that uses merge joins for better performance is mentioned however. But when using only three index permutations and no index levels, merge joins will not be possible in all cases.
- Rya plugs into the OpenRDF Sesame [3] SAIL API to implement storage and querying of RDF data. Therefore it delegates query execution plan generation to SAIL. Rdfbox implements a custom query execution engine. Based on [38] it appears that queries are primarily resolved based on triple patterns, that means that triple pattern in a graph pattern are individually resolved and the intermediate results then joined. This may be one reason why levels are not used in Rya. Section 4.1.1 briefly mentioned that such a resolution approach is dangerous because it limits the options to cut down on the size of intermediate results.
- In order to optimise query execution plans, an extra table with cardinalities is used. In Rdfbox, more fine grained statistics are used as an integral part of the indexing scheme (the statistics contained Rdfbox' level 1 indices are completely missing in Rya).

6.2 CumulusRDF – RDF Storage in the Cloud

CumulusRDF [32] is another triple store based on a BigTable [14] storage solution. It uses Apache Cassandra [33] as storage layer. Cassandra offers the same data model as BigTable (Section 2.5.2). But it uses a distributed hash table for distributed data storage. CumulusRDF provides a REST API for data management and allows to formulate single triple patterns as queries but it does not provide a SPARQL endpoint. It creates the same three indices SPO, OSP and POS as Rya (Section 6.1) and maps any query type to the corresponding index, reordering the triple elements in the request if necessary. Unlike Rya, it uses the row and the column key fields to create a structure reminiscent of Rdfbox index levels. Nothing is mentioned about the timestamp field. For an SPO entry the data format is the following:

$$(s,\ p{:}o,\ timestamp) \rightarrow -$$

The OSP index is structured alike. Because RDF data typically has a skewed distribution and many triples share a relatively small set of predicates, the POS index would have very large rows (many entries with the same row field) if stored like the SPO/OSP indices. But Cassandra does not support rows larger than memory. Therefore the POS index has a different format:

$$(po,\ s{:}\text{-},\ timestamp) \rightarrow \text{-}$$

An alternative storage schema is available. It uses Cassandra specific *super-columns* to save the second-level element and the regular *column* for the third-level element.

Compared to the Rdfbox Accumulo plugin, the following differences should be considered:

- CumulusRDF saves the RDF string representation and does not encode triple elements using a dictionary encoding.
- Only three indices are used. Index levels are modelled in the BigTable key structure. Rdfbox uses six indices and models index levels as separate databases. Although this work tried to use structured entries for modelling levels as well, that solution was abandoned for performance reasons.
- The query API is very limited and only resolves single triple patterns. Triple patterns can be resolved with one index lookup like in a Hexastore. But CumulusRDF does not do any joins between triple patterns to resolve entire graph patterns. Therefore it does not solve one of the hard problems of query resolution.
- CumulusRDF does not save statistics about cardinalities in the index. This may be because it does not join intermediate results or resolve any complex queries.

7

Conclusions and Future Work

With RDF data sets reaching the size of billions of triples and the Semantic Web continuing to grow at a fast pace, the resources of a single machine can easily become overwhelmed. The recent trend towards cloud computing and the increasing offer of infrastructure as a service (IaaS) providers, have made access to clusters of commodity-level hardware easy and economical. Therefore distributed databases have become a viable option for many new use cases.

This thesis focused on the implementation of two distributed index extensions and a distributed loading mechanism for the Rdfbox semantic data base management system. Previously only centralized index structures were used by Rdfbox. Hyperdex [20] and Apache Accumulo [22] were chosen as the distributed persistent storage layer for the distributed index structures.

During development it became clear that Hyperdex would not be a viable solution. An extensive investigation of the underlying problems revealed that its search API is not suitable for the access pattern imposed by Rdfbox. Despite various optimization attempts, satisfactory performance could not be achieved.

Accumulo however, produced more promising results. It is based on Google's BigTable [14] and designed for vast amounts of data. It builds on top of the proven and widely used Hadoop [24] framework to achieve horizontal scalability and persistent storage. Early evaluations proved that it scales much better with increasing data set sizes than Hyperdex and its API supports the Rdfbox use case efficiently.

Some core modules in Rdfbox had to undergo extensive optimisations to enable more efficient query resolution with distributed data stores that are inherently slow to access. From this arose one of the main contributions of this thesis, a scalable and efficient multi-threaded query execution engine.

Building on the new query execution engine, the Accumulo index plugin could unfold more of its potential. A thorough evaluation produced mixed results however. For medium sized data sets in the range of tens of millions of triples many indicators show that the plugin can work efficiently and the scaling behaviour of the index operations is adequate. But hard to overcome limits are imposed on the solution by the algorithmic complexity of SPARQL query resolution [36] and the circumstance of having to fetch data from remote locations to join them locally. This work attempted to distribute parts of the query resolution process to the remote nodes where the data is stored but this partially sacrificed the efficiency of the Hexastore index scheme and performed

significantly worse. Consequently an index extension scalable to very large RDF graphs could not be contributed. But many insights have been gained into the consequences of using a distributed index for Rdfbox. One important conclusion is that the centralized query resolution algorithms have severe limits when used together with distributed data. A proposition is, that parts of the query resolution have to be distributed such that they can execute where the data is stored, as opposed to transferring the data to where the algorithm executes.

