



**University of
Zurich^{UZH}**

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

Thesis January 30, 2013

Thomas Ritter
of Mauren, Liechtenstein

Student-ID: 07-057-789
thomas.ritter@uzh.ch

Advisor: **Cosmin Basca**

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

Acknowledgements

I would like to thank Prof. Abraham Bernstein for offering me to work on this challenging topic and pursue my master thesis within his research group. Further, I would like to thank my supervisor Cosmin Basca for answering my innumerable questions and for discussing issues and ideas regarding this work.

My sincere thanks also go to my family and friends who supported me for the last months.

Zusammenfassung

Rdfbox ist ein nativer Tripel-Speicher und implementiert das Hexastore-Indexierungskonzept. Hexastore permutiert die RDF Tripel-Elemente (Subjekt, Prädikat, Objekt) und baut daraus sechs Indexe auf. Rdfbox übernimmt dieses Konzept und wendet es auf Key-Value-Speicher an. Rdfbox nutzte dafür bisher Tokyo Cabinet und das Ziel dieser Arbeit ist es, Rdfbox um Kyoto Cabinet-, LevelDB- und Redis-Backends zu erweitern. Diese Backends sollen einfach austauschbar sein, auch für einzelne Indexe. Das Einlesen der Tripel soll analysiert und verbessert werden und zudem an die hinzugefügten Backends angepasst werden. Die Leistung dieser Backends wird dann verglichen und analysiert anhand von synthetischen und realen Datenbeständen und Abfragen.

Abstract

Rdfbox is a native triple store and implements the Hexastore indexing concept. Hexastore permutes RDF triple elements (subject, predicate, object) to build six indexes. Rdfbox takes this indexing scheme and maps it to key-value storage. Prior to this work, it used the Tokyo Cabinet key-value store as its indexing backend. This work extends it with Kyoto Cabinet, LevelDB and Redis backends and enables to easily exchange them, even on a per-index level. Further, Rdfbox's triple loading is analyzed, improved and adapted to the added backends. The backends' performance is then compared and analyzed with synthetic and real-world data sets and queries.

Table of Contents

1	Introduction	1
2	Background and Related Work	3
2.1	Resource Description Framework	3
2.2	Ontologies and Linked Data	4
2.3	SPARQL	7
2.4	Triple Stores	10
2.5	Hexastore	11
2.5.1	Index Operations	11
2.5.2	Shared Indexes	13
2.5.3	Example Query	14
2.5.4	Dictionary Encoding	14
3	Rdfbox	17
3.1	Index Backends	17
3.2	Mapping Hexastore Indexes to Key-Value Stores	19
3.3	Query Plan Optimization with Cardinalities	22
3.4	Rdfbox Architecture	24
4	Implementation	27
4.1	Cython	27
4.2	Configuration	28
4.3	Indexes	29
4.3.1	Tokyo Cabinet	29
4.3.2	Kyoto Cabinet	30
4.3.3	Kyoto Cabinet On Memory	31
4.3.4	LevelDB	31
4.4	Problems with Index Backends	32
4.4.1	Redis	33
4.4.2	Tokyo Cabinet In Memory	34
4.5	Unit Testing Indexes	35
4.6	Loading Triples	36

4.6.1	Encoding and Sorting	36
4.6.2	Merging and Loading into Index	38
4.6.3	Iterative Loading	39
5	Evaluation	41
5.1	Datasets and Queries	41
5.2	Loading Data	43
5.3	Queries	48
6	Discussion and Conclusion	53
6.1	Discussion	53
6.2	Conclusion	54
A	Appendix	59
A.1	SPARQL Queries	59

1

Introduction

The Semantic Web[1] is a concept developed by the W3C¹. Its goal is to enable machine-readable data on the web in a common format and to integrate data collections from multiple sources, as well as to further reuse this data across application and organization boundaries.

The Semantic Web concept is implemented using the Resource Description Framework [3] (RDF). RDF is a data representation format for graphs and this graph format is stored as a list of edges. These edges are triples (or quads) and consist of subject, predicate (or property) and object. The subjects and objects (also called resources) are nodes in a directed graph, with predicates as edges.

An increasing amount of RDF data is published on Web [2] and extracting information from these inter-linked graphs is becoming increasingly important. To efficiently query this data, RDF data is managed by so-called triple stores. These triple stores vary in their approach to store, index and retrieve RDF data. Relational-based triple stores use relational database systems to store RDF data and map the RDF graph data to a relational model. In contrast to them, native triple stores commonly employ a custom storage mechanism and try to adapt to the RDF graph data model.

Hexastore is part of a family of native triple store approaches and suggests a storage scheme for RDF data that employs six indexes corresponding to the six permutations of subject, predicate and object. With this extensive indexing approach, single statement queries with any combination of unbound variables are equally expensive to execute. This is a significant advantage compared to most relational-based triple stores, which often favor queries with bound predicates.

Motivation Rdfbox is an implementation of the Hexastore concept written in Python. It is designed to use key-value stores as indexing backends. Prior to this work, Rdfbox solely used Tokyo Cabinet, a key-value store, as its indexing backend. The goal for this work is to include additional key-value stores and enable to exchange them, even on a per-index basis. Further, a diverse set of backends should be included to compare the

¹<http://www.w3.org/2001/sw/>

B+ tree approach by Tokyo Cabinet with other storage techniques. The added backends are Kyoto Cabinet, LevelDB and Redis.

In addition, the goal is to analyze and improve triple loading and to adapt it to the indexing backends used.

Outline The remainder of this work is structured as follows. Chapter two introduces the ideas and tools behind the Semantic Web vision and explains the Hexastore indexing scheme in detail. Then in chapter three, the indexing backends are presented and the Hexastore index mapping to a key-value store is explained. Further, it is explained how Rdfbox optimizes query execution plans and the Rdfbox architecture related to indexing is outlined. Chapter four discusses implementation details regarding the configuration and integration of indexing backends and explains the unit testing approach. Further, the triple loading work-flow and its variants are examined. After that, chapter five evaluates the triple loading performance for all indexing backends and further evaluates their query execution performance. The last chapter discusses the results and concludes this work.

2

Background and Related Work

This section introduces the ideas behind the Semantic Web and explains its core technologies. First, the Resources Description Framework (RDF) is explained. After that, ontologies and linked data are discussed. This is followed by an introduction to SPARQL, a query language for RDF data and after that, triple stores and their storage mechanisms are explained. At the end, Hexastore and its indexing scheme is introduced.

2.1 Resource Description Framework

The Semantic Web idea is implemented using RDF[3], the Resource Description Framework¹, a model for Semantic Web data exchange. RDF allows to merge data sources with different schemas and to evolve schemas without the need to notify consumers necessarily.

RDF data is represented as triples in the form of subject, predicate (or property) and object. A triple in this form is also called a statement. One such triple statement in a RDF database could be: Alice knows John. **Alice** is the subject, **knows** is the predicate or property and **John** is the object. The subjects are also called resources and triples are statements about these resources.

The object in a triple might also appear as the subject of another triple. In the case above, the object **John** could be the subject in: John knows Linda. The collection of triples can be understood as a directed graph, where subjects and objects are nodes and the predicates are edges. However, not all objects are necessarily resources (or subjects) themselves. Objects may be literals as well. Consider this triple: Alice familyName 'Miller'. In this case, the object is a literal and the triple is a statement about the resource **Alice**.