For the purpose of writing triples into the Accumulo backend, a distributed loading mechanism has been implemented. The encoding bottleneck, something that has been identified by previous work as well [39], could not be solved in the scope of this thesis. But for the purpose of populating the index backend with entries, a highly scalable and efficient solution could be created. The distributed loader is implemented as a MapReduce [17] job and makes use of a high speed bulk-ingest interface of Accumulo. Evaluations have shown that this loader performs an order of magnitude better for the distributed index than the previous loaders available in Rdfbox.

7.1 Future Work

A pressing matter is the creation of a more efficient encoding process. The current process is a severe bottleneck in Rdfbox because its current implementation can not be fully distributed and it appears to scale exponentially. It is the most time consuming task when loading large data sets (Section 5.2.2) and this bottleneck affects not only evaluations but the usefulness of a distributed index in general.

As mentioned in Section 5.5, more evaluations need be performed on a better infrastructure to verify or discard some of the findings and to establish the behaviour of the Accumulo backend with truly large data sets.

The multi-threaded query execution engine described in Section 4.2.3 still suffers from several insufficiencies. In some cases it has been observed to consume considerable amounts of memory. This has to be optimised. Further, a bug that existed already prior to this work has been found in the selectivity estimation. An investigation is necessary to establish whether this affects query resolution performance significantly. Using wrong cardinalities when evaluating a query execution plan, could have severe effects on performance. Section 5.3.3 describes that some queries may be transferring very large ranges of Accumulo entries repeatedly. Sophisticated local caching could mitigate this problem and give the Accumulo index a considerable performance boost.

But all these measures may not resolve the underlying problem. In order to provide efficient query execution on RDF graphs, a different approach should be investigated. The distributed join implementation in this work performed badly (Section 4.2.4). Nevertheless further investigations in that direction should be conducted. Section 5.4 discussed the problem of centralized query resolution and proposes that a distributed approach may overcome the encountered limitations.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd international conference on Very large data bases*, pages 411–422. VLDB Endowment, 2007.

[2] Ben Adida, Mark Birbeck, Shane McCarron, and Steven Pemberton. RDFa in XHTML: Syntax and processing. *W3C Recommendation*, 2008.

[3] Aduna. openRDF.org website. *http://www.openrdf.org/*, 1997. [Online; accessed 24-March-2013].

[4] Marcelo Arenas, Alexandre Bertail, Eric Prud'hommeaux, and Juan Sequeda. A direct mapping of relational data to RDF. W3C recommendation, World Wide Web Consortium, 2012.

[5] Sören Auer, Sebastian Dietzold, Jens Lehmann, Sebastian Hellmann, and David Aumueller. Triplify light-weight linked data publication from relational databases. 2009.

[6] Dave Beckett. Rasqal RDF query library. *http://librdf.org/rasqal/*, 2003. [Online; accessed 24-March-2013].

[7] David Beckett. The design and implementation of the redland RDF application framework. *Computer Networks*, 39(5):577–588, 2002.

[8] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax. Technical report, W3C, January 2008.

[9] Tim Berners-Lee, Roy Thomas Fielding, and Larry Masinter. Uniform resource identifier (uri): Generic syntax, September 2004.

[10] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[11] Christian Bizer and Richard Cyganiak. D2r server-publishing relational databases on the semantic web. In *5th international Semantic Web conference*, page 26, 2006.

[12] Christian Bizer and Andreas Schultz. Benchmarking the performance of storage systems that expose SPARQL endpoints. In *Proc. 4 th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS)*, 2008.

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

[14] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):4, 2008.

[15] Souripriya Das, Seema Sundara, and Cyganiak Richard. R2RML: RDB to RDF mapping language. W3C recommendation, World Wide Web Consortium, 2012.

[16] Jeff Dean and Sanjay Ghemawat. leveldb – a fast and lightweight key/value database library by google. *https:// code.google.com/ p/ leveldb/*. [Online; accessed 24-March-2013].

[17] Jeffrey Dean and Sanjay Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[18] Leigh Dodds and Ian Davis. Linked data patterns, 2012.

[19] Orri Erling and Ivan Mikhailov. RDF support in the virtuoso DBMS. *Networked Knowledge-Networked Media*, pages 7–24, 2009.

[20] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: a distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012.

[21] The Apache Software Foundation. Apache zookeeper website. *http:// zookeeper. apache.org/*, 2010. [Online; accessed 25-March-2013].

[22] The Apache Software Foundation. Apache accumulo website. *http:// accumulo. apache.org/*, 2011. [Online; accessed 22-March-2013].

[23] The Apache Software Foundation. The wikipedia example explained, with performance numbers. *http:// accumulo.apache.org/ example/ wikisearch.html*, 2011. [Online; accessed 28-March-2013].

[24] The Apache Software Foundation. Apache hadoop website. *http:// hadoop.apache. org/*, 2012. [Online; accessed 24-March-2013].

[25] Ramanathan V. Guha and Dan Brickley. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[26] Stephen Harris and Dr. Nicholas Gibbins. 3store: Efficient bulk RDF storage. 2003.

[27] Tom Heath and Christian Bizer. *Linked Data: Evolving the Web into a Global Data Space*. Morgan & Claypool, 1st edition, 2011.

[28] Matthias Hert, Gerald Reif, and Harald Gall. A comparison of RDB-to-RDF mapping languages. In *Proceedings of the 7th International Conference on Semantic Systems (I-Semantics)*, Graz, Austria, September 2011.

[29] Adobe Systems Incorporated. Extensible metadata platform (XMP) website. *http://www.adobe.com/products/xmp/*, 2013. [Online; accessed 24-March-2013].

[30] Graham Klyne and Jeremy J. Carroll. Resource description framework (RDF): concepts and abstract syntax. W3C recommendation, World Wide Web Consortium, 2004.

[31] FAL Labs. Tokyo cabinet: a modern implementation of DBM. *http://fallabs.com/tokyocabinet/*, 2006. [Online; accessed 24-March-2013].

[32] Günter Ladwig and Andreas Harth. CumulusRDF: Linked data management on nested key-value stores. In *Proceedings of the 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS2011) at the 10th International Semantic Web Conference (ISWC2011)*, October 2011.

[33] Avinash Lakshman and Prashant Malik. Cassandra: A structured storage system on a P2P network. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 47–47. ACM, 2009.

[34] Frank Manola and Eric Miller. RDF primer. *W3C Recommendation*, 10:1–107, 2004.

[35] Akiyoshi Matono, Toshiyuki Amagasa, Masatoshi Yoshikawa, and Shunsuke Uemura. A path-based relational RDF database. In *Proceedings of the 16th Australasian database conference-Volume 39*, pages 95–103. Australian Computer Society, Inc., 2005.

[36] Jorge Pérez, Marcelo Arenas, and Claudio Gutierrez. Semantics and complexity of SPARQL. *The Semantic Web-ISWC 2006*, pages 30–43, 2006.

[37] Eric Prud'hommeaux and Andy Seaborne. SPARQL query language for RDF. W3c recommendation, World Wide Web Consortium, 2008.

[38] Roshan Punnoose, Adina Crainiceanu, and David Rapp. Rya: a scalable RDF triple store for the clouds. In *Proceedings of the 1st International Workshop on Cloud Intelligence*, page 4. ACM, 2012.

[39] Thomas Ritter. Extending rdfbox with centralized RDF management. efficient RDF indexing and loading. Master's thesis, 2013.

[40] Jack Rusher. Rethorical device: Triple store. *http://www.w3.org/2001/sw/Europe/events/20031113-storage/positions/rusher.html*, 2003. [Online; accessed 22-March-2013].

[41] Michael Schmidt, Olaf Görlitz, Peter Haase, Günter Ladwig, Andreas Schwarte, and Thanh Tran. Fedbench: A benchmark suite for federated semantic data query processing. *The Semantic Web–ISWC 2011*, pages 585–600, 2011.

[42] Guus Schreiber and Mike Dean. OWL web ontology language reference. W3C recommendation, W3C, 2004. http://www.w3.org/TR/2004/REC-owl-ref-20040210/.

[43] Mark Slee, Aditya Agarwal, and Marc Kwiatkowski. Thrift: Scalable cross-language services implementation. *Facebook, Jan*, 2007.

[44] Cathrin Weiss and Abraham Bernstein. On-disk storage techniques for semantic web data – are b-trees always the optimal solution? In *The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems*, page 49, 2009.

[45] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: sextuple indexing for semantic web data management. *PVLDB*, 1(1):1008–1019, 2008.

[46] Kevin Wilkinson. Jena property table implementation. In *Second International Workshop on Scalable Semantic Web Knowledge Base Systems*. Hewlett-Packard Development Company, L.P, Citeseer, November 2006.

A

Appendix

A.1 Preliminary Evaluations

For preliminary evaluations during the development phase small data (benchmark_50k and benchmark_1m, see Table A.4) sets have been used to test for functionality and perform preliminary performance evaluations.

The queries are of two categories to check for different aspects of the implementation. Unselective queries like PQ1 shown in listing A.1, cover many entries in the index backends and require a lot of data to be transferred between Rdfbox and index backends. These queries can point out bottlenecks in the communication between Rdfbox and the distributed indices. Selective queries like PQ2 shown in listing A.2 retrieve various information for a specific node. These queries primarily perform joins and can reveal problems with the join operation. Note that *:product:* has to be replaced by a specific product URI when executing.