In RDF, resources (subjects) and predicates have to be uniquely identifiable. For this purpose, URIs are used to distinguish between them. The following could be a collection of triples:

¹<http://www.w3.org/RDF/>

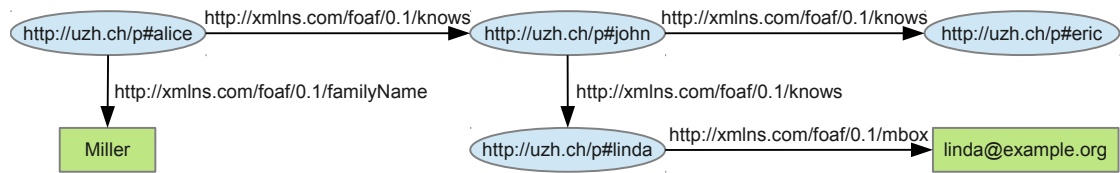


Figure 2.1: A directed graph representation of the triple collection. Resources (blue ellipses) and literals (green boxes) are nodes, predicates are edges (arrows).

```

<http://uzh.ch/p#alice> <http://xmlns.com/foaf/0.1/knows> <http://uzh.ch/p#john>
<http://uzh.ch/p#john> <http://xmlns.com/foaf/0.1/knows> <http://uzh.ch/p#linda>
<http://uzh.ch/p#john> <http://xmlns.com/foaf/0.1/knows> <http://uzh.ch/p#eric>
<http://uzh.ch/p#alice> <http://xmlns.com/foaf/0.1/familyName> "Miller"
<http://uzh.ch/p#linda> <http://xmlns.com/foaf/0.1/mbox> "linda@example.org"
  
```

The RDF elements that are not literals are placed inside angle brackets. Since RDF data are part of the Semantic Web, URIs commonly use *http://* as their protocol and a domain name and path that reflects the resource it describes. These URIs may be actual, resolvable Web resources, but this is not a requirement. The only restriction is that they must be unique.

Subjects may be URI references or blank nodes, but not literals. Blank nodes may appear as objects and subjects, but do not have a unique URI, but a placeholder name that is only unique for the data set they are part of. Blank nodes are mentioned here for completeness. Explaining their purpose is not in the scope of this introduction.

In figure 2.1, the same triples represented in textual form are visualized as a graph. Resources are visualized as blue ellipses, while literals are green boxes. The edges between them are predicates. Two nodes connected with an arrow form a triple. In the graph it is easily visible that **John** is the object in the triple formed with **Alice**, and the subject in both triples with **Eric** and **Linda**.

In the triples presented above, the predicates **knows**, **familyName** and **mbox** are well established predicates defined by the FOAF (Friend Of A Friend) vocabulary². In this vocabulary, a collection of other predicates are defined, such as **birthday**, **gender** or **firstName** to describe a person's attributes or relationships of this person to other persons or entities. FOAF also includes concepts, such as **Person** or **Document**, which allow to categorize resources. All these predicates and concepts have a well-defined URI such as the predicates **knows** and **mbox** used above. This collection of well-defined predicates and concepts is also called an ontology.

2.2 Ontologies and Linked Data

Many ontologies for a diverse set of subject matters exist. The most popular ontologies are *Dublin Core*³ for document meta-data, *FOAF* for people and organizations and

²<http://xmlns.com/foaf/spec/>

³<http://dublincore.org/>

their social network, and several smaller ontologies such as *Basic Geo Vocabulary*⁴ which defines spatial data such as longitude, latitude and altitude, and BIO, a vocabulary for biographical information.

Most of these ontologies are, however, not formal standards. They are conceptual proposals on how to structure and format data in a specific domain. When publishing RDF data, it is always possible to use own, custom predicates and concepts. Reusing existing ontologies, however, enables the published RDF data to be combined with other data in the same domain.

It is always possible to add a triple to a collection of RDF data. This makes it possible to change a schema over time. When using custom predicates and concept for domain data, it is still possible to introduce an ontology at a later point in time, by adding statements about the same resources. For example, if the FOAF vocabulary did not yet exist, information about the email address of a resource could be expressed with the custom predicate `http://uzh.ch/ont#hasEmailAddress`. At a later point in time, the FOAF vocabulary could be supported by adding a triple with the same subject, but the `foaf:mbox` predicate and the same email address as a literal object. The old triple could still be part of the RDF database and the new triple would just be an extension to the original data. This RDF feature makes RDF data management flexible, and allows database schemas to evolve over time. However, old data needs to be converted to a new schema. For data describing the same domain in different ontologies, *ontology mediation* can be a solution to the problem.

RDF is therefore only a representation format. The ontologies used define the semantics, but RDF enables multiple schemas and ontologies to co-exist. Further, two databases may easily be merged, even if they use different ontologies. By adding statements about resources, a progressive schema alignment is still possible.

The uniqueness of RDF resources lead to a key feature in the Semantic Web concept: the possibility to link resources between RDF databases. This concept is called Linked Data and plays a key role in the Semantic Web. Arguably the most popular collection of RDF data is a project called DBpedia⁵. DBpedia aims to extract information taken from the online encyclopedia Wikipedia, and builds a RDF database based on it. The information is, among other sources, based on the Wikipedia article info boxes and the article categories. When publishing RDF data, not only existing ontologies can be reused, but links may be established to resources defined in popular RDF databases, such as the DBpedia database. Resources in these data hubs are often linked to and resources are linked to other, popular databases. These linked data hubs are depicted in figure 2.2.

DBpedia is perhaps most popular, because of its inherent universal scope, but other, more specialized databases are often linked to as well, such as GeoNames⁶, which is a database with geographic and spatial data. It describes resources such as cities, federal states and countries, but also mountains, roads and other geographical concepts. Every

⁴<http://www.w3.org/2003/01/geo/>

⁵<http://dbpedia.org/>

⁶<http://www.geonames.org/>



<http://lod-cloud.net/>

item, called “feature” in GeoNames, has a stable, unique URI which other databases can link to. Moreover, these features are linked to each other internally. For example, cities are linked to federal states, and federal states are linked to their countries. Therefore when publishing RDF data, any resource could link to a geographical concept in GeoNames, without having to specify any further information. Moreover, links to equivalent concepts in other popular databases may be established in addition. For example, when describing the location of a person in a local RDF database, it is possible to link to a resource in GeoNames.

```
<http://uzh.ch/p/alice> <http://xmlns.com/foaf/0.1/based_near>  
    <http://sws.geonames.org/2657896/>
```

This RDF triple states that **Alice** is based near Zürich, Switzerland. When consulting the GeoNames database, we could find triples for the linked resource.

```
<http://sws.geonames.org/2657896/> <http://www.geonames.org/ontology#name> "Zurich"
<http://sws.geonames.org/2657896/> <http://www.geonames.org/ontology#population> 341730
<http://sws.geonames.org/2657896/> <http://www.w3.org/2003/01/geo/wgs84_pos#lat> 47.37
<http://sws.geonames.org/2657896/> <http://www.w3.org/2003/01/geo/wgs84_pos#long> 8.55
<http://sws.geonames.org/2657896/> <http://www.geonames.org/ontology#parentCountry>
    <http://sws.geonames.org/2658434/>
```


This is an extract from the GeoNames database. It shows statements about the name for the resource, the population, the latitude and longitude – using the aforementioned Basic Geo Vocabulary – and the parent country, a link to a resource in the same database. With a stable, unique URI for the city of Zürich it is possible for any entity to publish data that is related to Zürich with a link to the GeoNames URI. With this information, the two data sets may be combined. When extracting information about a resource in the local database, with a link to a GeoNames concept, suddenly richer queries are conceivable. For example, with the initial triple collection, statements about the city the people reside in could be added. Then, a query to find people in a specific *country* is feasible, even though only a link to the *city* in the GeoNames database is expressed. The information on which country a city belongs to is part of the GeoNames database. Further, information added at a later point in time to the GeoNames database enable even richer queries, without changing the local database.

2.3 SPARQL

As demonstrated in the Linked Data example, a query mechanism for RDF data is essential. SPARQL is the most widely used query language for RDF data and its syntax is similar to SQL. Although SQL and SPARQL share some keywords and syntax, the underlying data model is quite different. SQL is designed to retrieve relational data, whereas SPARQL retrieves RDF elements based on a graph of data. Consider this query:

```
SELECT ?name WHERE {  
  <http://uzh.ch/p/alice> <http://xmlns.com/foaf/0.1/knows> ?name  
}
```

Listing 2.1: SQL, SPARQL query

This query (2.1) is a statement-based query. It includes one triple pattern, in which the subject and the predicate are specified, but the object is a variable, which is marked as *?name*. The subject and predicate are called *bound* variables, while the object in this query is *unbound*.