Listing A.1: PQ1 – An unselective query focusing on data transfer

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?s ?o WHERE {
        ?s rdfs:label ?o.
}
LIMIT 300
```

Listing A.2: PQ2 – A selective query focusing on joins

```
PREFIX b: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1
       ?propertyTextual2 ?propertyTextual3 ?propertyNumeric1
       ?propertyNumeric2
WHERE {
         :product: rdfs:label ?label .
         :product: rdfs:comment ?comment .
         :product: b:producer ?p .
         ?p rdfs:label ?producer .
         :product: dc:publisher ?p .
         :product: b:productFeature ?f .
         ?f rdfs:label ?productFeature .
         :product: b:productPropertyTextual1 ?propertyTextual1 .
         :product: b:productPropertyTextual2 ?propertyTextual2 .
         :product: b:productPropertyTextual3 ?propertyTextual3 .
         :product: b:productPropertyNumeric1 ?propertyNumeric1 .
         :product: b:productPropertyNumeric2 ?propertyNumeric2 .
}
```

A.1.1 Preliminary Evaluation of Hyperdex Backend

Table A.1 lists the results for the preliminary evaluation of Hyperdex index backends. Rdfbox has been running on a single machine with two 2.53 GHz CPU cores, 4 GB Memory, a solid state drive and Ubuntu Linux 12.04 operating system. Hyperdex version 1.0.rc3-2 has been set up with two deamons (nodes) on a different machine with two 3.16 GHz CPU cores, 2 GB memory, 7200 RMP hard drive and Ubuntu Server 12.04 operating system. Both machines were connected on a 100 MBit LAN. The values are averages over 5 runs. For warm-cache, the queries were run ten times but measurements were averaged over the last five runs only.

			PQ1		**PQ2**	
Index	**Data Set**	**Loading**	**cold**	**warm**	**cold**	**warm**
Hyperdex	benchmark_50k	355.88 s	117.11 s	118.12 s	14.90 s	14.50 s
Hyperdex MS	benchmark_50k	410.51 s	81.86 s	82.63 s	2.41 s	2.42 s
Hyperdex	benchmark_1m	9822.23	3313.63	3309.42	579.56	581.76
Hyperdex MS	benchmark_1m	8404.09	1558.27	1549.46	41.41	39.31

Table A.1: Loading and query times for Hyperdex indices. Hyperdex refers to the single space implementation and Hyperdex MS to the multi-space implementation. 'cold' stands for cold-cache. 'warm' stands for warm-cache.

A.1.2 Preliminary Evaluation of Accumulo Backend

Table A.2 lists the results for the preliminary evaluation of Accumulo index backends. Rdfbox has been running on a single machine with two 2.53 GHz CPU cores, 4 GB Memory, a solid state drive and Ubuntu Linux 12.04 operating system. The Accumulo setup is the same as for the distributed loading evaluation described in Section 5.2.2. The values are averages over 5 runs. For warm-cache, the queries were run ten times but measurements were averaged over the last five runs only.

		PQ1		**PQ2**	
Setup	**Data Set**	**cold**	**warm**	**cold**	**warm**
single-threaded, local-join	benchmark_50k	3.62 s	2.76 s	0.87 s	0.71 s
multi-threaded, local-join	benchmark_50k	0.64 s	0.60 s	0.37 s	0.31 s
multi-threaded, dist. join	benchmark_50k	20.80 s	20.55 s	-	-
single-threaded, local-join	benchmark_1m	4.97 s	4.54 s	1.16 s	1.09 s
multi-threaded, local-join	benchmark_1m	0.93 s	0.78 s	0.70 s	0.59 s
multi-threaded, dist. join	benchmark_1m	435.28 s	436.75 s	-	-

Table A.2: Preliminary evaluation query execution times for Accumulo indices. Single-threaded and multi-threaded refers to the used query execution engine. 'cold' stands for cold-cache. 'warm' stands for warm-cache.

Setup	**Data Set**	**Loading Time**
multi-table	benchmark_50k	176.09 s
single-table	benchmark_50k	163.26 s
Map/Reduce Loading	benchmark_50k	72.93 s
Map Loading	benchmark_50k	49.93 s
multi-table	benchmark_1m	1,232.22 s
single-table	benchmark_1m	1,297.46 s
Map/Reduce Loading	benchmark_1m	310.51 s
Map Loading	benchmark_1m	152.51 s