This query returns a list of objects that appear in triples with the subject `http://uzh.ch/p/alice` and the predicate `http://xmlns.com/foaf/0.1/knows`. Applied to the triples presented in Section 2.1, the result would be a list with one element, the resource `http://uzh.ch/p/john`.

The query in 2.1 can also be formulated as in 2.2, where RDF elements are written with *prefixes*. Prefixes are a way to shorten RDF elements and using keywords instead of lengthy base URIs. When using RDF elements with prefixes, the angle brackets are omitted and the prefix, followed by a colon and the rest of the URI is written. Instead of writing `<http://uzh.ch/p/alice>`, a prefix `uzh` is defined, which expands to `http://uzh.ch/p/`, and the resource can be rewritten as `uzh:alice`. The same principle applies to the `foaf` keyword in the query.

```

PREFIX uzh: <http://uzh.ch/p/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name WHERE {
  uzh:alice foaf:knows ?name
}

```

Listing 2.2: SQ1.1, SPARQL query

```

PREFIX uzh: <http://uzh.ch/p/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friendOfFriend ?emailAddress WHERE {
  uzh:alice foaf:knows ?friend .
  ?friend foaf:knows ?friendOfFriend .
  ?friendOfFriend foaf:mbox ?emailAddress
}

```

Listing 2.3: SQ2, SPARQL query

SPARQL of course allows for more complicated terms. Consider the query in 2.3. It is a combination of three query patterns, called a *basic graph pattern*. The first statement queries for all *objects* that Alice *knows*. The second statement reuses the variable *?friend* from the first statement and queries for all objects that Alice’s friends know. These are eventually Alice’s friends of friends. These friends of friends are captured in the *?friendOfFriend* variable and are reused for the third and last statement. The third statement searches for triples with an *mbox* predicate and captures the objects in the *?emailAddress* variable.

When two statements use one or more of the same variables, these statements are *joined*.

In the **SELECT** clause we define the projections *?friendOfFriend* and *?emailAddress* as the variables which are part of the result, disregarding the *?friend* variable which is part of the basic graph pattern.

The basic graph pattern in query 2.3 follows a chain pattern, as the object of the first query statement is an unbound variable, which is used for the subject of the following statement, effectively chaining query statements.

Applied to the original triple collection, there is only one result which fulfills this query:

friendOfFriend	emailAddress
<http://uzh.ch/p/linda>	linda@example.org

The SPARQL query can also be viewed as a graph pattern that has to be matched. The results are the sets of RDF elements that can be substituted with the variables

in the query statements to match a part of the RDF data graph. Each set of RDF elements that fulfill the query pattern is a result of the query. Therefore, applied to the query above and the original collection of triples as the graph, the first query statement matches `uzh:john` as the object. In the second statement, the *?friend* variable can thus be replaced with `uzh:john`. The second query statement then matches with two triples and the objects for *?friendOfFriend* are `uzh:linda` and `uzh:eric`. The third query statement thus may match either of these two objects as the subjects. However, `uzh:linda` is the only subject that matches with the `mbox` predicate, as no triple exists with the subject `uzh:eric` and the predicate `foaf:mbox`. Therefore, only one set of RDF elements matches with the basic graph pattern. Translated to natural language, the query tries to find Alice's friends of friends with an email address entry.

If the email address entry is not mandatory, the third query pattern can be defined as *optional*:

```
PREFIX uzh: <http://uzh.ch/p/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?friendOfFriend ?emailAddress WHERE {
  uzh:alice foaf:knows ?friend .
  ?friend foaf:knows ?friendOfFriend .
  OPTIONAL { ?friendOfFriend foaf:mbox ?emailAddress }
}
```

Listing 2.4: SQ3, SPARQL query

Note that compared to the previous query, this query uses an **OPTIONAL** basic graph pattern for the third statement. For this query, there are two result sets:

friendOfFriend	emailAddress
<http://uzh.ch/p/eric>	
<http://uzh.ch/p/linda>	linda@example.org

This time, `uzh:eric` matches as well, because the `foaf:mbox` predicate is optional.

To readopt the GeoNames example from Section 2.2, a query for all people based in Switzerland could be:

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX gn: <http://www.geonames.org/ontology#>

SELECT ?person ?cityName WHERE {
  ?person rdf:type foaf:Person .
  ?person foaf:based_near ?city .
  ?city gn:name ?cityName .
  ?city gn:parentCountry ?parentCountry .
  ?parentCountry gn:name "Switzerland"
}
```

Listing 2.5: SQ4, SPARQL query

The query in 2.5 is based on two assumptions. First, it is assumed that resources describing a person feature a triple that matches the first query statement and that these resources have information about their location via the `foaf:based_near` predicate (second statement). If a resource does not feature triples with the `rdf:type` and `foaf:based_near` predicates, it will not match the query. Second, it is assumed that the *?city* variable in the second statement is a GeoNames resource. This is the link from the local RDF database to the public GeoNames database. It is assumed that the GeoNames data set is not part of the local data set. Therefore to be able to match the third, fourth and fifth statement, access to the GeoNames database is expected. This consideration is, however, at the semantic level. SPARQL does not infer which statements link to external sources.

These example queries only show a small subset of the SPARQL query language. It features more sophisticated constructs, but these are not in the scope for this paper. The W3C Recommendation for SPARQL⁷ is the official reference specification.

2.4 Triple Stores

An increasing amount of RDF data is produced and published, containing millions, or in some cases even billions of triples. An instrument to query this data – SPARQL – was introduced in Section 2.3. However, the mechanisms to store and index RDF data becomes more important as the RDF data sets grow.

There are many different ways to store RDF data and solutions to this problem are called triple stores. One approach to store triples builds on top of relational databases.

Relational-based triple stores may put all triples into a (possibly large) table, with three columns for subjects, predicates and objects. To speed up querying, one or more indexes on either of the columns are used. The advantage to this approach is, that already available relational database systems — a proven technology — can be reused to store the triples. The disadvantage is the limited scalability. With queries consisting of many statements, expensive self-joins are necessary. If not all three columns are indexed, these joins become more expensive, as the database engine has to perform table-scans.

A more sophisticated approach to store triples in relational databases is to apply a “vertical partitioning” scheme. Triples are not all put into one table but are split based on their predicate. For each distinct predicate, a table is built with subject and object columns. This reduces the problem of the one-table-approach, but does not solve the scalability issue at large. This approach still works best for queries with a small number of statements. The more statements the query includes, the more table joins are necessary. Moreover, if predicates are unbound, all tables have to be joined, which is even worse.

Therefore triple stores based on relational databases commonly favor query patterns where either the subject, predicate or object is bound (known).

⁷<http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115/>

triple name	subject	predicate	object
T1	Jack	knows	Emma
T2	Jack	knows	Ben
T3	Jack	knows	Nicki
T4	Jack	likes	Basketball
T5	Emma	knows	Ben
T6	Emma	knows	Nicki
T7	Emma	likes	Football
T8	Ben	knows	Nicki
T9	Ben	likes	Basketball
T10	Ben	likes	Football

Table 2.1: A collection of ten example triples. Elements are simplified to a name instead of a full URI.

2.5 Hexastore

Hexastore[5] is a triple store architecture developed at the University of Zürich. Unlike some of the competing triple store solutions, it does not rely on relational database infrastructure, but is a native approach to store RDF triples. As explained in Section 2.4, many triple stores based on relational databases favor query patterns with bound subjects, predicates or objects. For example, triple stores that implement vertical partitioning favor queries for graph patterns where predicates are bound. If a triple store based on vertical partitioning executes a query with unbound predicates, all tables have to be consulted to answer the query, which is expensive.

Hexastore, however, indexes triples without favoring any query pattern. Therefore, to answer single-statement queries, it does not matter if the query pattern has an unbound subject, predicate or object; any query pattern is equally expensive to evaluate. Consider the triples in table 2.1 as the collection of triples to index. The RDF elements use generic names instead URIs for the sake of simplicity. The triples use the subjects **Jack**, **Emma** and **Ben** and the predicates used are **knows** and **likes**. Some subjects appear again as objects, these are: **Emma** and **Ben**. The other objects – **Nicki**, **Basketball** and **Football** – appear as objects only. The objects in these triples are all resources, although **Basketball** and **Football** could be string literals as well.

2.5.1 Index Operations

Hexastore now permutes every RDF triple consisting of the subject (S), the predicate (P) and the object (O) to form six indexes named after the order of the RDF elements they represent: SPO, SOP, PSO, POS, OSP, OPS. Consider figure 2.3, where the collection of triples is indexed. Note that not all indexes are depicted – OSP and SOP are missing – and the indexes themselves are not fully elaborated to avoid an overcrowded figure. In all indexes, only one subject, predicate or object is depicted as a starting

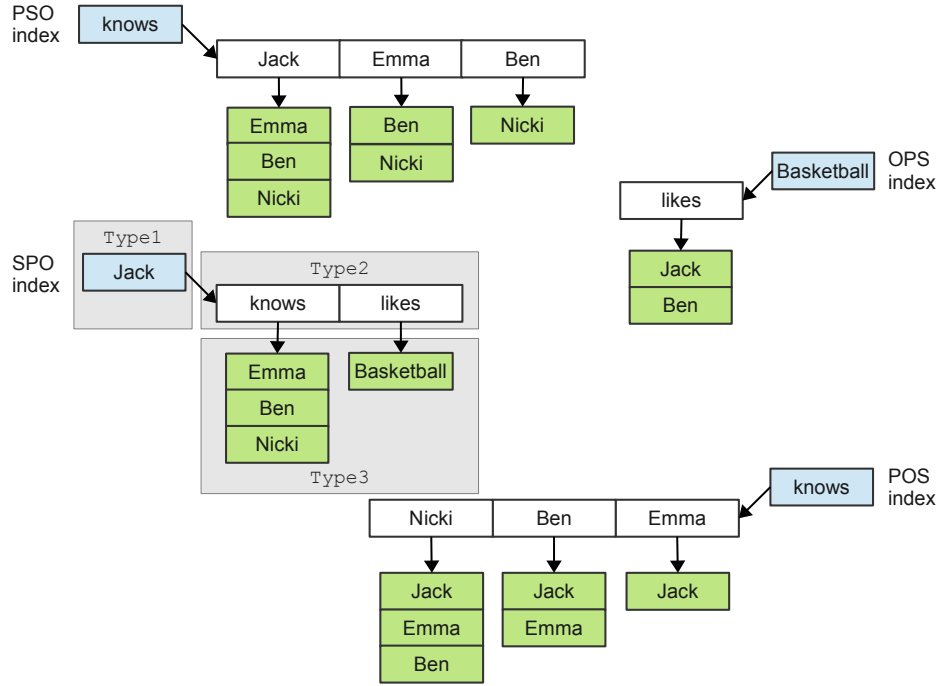


Figure 2.3: A graphical representation of the Hexastore index. The OSP and SOP indexes are missing. Each index has only one heading RDF element to avoid an overcrowded figure.

point, although there are more elements. For example, in the SPO index, only **Jack** is depicted as a starting point. The subjects **Emma** and **Ben** are not depicted, although they are part of the SPO index.

These indexes, as explained above, store triples in permuted order. Each of these six indexes has a primary RDF element (subject, predicate or object) and prioritizes the remaining two RDF elements. If the primary element is defined as **Type1**, the secondary element as **Type2** and the third element as **Type3**, each index has two levels. The first-level index stores **Type1** elements each with a pointer to a second-level index with **Type2** elements. In this second-level index, each **Type2** element points to a set of **Type3** elements. The first-level index provides the following operation:

$$getLevel2Index(a) : \text{Type1} \longrightarrow \text{Index}[\text{Type2}]$$

The first operation, **getLevel2Index** returns for a given **Type1** element its second-level **Type2** index. The second operation is provided by the second-level index:

$$getType3Set(b) : \text{Type2} \longrightarrow \text{Set}[\text{Type3}]$$

The second operation, **getType3Set**, returns for the given **Type2** element the set of **Type3** elements.

name	query pattern	index	operation
QP1	$\langle s, ?p, \cdot \rangle$	SPO	getLevel2Index(subject)
QP2	$\langle s, p, ?o \rangle$	SPO	getType3Set(predicate)
QP3	$\langle s, \cdot, ?o \rangle$	SOP	getLevel2Index(subject)
QP4	$\langle s, ?p, o \rangle$	SOP	getType3Set(object)
QP5	$\langle ?s, p, \cdot \rangle$	PSO	getLevel2Index(predicate)
QP6	$\langle s, p, ?o \rangle$	PSO	getType3Set(subject)
QP7	$\langle \cdot, p, ?o \rangle$	POS	getLevel2Index(predicate)
QP8	$\langle ?s, p, o \rangle$	POS	getType3Set(object)
QP9	$\langle ?s, \cdot, o \rangle$	OSP	getLevel2Index(object)
QP10	$\langle s, ?p, o \rangle$	OSP	getType3Set(subject)
QP11	$\langle \cdot, ?p, o \rangle$	OPS	getLevel2Index(object)
QP12	$\langle ?s, p, o \rangle$	OPS	getType3Set(predicate)

Table 2.2: The query patterns and the equivalent index operations.

For the SPO index, **Type1** is **S** (subject), **Type2** is **P** (predicate) and **Type3** is **O** (object). The SPO index therefore provides two operations. First, `getLevel2Index(Type1)` for a given subject (**Type1**) returns an index with corresponding predicates (**Type2**). Second, `getType3Set(Type2)` returns, for a given predicate (**Type2**) a set of objects (**Type3**). The SPO index is therefore able to answer two kinds of queries. First, the query $\langle s, ?p, \cdot \rangle$, where the subject is bound (known) and a list of predicates for the given subject is requested: this can be answered with the operation `getLevel2Index(Type1)`. Second, the query $\langle s, p, ?o \rangle$, where the subject and the predicate are bound (known), and a list of objects for the given subject and predicate is requested: this can be answered with the operation `getType3Set(Type2)`. Table 2.2 shows all single statement based query patterns and their index operation to answer the query.

2.5.2 Shared Indexes

The main advantage of the Hexastore indexing scheme is its ability to answer any single-statement query with a single index lookup. No merging or (table) joining is necessary to answer queries, no matter which RDF elements are unbound. This exhaustive indexing, however, comes at a price. Compared to other triple stores, the storage space demands are higher, as Hexastore trades query answering efficiency with space. However, some index data structures have duplicates. In table 2.2, QP2 and QP6 are the same ($\langle s, p, ?o \rangle$). These two queries retrieve a set of objects for a given subject and predicate. In the index operation, **Type1** and **Type2** can be switched to retrieve the same **Type3** set. Hence, to get a list of objects for a given subject and predicate, `getType3Set(Type2)` can be executed on the SPO or PSO index, only **Type1** and **Type2** have to be switched to receive the same **Type3** set of objects. The same principle holds true for the query pairs QP4 and QP10, and QP8 and QP12. Therefore in three indexes, the **Type3** sets can be shared. This results in a worst-case fivefold size increase, instead of a worst-case