Table A.3: Preliminary evaluation loading times for Accumulo indices. Multi-table and single-table refer to conventional loading and the data schema used. The schema depends on whether local or distributed join is used. Map/Reduce and Map Loading are the distributed variants.

A.2 Evaluation of the Accumulo Backend

A.2.1 Accumulo Encoding Phase Evaluation

	Buffer Size				
Data Set	**1 MB**	**10 MB**	**20 MB**	**30 MB**	**50 MB**
benchmark_100m	-	-	-	-	316,800 s*
dbpedia-subset	74,529 s	70,292 s	47,498 s	79,292 s	93,381 s
life-science	-	-	195,018	-	-
benchmark_25m	-	-	14,498	-	13,789
sp2bench	27,235 s	3,416	2,111 s	2,623	2,185

Table A.4: Encoding times measured with the dumper process for different buffer sizes. *The benchmark_100m data set measurement is approximate. The exact time could not be measured due to a software error.

Buffer Size	**1 MB**	**20 MB**	**50 MB**
Parsing	120.615151 s	49.249007 s	124.386944 s
Encoding	73,948.46138 s	1,982.024661 s	92,971.977185 s
Dumping	152.057585 s	49.096949 s	153.439845 s
Other	307.866294 s	30.37178 s	131.180588 s
Total	**74,529.00041 s**	**2,110.742397 s**	**93,380.984562 s**

Table A.5: Time splits for different encoding sub-tasks for the dbpedia-subset data set encoded with 1 MB, 20 MB and 50 MB buffer.

A.2.2 Accumulo Distributed Loading Phase Evaluation

Data Set	**Map/Reduce Loading**	**Map-only Loading**
dbpedia-subset	6,826 s	5,078 s
benchmark_100m	15,413 s	8,959 s
life-science	7,739 s	5,420 s
benchmark_25m	3,976 s	2,256 s
sp2bench	1,558 s	849 s

Table A.6: MapReducde loading times. Gathered on a cluster of 4 nodes, running a maximum of 3 map tasks concurrently and a maximum of 4 reduce tasks concurrently.

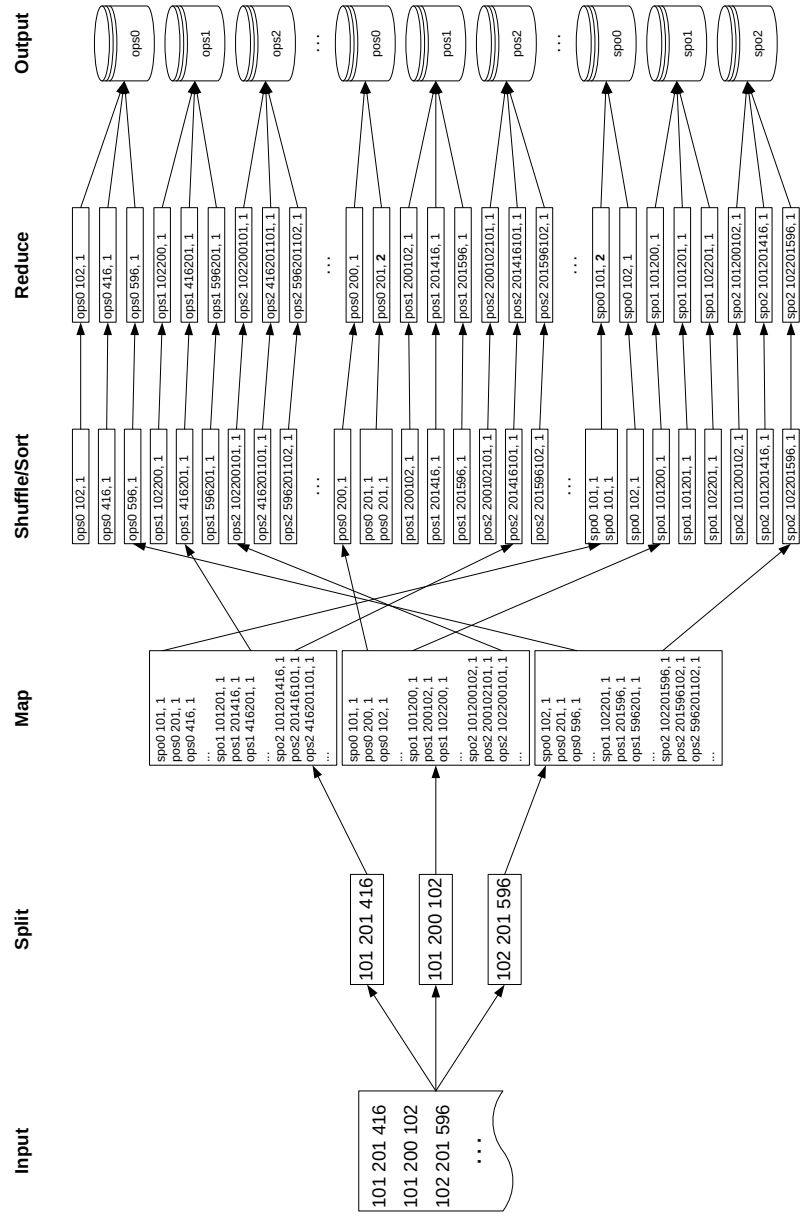

Figure A.1: MapReduce Accumulo Loader Process. The input file is split into chunks at line boundaries. The size of the individual splits can be configured. Each chunk is sent to a map process that creates all permutations for all indices. The MapReduce framework then sorts permutations by index and key and forwards them in-order to reducer processes that count the cardinalities. The reducer output is written directly into the corresponding Accumulo table on DFS level.

Figure A.2: MapReduce Map-Only Accumulo Loader Process. The input file is split into chunks at line boundaries. The size of the individual splits can be configured. Each chunk is sent to a map process that creates all permutations for all indices. No sorting and counting takes place. The map output is written directly into the corresponding Accumulo table on DFS level. Ordering and cardinality counting is done by Accumulo.

A.2.3 Accumulo Query Execution Time Evaluation

The following listings present the queries used for the evaluation of Query Execution times. Then the query execution setting is described and the raw results are listed.

Life Science Queries

Life science queries are based off the FedBench [41] life science queries adapted to only use features that the Rdfbox query execution engine can process. They are the same queries as used by [39] for the most part.

Listing A.3: Life Science Queries

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX drugbank: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX drugbank-drugs: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/drugs/>
PREFIX drugbank-category: <http://www4.wiwiss.fu-berlin.de/drugbank/resource/
    drugcategory/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-owl-drug: <http://dbpedia.org/ontology/drug/>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX chebi: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX purl: <http://purl.org/dc/elements/1.1/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/bio2rdf#>
PREFIX disease: <http://www4.wiwiss.fu-berlin.de/diseasome/resource/diseases/>

#LQ 1
SELECT ?drug ?melt WHERE {
    ?drug drugbank:meltingPoint ?melt .