```

SELECT ?s WHERE {
  ?s <knows> <Ben> .
  ?s <knows> <Nicki>
}

```

Listing 2.6: SQ5, SPARQL query

sixfold size increase for the index storage space, compared to triple stores with only one index. Experiments in [7] show, however, that the storage penalty compared to real-world triple stores is much lower. In their evaluation, a B-tree indexed MySQL triple store with indexes on each column used only 33% less storage.

2.5.3 Example Query

Consider the query in 2.6. It finds all subjects – or resources – that know both **Ben** and **Nicki**, or: this query returns a set of subjects that have triples with predicate **knows** and the objects **Ben** and **Nicki**. The query can be translated to our single-statement query form as a combination of $\langle ?s, \text{knows}, \text{Ben} \rangle$ and $\langle ?s, \text{knows}, \text{Nicki} \rangle$. Each of these two queries can be answered by the POS or OPS index (see table 2.2). The POS index is consulted for the first triple pattern $\langle ?s, \text{knows}, \text{Ben} \rangle$. The operation translates to `getType3Set(Ben)`, which returns the set of subjects: **Jack** and **Emma**. The second part of the query is $\langle ?s, \text{knows}, \text{Nicki} \rangle$. The same operation (`getType3Set(Nicki)`) returns the set of subjects: **Jack**, **Emma** and **Ben**. To solve the query, the two sets are intersected and result in the set of subjects: **Jack** and **Emma**. This is consequently the result of the query.

This query can be answered graphically as well by means of figure 2.3. The POS index is consulted for the predicate **knows**. The predicate **knows** points to an index with the corresponding objects **Nicki**, **Ben** and **Emma**, which are all objects that appear in triples with the predicate **knows**. To solve the query, the (subject) vectors are retrieved from the objects **Ben** and **Nicki** and intersected. Subjects that appear in both vectors are valid solutions to the query.

2.5.4 Dictionary Encoding

As explained in Section 2.1, RDF elements are commonly variable-length URIs or literals. Instead of storing these URI and literal strings in their full representation, all RDF elements (subjects, predicates and objects) are encoded to integer identifiers. This has two main advantages. First, RDF elements can be stored more efficiently due to their fixed-size nature. Second, RDF elements commonly appear in more than one triple and each time they are stored in an index, only the identifier integer is stored, and not the full representation. This saves more storage space the more the particular RDF element appears in the collection. This holds true only for URIs or literals that require more space than their identifier, which is usually the case.

The disadvantage to this approach is, that a mapping from URI or literal to its identifier, and a mapping from identifier to its URI or literal have to be maintained. Further, when answering queries with many results, all these results have to be converted from their identifier to their URI or literal representation.

Please note that Hexastore does not store a table of RDF triples. The identifiers are stored in the six indexes, and a mapping from identifier to URI or literal, and a mapping from URI or literal to its identifier is stored. Therefore, the indexes and the two-way dictionary are the database.

3

Rdfbox

Rdfbox is a native triple store and an implementation of the Hexastore indexing scheme. It is implemented in Python and Cython. The Cython programming language is discussed in Section 4.1 in more detail.

Rdfbox is designed to be modular and uses open source key-value stores as storage engines. The details on how the Hexastore vector storage is applied in a key-value store context are discussed in Section 3.2. First, however, the index backends and their key features are discussed.

3.1 Index Backends

The index backends are open source key-value stores. They all provide basic operations to store, retrieve and delete key-value records. Further, they allow to iterate over records sorted by key. Moreover, they allow to jump to a specific *key* or jump to a *key prefix* and iterate over the records in sorted order.

The index backends are discussed in the following. A comparison can be found in table 3.1.

Tokyo Cabinet Tokyo Cabinet[13] is a key-value store written in C. Keys and values may be either character strings or binary strings. It is further possible to iterate over the records, sorted by key. Records are stored in a B+ tree, hash table or fixed length array. These data structures are persisted in a file.

It is a successor to the GNU dbm (GDBM)¹ and QDBM². Tokyo Cabinet offers APIs for several programming languages, including C, Perl, Ruby, Java and Lua.

¹<http://www.gnu.org.ua/software/gdbm/>

²<http://fallabs.com/qdbm/>

	Tokyo Cabinet	Kyoto Cabinet	LevelDB	Redis
written in	C	C++	C++	C
licence	LGPL	GPL	BSD	BSD
sorted	yes	yes	yes	no
values	BCS	BCS	BCS	BCS, data structures
architecture	embedded	embedded	embedded	client/server
data structures	B+ tree, hash table, fixed-length array	B+ tree, hash table	Log-structured merge tree (LSM)	custom
persistence	file, IM	file, directory, IM	directory	file, IM

BCS = binary/character strings, IM = in memory

Table 3.1: The used key-value stores and their properties.

Kyoto Cabinet Kyoto Cabinet[14] is a successor to Tokyo Cabinet, written in C++. Its features are similar to Tokyo Cabinet, i.e. it is a key-value store, where keys and values may be either character or binary strings. Kyoto Cabinet allows to iterate over records, sorted by key. It offers to store records in a hash table or B+ tree. The records may be either stored in main memory, or persisted to a file or directory based structure.

Kyoto Cabinet offers APIs for C, Java, Python, Ruby, Perl and Lua.

LevelDB LevelDB[15] is a key-value store developed at Google. It is based on the ideas of BigTable[10] tablets and the log-structured merge tree[11]. Keys and values are arbitrary byte arrays and records are stored sorted by key. Further, LevelDB allows for efficient batch inserts.

Records may be compressed, using Google’s Snappy³ compression library. Snappy is designed to be fast at compression and decompression, but results are less efficiently compressed, compared to modern fast compression algorithms. LevelDB uses Snappy by default, claiming no significant performance loss, but using significantly less storage space.

Redis Redis[16] is a key-value store, based on the client-server model. Keys are arbitrary character or byte strings. Values may be either arbitrary character or byte strings as well, but Redis allows for more sophisticated data structures as values, such as hashes (dictionary structures), lists, sets and sorted sets.

Redis instances are accessed via TCP or Unix sockets, and communication between client and server uses a custom protocol. Redis includes a command-line client, which

³<http://code.google.com/p/snappy/>

triple name	subject	predicate	object	S ID	P ID	O ID
T1	Jack	knows	Emma	101	201	102
T2	Jack	knows	Ben	101	201	103
T3	Jack	knows	Nicki	101	201	104
T4	Jack	likes	Basketball	101	202	150
T5	Emma	knows	Ben	102	201	103
T6	Emma	knows	Nicki	102	201	104
T7	Emma	likes	Football	102	202	160
T8	Ben	knows	Nicki	103	201	104
T9	Ben	likes	Basketball	103	202	150
T10	Ben	likes	Football	103	202	160

Table 3.2: The original ten triples, with made-up three-digit identifiers.

allows to create, update and delete records and data structures with human-readable commands.

Redis may be operated to run in main memory only, or it can be configured to persist its records to a file.

The Redis backend had to be abandoned. The reasons are discussed in Section 4.4.

3.2 Mapping Hexastore Indexes to Key-Value Stores

Rdfbox implements the Hexastore vector storage on top of a key-value store. The mapping of Hexastore’s index vectors to a key-value store is depicted in figure 3.1. In this figure, the triple collection from Section 2.5 is reused. This time, the full SPO index is pictured. In the Hexastore index, all resources, the three subjects **Jack**, **Emma** and **Ben** are **Type1** in the SPO index. All three subjects form triples with the predicates **knows** and **likes** (**Type2**). However, this is arbitrary, as subjects could form triples with any predicates, and in real-world data, subjects would most likely not share the exact same set of predicates as they do in this example. In the bubbles next to the RDF elements their identifiers are marked. These identifiers correspond to the identifiers used in table 3.2. In this table, the original triple collection from Section 2.5 is reused and made-up dictionary-encoded triple identifiers are marked next to them.

The mapping from Hexastore vectors to key-value pairs works as follows. The first-level index just uses **Type1** identifiers as keys. The second-level index concatenates **Type1** and **Type2** identifiers and uses them as keys. The third-level sets are all **Type** identifiers concatenated. To navigate the indexes, the identifiers have to be concatenated. To “collect” second-level or third-level element identifiers, Rdfbox jumps to a key prefix and iterates over the records until the prefix changes. Because the keys are concatenated fixed-length identifiers, they may easily be extracted from the record keys.

In the SPO index as used by Rdfbox (figure 3.1), only identifiers are used. These are the same identifiers that are marked in the bubbles in the Hexastore index above. In the

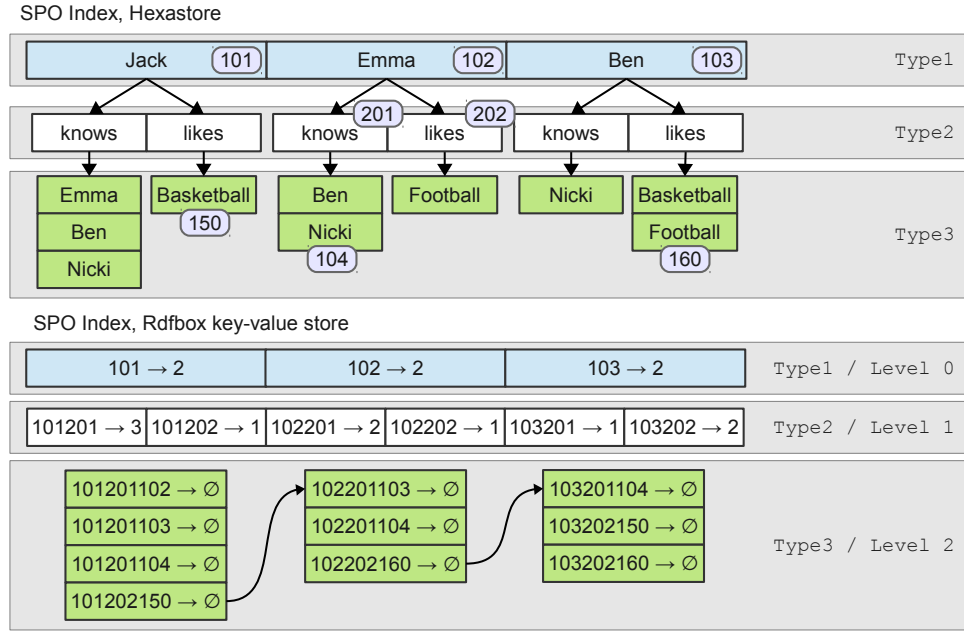


Figure 3.1: An SPO index, graphically represented as a Hexastore index (top) and mapped to a key-value store in Rdfbox (bottom).

Rdfbox index mapping, the SPO index is splitted into levels. These levels correspond to the **Types** used in Hexastore. **Type1** corresponds with level 0, **Type2** with level 1, and **Type3** with level 2. Each of these levels can be stored as separate indexes. Remember that the Hexastore vectors are mapped to key-value stores, therefore these levels are a collection of key-value records. In the figure, each record is symbolized as a green box, where each key on the left points to its value with the “→” symbol.

In the Rdfbox SPO index level 0, the subject identifiers appear as keys, with integers as their values. The values indicate the number of distinct **Type2** elements that are stored in the index vector belonging to this **Type1** element. This number is called the *cardinality*. In the example, all subjects (**Type1**) have two distinct predicates (**knows** and **likes**), therefore all level 0 records have cardinality 2.

In the SPO index level 1, the same principle applies. This time, however, the key is a concatenation of **Type1** and **Type2** identifiers. The values are cardinalities (the number of **Type3** elements) for the combination of subject and predicate. In the first record of level 1, the key is 101201, which is a concatenation of the identifiers for the subject **Jack** and the predicate **knows**. The cardinality (value) in this record is 3, since there are three objects for this combination of subject and predicate.

In the last level, the key is a concatenation of all three element identifiers, and since the cardinality for a whole triple is always 1, the value is unused. This is marked with the “∅” symbol. The first key in level 2 is 101201102, which is a concatenation of the element identifiers for the triple <Jack, knows, Emma>. The arrows in the index level

2 show that these records continue in the next column. If there was enough space, the boxes that symbolize the records would be placed next to each other, as they are in level 0 and 1.

By default, Rdfbox uses 8 bytes (=64 bits) for the (integer) identifier. Thus, when using 8 bytes for identifiers, in level 0 the keys are 8 bytes long, in level 1 they are 16 bytes long, and in level 2 they are 24 bytes long. In the example, however, all identifiers are three-digit numbers for the sake of simplicity. Thus, in level 1, concatenated identifiers span six digits, and in level 2 nine digits.

How are the Hexastore index operations applied to the indexes in Rdfbox? In Section 2.5.1, two different operations are defined on each index: `getLevel2Index(Type1)` and `getType3Set(Type2)`.

In the first operation, a set of `Type2` elements for the given `Type1` element is returned. In the SPO example, a list of predicates is returned for the given subject. To implement this operation in Rdfbox, the possibility to jump to a key prefix becomes important. To receive a list of `Type2` elements for a given `Type1` element, Rdfbox jumps to the key prefix that is the `Type1` identifier and then iterates over the records until the key prefix changes. In the SPO example, to retrieve a list of predicates for the subject **Emma**, Rdfbox jumps to the key prefix 102 and iterates over the records until the prefix changes (to 103). In these records Rdfbox is able to extract the identifiers for the (`Type2`) predicates, as the key is a concatenation of `Type1` and `Type2` identifiers.

The second operation (`getType3Set(Type2)`) works similar. To receive a set of `Type3` elements for given `Type1` and `Type2` elements, the level 2 index is used. The `Type1` and `Type2` identifiers are concatenated and form the key prefix to jump to. For example, to get a set of all objects for the subject **Emma** and the predicate **knows**, Rdfbox concatenates the subject and predicate identifiers to 102201. Then Rdfbox iterates over the records until the key prefix changes (to 102202). For each record, the object identifier is extracted from the key, as seen in the first operation. Of course, this indexing scheme based on the SPO index applies to all other indexes as well.

The important feature for the key-value based indexing backend is the ability to jump to a key prefix and iterating over the records *sorted by key*. These two features allow for a mapping of Hexastore vectors to a key-value store.

The sorted storage and iteration becomes important as well when executing queries with more than one statement. Consider this query:

```
SELECT ?name WHERE {
  <Emma> <knows> ?name .
  <Ben> <knows> ?name
}
```

Listing 3.1: SQ6, SPARQL query

This query in 3.1 finds objects that form a triple with the subjects **Emma** and **Ben** via the predicate **knows**. To execute the query, Rdfbox consults the SPO index level 2, jumps to the key prefix that form **Emma** and **knows** (102201) and extracts all object identifiers, which are 103 and 104 – the objects **Ben** and **Nicki**. Rdfbox then continues

with the second query statement and again consults the SPO index level 2, and jumps to the key prefix that form **Ben** and **knows** (103201) and extracts all object identifiers, which in this case is only 104, the identifier for the object **Nicki**. The result of the query are the objects that appear in both object sets, which is only **Nicki** in this case.

To intersect these two object sets, a simple merge-join is possible, since the two object sets are *sorted*. If the two object sets were unsorted, a sort-merge join would be necessary, or any other join-algorithm, which will be expensive for larger object sets than those in this simplistic case. This is an important characteristic for Hexastore vectors: the identifiers are always stored in sorted order, or at least the iteration sorted by key is efficient.

3.3 Query Plan Optimization with Cardinalities

Up until now, the level 0 index was unused, so were the cardinalities (values) in level 0 and level 1. These play an important role in the query execution optimization.

Reconsider query 3.1, where the query consists of two statements which are joined by the unbound object variable. Each of the two query statements has a set of objects as their result set. The result for the query as a whole is a merge-join of these two object lists. For two statements, this merge-join is straight-forward, as the object lists are retrieved from the index in sorted order. The engine that executes this query could first retrieve the object list for the first statement and then retrieve the object list for the second statement and merge-join them. The engine could also reverse the order and first retrieve the object list for the second statement, the end result is still the same and the merge-join is equivalent.

When adding a third statement to the query that also joins with the unbound *?var* variable, the order of execution is significant:

```
SELECT ?var WHERE {
  <subject1> <predicate1> ?var . # 1st statement, 1000 results
  <subject2> <predicate2> ?var . # 2nd statement, 100 results
  <subject3> <predicate3> ?var . # 3rd statement, 10 results
}
```

Listing 3.2: SQ7, SPARQL query

When executing the query in 3.2, the number of objects retrieved for each individual statement is important. The number of triples a query statement matches is called the *selectivity*. A high selectivity matches few triples in the graph, whereas low selectivity matches many triples in the graph. To optimize the execution of this query, the cardinalities stored in the indexes are used.

When starting with the statements that yields the least number of objects, subsequent merge-joins have to merge smaller lists, which is more efficient. For example, in the query 3.2, the objects are joined over the same variable, each of the statements individually yield a fixed number of elements that match the RDF data graph. This number is the cardinality for the given subject and predicate. The cardinality is in the range between

zero and the number of triples in the data graph. It is equal to zero if no triples match and equal to the number of triples in the data graph, if all triples match the statement.

Now assume that the first statement yields 1000 results, the second statement 100 results and the third 10 results. The final result for all three statements is the intersection of all these lists, which is minimum zero if no element is in all three result lists and maximum 10, if all elements from the third statement appear in the result list of the other two statements.

When merge-joining the first two statements, the resulting list of elements may have a size between 0 and 100, depending on how many elements intersect. These worst-case 100 elements then are merge-joined with the 10 results of the third statements, yielding between 0 and 10 elements as the final result. However, when merging these lists in reversed order starting with the third statement, a list of 10 elements is merge-joined with 100 elements from the second statement. These worst-case 10 elements then are merge-joined with the 1000 elements from the first statement, which again yields a maximum of 10 elements. Therefore the size of the intermediate, merge-joined result lists are minimized when starting with the statement that yields the least number of results.

To optimize the query execution plan, Rdfbox stores the cardinality for the combination of RDF elements coded in the key as the value of the record. In the index level 0, the keys are identifiers for the **Type1** elements it stores. The value (cardinality) therefore is the number of distinct **Type2** elements this **Type1** element yields. In level 1, the key is a concatenation of **Type1** and **Type2** elements and its value (cardinality) is the number of distinct **Type3** elements it yields. Derived from this are two operations:

$$getCardinality1(a) : \text{Type1} \longrightarrow \text{Int}(E)$$

$$getCardinality2(a, b) : \text{Type1}, \text{Type2} \longrightarrow \text{Int}(E)$$

Therefore, the first operation uses index level 0, the second operation uses index level 1. In the index level 2, the keys are a concatenation of all three element identifiers, which encodes a whole triple. The values are therefore unused and reserved for later use.

Applied to query 3.2, the number of results each statement yields can be answered with the operation `getCardinality2(Type1, Type2)` on the SPO index.

The first operation is used in the following query:

```
SELECT ?p WHERE {
  <subject> ?p ?var .           # first statement
  ?var <predicate> "a literal" . # second statement
}
```

Listing 3.3: SPARQL query

Either of these statements could match the least number of triples and would therefore be chosen as the first statement in the query execution plan. Note that the statements are joined with the `?var` variable. To build an execution plan, Rdfbox first analyzes

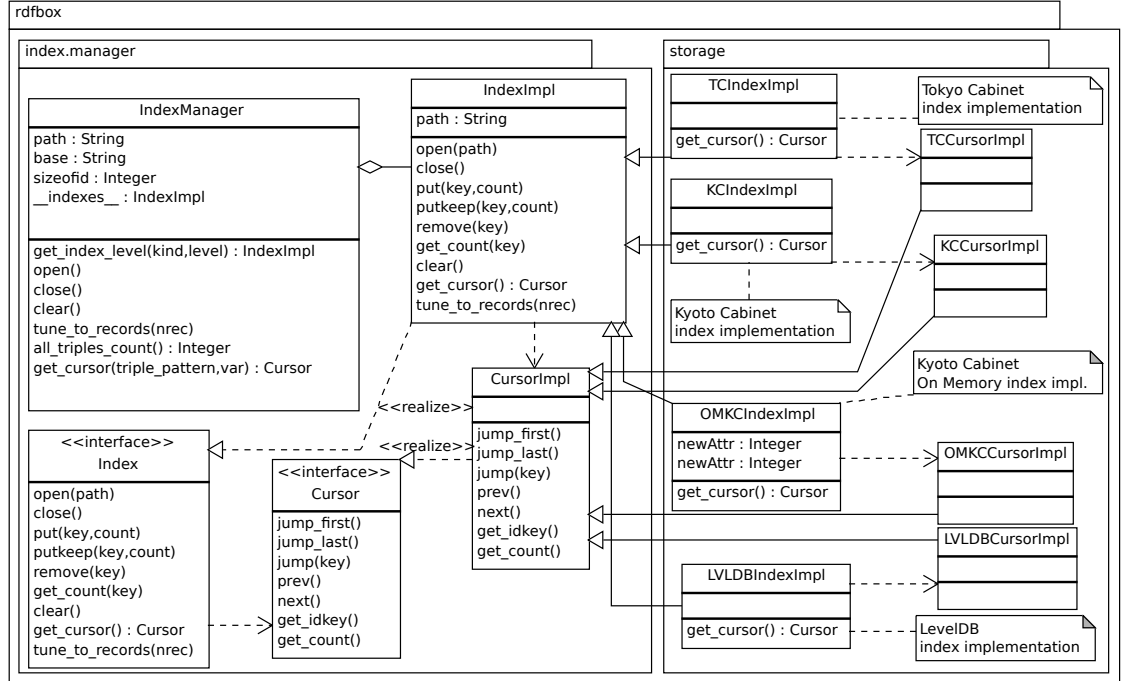


Figure 3.2: A stripped-down UML diagram of the Rdfbox indexing module with its backends.

the first statement. The cardinality for the predicate for the given subject is answered with `getCardinality1(subject)` on the SPO index. The cardinality for the object for the given subject is answered with `getCardinality1(subject)` on the SOP index. Rdfbox then continues with the second statement, where `getCardinality2(predicate, "a literal")` returns the cardinality for the subject. Either of these cardinalities could be the lowest and Rdfbox would then retrieve the elements and merge-join these with the list of elements with the next higher cardinality.

To summarize, query optimization is an important and non-trivial task for triple stores. The cardinalities stored in the Rdfbox indexes offer the possibility optimize the query execution plan. The order in which query statements are joined influences the size of intermediate results and thus are the starting point for optimization. The mechanism presented above is an insight to the mechanisms in Rdfbox. The work in [21] offers a more detailed discussion.

3.4 Rdfbox Architecture

In this section, the Rdfbox architecture relevant to the indexing is discussed.

The Rdfbox architecture for the indexing is depicted in the UML diagram in figure 3.2. The **IndexManager** is the interface to the indexing backend and is one of the central components in Rdfbox and offers a uniform access to the indexing backends.

The `IndexManager` holds an `Index` object for each index level, i.e. in theory three `Index` object for each index permutation (SPO, SOP, PSO, POS, OSP, OPS) resulting in 18 `Index` objects. However, three level 2 `Index` objects are shared (see Section 2.5.2) and access to them is relayed by the `IndexManager` and transparent to the client.

Each index level is a separate key-value store, represented by an `Index` object as an interface. Each `Index` offers, among others, operations to `open()` and `close()` the index, and to add and remove records (`put(key, count)` and `remove(key)`). The value for a key, which denotes its cardinality, is retrieved with `get_count(key)`. All records are removed with `clear()`. To optimize an index for the expected number of records to store, `tune_to_records(nrec)` is offered. The `Index` offers more methods, which are not relevant in this context and are left out of the UML diagram for clarity.

To iterate over records stored in the index level, `Index` objects offer a `Cursor` with `get_cursor()`. The `Cursor` is an interface as well and offers the methods `jump_first()` to jump to the first record in the index (sorted by key) and `jump_last()` to jump to the last record. With `jump(key)`, the cursor jumps to the first record that matches the key. With `prev()` and `next()`, the cursor jumps to the previous or next record. With `get_idkey()` the current record's key is retrieved, and its cardinality value is retrieved with `get_count()`. All methods return a boolean *True* if successful, and *False* if the operation was unsuccessful or the method is not implemented, except `get_idkey()` and `get_count()`, which return *NULL* if the cursor does not point to a record.

The key-value stores that are used as indexing backends must implement the `Index` and `Cursor` interface. The classes `IndexImpl` and `CursorImpl` act as base classes which implement the `Index` and `Cursor` interfaces. Therefore, indexing backends inherit from `IndexImpl` and offer a `CursorImpl` object via `get_cursor()`. The index backends discussed in Section 3.1 are depicted in the `storage` module on the right.

This section provides a broad overview to the Rdfbox architecture. Details are discussed in Section 4.

4

Implementation

This chapter focuses on the implementation details for the indexing backends and the triple loading. It begins with an introduction to the Cython programming language. Then, the configuration mechanism is discussed. After that, the indexing backend implementations are examined. This is followed by a discussion about the Redis and Tokyo Cabinet in-memory backend problems.

The last part discusses the triple loading workflow and its variants.

4.1 Cython

Rdfbox is implemented using the Cython programming language¹. Cython is a super-set of the Python programming language and Cython code is translated to C and then compiled by a C compiler. The compiled modules are accessible from Python code, as the translated Cython code makes use of the CPython API.

Cython is designed to provide two use-cases. First, Cython allows to write C extensions in Python. The idea is to speed up Python code by decorating it with C types. This is especially useful for looping constructs where a lot of run time is spent in. When implementing these in Cython and annotating them with proper C types, code often runs orders of magnitude faster than its Python equivalent[12]. Because Cython is a super-set of Python, code might be gradually optimized where needed, since pure Python is valid Cython code.

The second use case is to call native C and C++ code. As Cython is compiled to C code, native C code may be called from Cython, allowing C libraries to be included into Cython code. This second use case enables Rdfbox to seamlessly include indexing backends with C APIs and making these modules available to other Cython and Python code.

The listing in 4.1 shows a simple Cython code example. It defines a Cython class called `CythonClass` with three functions. The first function `normal_python_function` is regular, pure Python inside a Cython class.

¹<http://www.cython.org>