}

#LQ 2
SELECT ?predicate ?object WHERE {
    drugbank-drugs:DB00201 ?predicate ?object .
}

#LQ 3
SELECT ?Drug ?IntDrug ?IntEffect WHERE {
    ?Drug rdf:type dbpedia-owl:Drug .
    ?y owl:sameAs ?Drug .
    ?Int drugbank:interactionDrug1 ?y .
    ?Int drugbank:interactionDrug2 ?IntDrug .
    ?Int drugbank:text ?IntEffect .
}

#LQ 4
SELECT ?drugDesc ?cpd ?equation WHERE {
    ?drug drugbank:drugCategory drugbank-category:cathartics .
    ?drug drugbank:keggCompoundId ?cpd .
    ?drug drugbank:description ?drugDesc .
    ?enzyme kegg:xSubstrate ?cpd .
    ?enzyme rdf:type kegg:Enzyme .
    ?reaction kegg:xEnzyme ?enzyme .
    ?reaction kegg:equation ?equation .
}

#LQ5
SELECT ?drug ?keggUrl ?chebiImage WHERE {
```

```
    ?drug rdf:type drugbank:drugs .
    ?drug drugbank:keggCompoundId ?keggDrug .
    ?keggDrug bio2rdf:url ?keggUrl .
    ?drug drugbank:genericName ?drugBankName .
    ?chebiDrug purl:title ?drugBankName .
    ?chebiDrug chebi:image ?chebiImage .
}

#LQ6
SELECT ?drug ?title WHERE {
         ?drug drugbank:drugCategory drugbank-category:micronutrient .
         ?drug drugbank:casRegistryNumber ?id .
         ?keggDrug rdf:type kegg:Drug .
         ?keggDrug bio2rdf:xRef ?id .
         ?keggDrug purl:title ?title .
}

#LQ7
SELECT ?drug ?enzyme ?reaction  Where {
        ?drug1 drugbank:drugCategory drugbank-category:antibiotics .
        ?drug2 drugbank-category:drugCategory drugbank-category:antiviralAgents .
        ?drug3 drugbank-category:drugCategory drugbank-category:
            antihypertensiveAgents .
        ?I1 drugbank:interactionDrug2 ?drug1 .
        ?I1 drugbank:interactionDrug1 ?drug .
        ?I2 drugbank:interactionDrug2 ?drug2 .
        ?I2 drugbank:interactionDrug1 ?drug .
        ?I3 drugbank:interactionDrug2 ?drug3 .
        ?I3 drugbank:interactionDrug1 ?drug .
        ?drug owl:sameAs ?drug5 .
        ?drug5 rdf:type dbpedia-owl:Drug .
        ?drug drugbank:keggCompoundId ?cpd .
        ?enzyme kegg:xSubstrate> ?cpd .
        ?enzyme rdf:type kegg:Enzyme> .
        ?reaction kegg:xEnzyme> ?enzyme .
        ?reaction kegg:equation> ?equation .
}
LIMIT 1

#LQ8
SELECT ?drug  WHERE {
        ?drug1 drugbank:possibleDiseaseTarget disease:302 .
        ?drug2 drugbank:possibleDiseaseTarget disease:53 .
        ?drug3 drugbank:possibleDiseaseTarget disease:59 .
        ?drug4 drugbank:possibleDiseaseTarget disease:105 .
        ?I1 drugbank:interactionDrug2 ?drug1 .
        ?I1 drugbank:interactionDrug1 ?drug .
        ?I2 drugbank:interactionDrug2 ?drug2 .
        ?I2 drugbank:interactionDrug1 ?drug .
        ?I3 drugbank:interactionDrug2 ?drug3 .
        ?I3 drugbank:interactionDrug1 ?drug .
        ?I4 drugbank:interactionDrug2 ?drug4 .
        ?I4 drugbank:interactionDrug1 ?drug .
        ?drug drugbank:casRegistryNumber ?id .
        ?keggDrug rdf:type kegg:Drug .
        ?keggDrug bio2rdf:xRef ?id .
        ?keggDrug dc:title ?title .
}

#LQ9
SELECT ?d ?drug5 ?cpd ?enzyme ?equation WHERE {
        ?drug1 drugbank:possibleDiseaseTarget disease:261 .
```

```
        ?l1 drugbank:interactionDrug2 ?drug1 .
        ?l1 drugbank:interactionDrug1 ?drug .
        ?drug drugbank:possibleDiseaseTarget  ?d .
        ?drug owl:sameAs ?drug5 .
        ?drug5 rdf:type dbpedia-owl:Drug .
        ?drug drugbank:keggCompoundId ?cpd .
        ?enzyme kegg:xSubstrate ?cpd .
        ?enzyme rdf:type kegg:Enzyme .
        ?reaction kegg:xEnzyme ?enzyme .
        ?reaction kegg:equation ?equation .
}
LIMIT 1

#LQ10
SELECT ?drug5 ?drug6
WHERE {
        ?drug1 drugbank:possibleDiseaseTarget  disease:319 .
        ?drug1 drugbank:possibleDiseaseTarget  disease:270 .
        ?l1 drugbank:interactionDrug1 ?drug1 .
        ?l1 drugbank:interactionDrug2 ?drug .
        ?drug1 owl:sameAs ?drug5 .
        ?drug owl:sameAs ?drug6 .
}
LIMIT 10000
```

Query	Avg. Execution Time	Standard Error
LQ 1	3.65 s	0.04
LQ 2	0.31 s	0.01
LQ 3	50.59 s	3.45
LQ 4	0.39 s	0.00
LQ 5	33.52 s	0.14
LQ 6	2.41 s	0.14
LQ 7	N/A	N/A
LQ 8	0.23 s	0.04
LQ 9	637.30 s	N/A
LQ 10	8.41 s	0.08

Table A.7: Query execution times for life-science data set. Average is calculated over 5 passes. *LQ 7* did not finish and was aborted after several hours. *LQ 9* is a single measurement of the time required to return a single result (LIMIT 1).

SP^{2}Bench Queries

SP^{2}Bench queries originate from FedBench [41] and are adapted to only use features, which the Rdfbox query execution engine can process. They are the same queries as used by [39].

Listing A.4: SP2Bench Queries

```
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
```

```
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX person: <http://localhost/persons/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>

#SQ 1
SELECT ?yr
WHERE {
    ?journal rdf:type bench:Journal .
    ?journal dc_title "Journal 1 (1940)" .
    ?journal dcterms:issued ?yr
}

#SQ 2
SELECT ?inproc ?author ?booktitle ?title ?proc ?ee ?page ?url ?yr
WHERE {
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?author .
    ?inproc bench:booktitle ?booktitle .
    ?inproc dc:title ?title .
    ?inproc dcterms:partOf ?proc .
    ?inproc rdfs:seeAlso ?ee .
    ?inproc swrc:pages ?page .
    ?inproc foaf:homepage ?url .
    ?inproc dcterms:issued ?yr
}
LIMIT 100

#SQ 3
SELECT DISTINCT ?person ?name
WHERE {
    ?article rdf:type bench:Article .
    ?article dc:creator ?person .
    ?inproc rdf:type bench:Inproceedings .
    ?inproc dc:creator ?person .
    ?person foaf:name ?name
}

#SQ 4
SELECT ?subject ?predicate
WHERE {
    ?subject ?predicate person:Paul_Erdoes
}
LIMIT 1000
```

Query	Avg. Execution Time	Standard Error
SQ 1	0.29 s	0.00
SQ 2	70.06 s	3.37
SQ 3	N/A s	N/A
SQ 4	0.45 s	0.00

Table A.8: Query execution times for sp2bench data set. Average is calculated over 5 passes. *SQ 3* did not finish and was aborted after several hours of execution.

Berlin SPARQL Benchmark Queries

These queries originate from the Berlin SPARQL Benchmark (BSBM) [12] but are adapted to only use features, that the Rdfbox query execution engine can process.

Listing A.5: Berlin SPARQL Benchmark Queries

```
PREFIX bsbm-inst: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>

#BQ 1
SELECT DISTINCT ?product ?label
WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
        ProductType13> .
    ?product bsbm:productFeature <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
        instances/ProductFeature148> .
    ?product bsbm:productFeature <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
        instances/ProductFeature580> .
    ?product bsbm:productPropertyNumeric1 ?value1 .
LIMIT 10

#Query 2 is PQ 2

#BQ 3
SELECT ?product ?label ?p1 ?p3
WHERE {
    ?product rdfs:label ?label .
    ?product rdf:type <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
        ProductType9> .
        ?product bsbm:productFeature <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
            v01/instances/ProductFeature353> .
        ?product bsbm:productPropertyNumeric1 ?p1 .
        ?product bsbm:productPropertyNumeric3 ?p3 .
}
LIMIT 10

#BQ 4
SELECT ?product ?label ?p1
WHERE {
        ?product rdfs:label ?label .
        ?product rdf:type <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances
            /ProductType51> .
        ?product bsbm:productFeature <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/
            v01/instances/ProductFeature241> .
            ?product bsbm:productFeature <http://www4.wiwiss.fu-berlin.de/bizer/
                bsbm/v01/instances/ProductFeature2269> .
            ?product bsbm:productPropertyNumeric1 ?p1 .
}
LIMIT 20

#BQ 5
SELECT DISTINCT ?product ?productLabel ?origProperty1 ?origProperty2 ?simProperty1
     ?simProperty2
WHERE {
```

```
        ?product rdfs:label ?productLabel .
        <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
            dataFromProducer492/Product24166> rdf:type ?prodtype.
        ?product rdf:type ?prodtype .
        <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
            dataFromProducer492/Product24166> bsbm:productFeature ?prodFeature .
        ?product bsbm:productFeature ?prodFeature .
        <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
            dataFromProducer492/Product24166> bsbm:productPropertyNumeric1 ?
            origProperty1 .
        ?product bsbm:productPropertyNumeric1 ?simProperty1 .
        <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
            dataFromProducer492/Product24166> bsbm:productPropertyNumeric2 ?
            origProperty2 .
        ?product bsbm:productPropertyNumeric2 ?simProperty2 .
}
LIMIT 5

#BQ 6
SELECT ?product ?label
WHERE {
        ?product rdfs:label ?label .
    ?product rdf:type bsbm:Product .
}
LIMIT 2000

#BQ 7
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review ?revTitle
       ?reviewer ?revName ?rating1 ?rating2
WHERE {
        <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/
            dataFromProducer324/Product15766> rdfs:label ?productLabel .
        ?offer bsbm:price ?price .
        ?offer bsbm:vendor ?vendor .
        ?vendor rdfs:label ?vendorTitle .
        ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE> .
    ?offer dc:publisher ?vendor .
    ?offer bsbm:validTo ?date .
}

#BQ 8
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1 ?rating2 ?rating3
     ?rating4
WHERE {
        ?review bsbm:reviewFor <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
            instances/dataFromProducer352/Product17091> .
        ?review dc:title ?title .
        ?review rev:text ?text .
        ?review bsbm:reviewDate ?reviewDate .
        ?review rev:reviewer ?reviewer .
        ?reviewer foaf:name ?reviewerName .
}
LIMIT 20

#Query 9 can not be applied to Rdfbox

#BQ 10
SELECT DISTINCT ?offer ?price ?deliveryDays ?date
WHERE {
        ?offer bsbm:product <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/
            instances/dataFromProducer572/Product28147> .
        ?offer bsbm:vendor ?vendor .
```

```
        ?offer dc:publisher ?vendor .
        ?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#US> .
        ?offer bsbm:deliveryDays ?deliveryDays .
        ?offer bsbm:price ?price .
        ?offer bsbm:validTo ?date .
}
LIMIT 10
```

	benchmark_25m		benchmark_100m	
Query	**Avg. Time**	**StdErr**	**Avg. Time**	**StdErr**
BQ 1	18.00 s	0.39	58.63 s	0.23
PQ 2	0.66 s	0.06	0.71 s	0.02
BQ 3	16.11 s	0.06	58.63 s	0.18
BQ 4	8.44 s	0.02	17.08 s	0.09
BQ 5	71.65 s	0.13	9.71 s	0.40
BQ 6	61.52 s	0.14	233.47 s	0.24
BQ 7	229.71 s	1.22	673.59 s	1.37
BQ 8	N/A s	N/A	N/A s	N/A
BQ 10	1.54 s	0.10	1.68 s	0.03

Table A.9: Query execution times for the benchmark_25m and benchmark_100m data sets. Average is calculated over 5 passes. PQ 2 is Query 2 from BSBM., Query 9 is not applicable to Rdfbox.

List of Figures

List of Tables

List of Listings