```

cdef class CythonClass:

    def normal_python_function(self, var1, var2):
        return var1 + var2

    cpdef cython_function(self, some_var):
        cdef int i
        cdef long result = 0
        for i in range(some_var):
            result += i
        return result

    cdef long cython_only_function(self, int some_var):
        cdef int i
        cdef long result = 0
        for i in range(some_var):
            result += i
        return result

```

Listing 4.1: A simple Cython code example

The second function `cython_function` uses `cpdef` instead of `def` to define a method. This denotes that the method is callable from Python as well as Cython code. The variables `i` and `result` are prefixed with `cdef` and their C type. The rest of the method is regular Python. This method may be optimized in parts, but wrapper code is generated for it to be callable from Python code.

The third function `cython_only_function` is fully decorated with C types. It uses the `cdef` keyword to mark the function as callable from Cython only. All variables are declared as C types. This includes the return type. This function may be translated to a C equivalent and can therefore be fully optimized.

4.2 Configuration

As discussed in Section 3.4, all `Index` objects are wrapped in `IndexImpl` classes and managed by the `IndexManager`. The index backend that is chosen is based on the `Rdfbox` configuration. The configuration file is a YAML file in which the backend configuration is defined. It may look like in listing 4.2.

The sample configuration in 4.2 uses Tokyo Cabinet as its default index backend. To use a different index backend, this value may be set to any of the supported backends, which is either `tokyocabinet` for Tokyo Cabinet, `kyotocabinet` for the file-based Kyoto Cabinet backend, `omkyotocabinet` for the on-memory-based (main memory) Kyoto Cabinet backend, or `leveldb` for the LevelDB backend.

The `index.storage` block allows for a more fine-grained setting of index backends. Each index and index level may be set to a custom backend. The configuration in the `index.storage` block works with prefix matching. The key is the index prefix that defines which backends to use for all indexes. The first line (key: 's') defines that all indexes starting with an S should be stored in a Kyoto Cabinet file-based index. These

```
# backend keywords:
# tokyocabinet, kyotocabinet, omkyotocabinet, leveldb
default_index_storage: 'tokyocabinet'

index_storage: {
  s: 'kyotocabinet',
  p: 'omkyotocabinet',
  o: 'leveldb',
  spo: 'leveldb',
  spo2: 'tokyocabinet',
}
```

Listing 4.2: A sample configuration file

indexes include all levels of the SPO and SOP indexes. The same principles applies to the configuration with the keys 'p' (all levels of POS and POS indexes) and 'o' (all levels of OPS and OSP indexes). The last two lines, however, override some of the settings made with the 's' key for the SPO and SOP indexes. The 'spo' key sets all levels of the SPO index to LevelDB. Further, the 'spo2' key sets the SPO index level 2 to Tokyo Cabinet.

As a general rule applies: settings made by longer, more specific keys override shorter, less specific keys. If no specific setting for an index level is made in the `index_storage` block, the default index backend is set.

The YAML configuration file is directly mapped to a Python object, in which the top-level keys are attributes for the configuration object. Key-value blocks like `index-storage` are mapped to a dictionary.

If no configuration is detected, the `IndexManager` chooses Tokyo Cabinet as a backend. This also applies if an index backend is misspelled or any other errors occur.

4.3 Indexes

As discussed in Section 3.4, all index backend implementations follow the `Index` and `Cursor` interface, by inheriting from `IndexImpl` and `CursorImpl` respectively. To implement this interface, relevant calls are made to the actual C API of the key-value store. In this section, the APIs to these key-value stores are discussed as well as their mapping to the `Rdfbox` interface.

4.3.1 Tokyo Cabinet

The Tokyo Cabinet index backend can be found in the `rdfbox.storage.tokyocabinet.index` package. The index is implemented in the `TCIndexImpl` class and its cursor is

implemented in the `TCCursorImpl` class. The design of the Tokyo Cabinet backend, which was part of `Rdfbox` before this work, influenced the design of the other backends.

The `TCIndexImpl` object has to mediate between the `Rdfbox` object-oriented world and the C API of Tokyo Cabinet. A new B+ tree data structure is created with `tcdbnew()`, which returns a pointer to the `TCBDB` struct. This struct pointer is held by the `TCIndexImpl` object and used for subsequent calls. This struct is then used for other C API methods to manipulate the tree data structure. Because the data structure is file-based, it has to be opened first with `tcdbopen(TCBDB* tcb, char* path, int omode)`. The first parameter is the struct returned by `tcdbnew()`, the second parameter is the string to the file path and the third parameter is a flag parameter. The flag parameter uses an integer in which options may be enabled and disabled bit-wise. The flags provided indicate whether to open the data structure read-only, whether locking should be enabled and if the file should be created if it does not exist. The API details can be found in the “Fundamental Specification of Tokyo Cabinet”². The Tokyo Cabinet indexes have a `.tcb` file suffix.

To iterate over records and jump to a key or key prefix, a cursor has to be created with `tcdbcurnew(TCBDB* bdb)`. This method returns a pointer to a `BDBCUR` struct, which represents the cursor and is held by the `TCCursorImpl` object. To jump to a key prefix, the method `tcdbcurjump(BDBCUR* cursor, void* key, int key-size)` is used. The key may be any data type (cast to a void pointer), provided with the key size. The method `tcdbcurnext(BDBCUR* cursor)` jumps to the next record.

These API calls are just a selection of some of the index and cursor interface, to give a broad overview on how the index and cursor interfaces are translated to the C API.

4.3.2 Kyoto Cabinet

The Kyoto Cabinet index backend is structured similarly to the Tokyo Cabinet backend. Because Kyoto Cabinet is a successor to Tokyo Cabinet, the two C APIs are very similar.

The index and cursor implementations can be found in the `rdffox.storage.kyotocabinet.index` package as `KCIndexImpl` and `KCCursorImpl`.

The Kyoto Cabinet API uses polymorphic databases. Therefore access to all kinds of data structures is achieved with the same API methods. A new data structure is created with `kcdbnew()`, which returns a pointer to a `KCDB` struct. This could be any data structure Kyoto Cabinet offers. Not until the data structure is opened with `kcdbopen(KCDB* kcdb, char* path, int omode)`, the data structure used is known. The data structure is chosen by file path suffix. The suffix `.kct` indicates a file tree data structure. The `omode` integer uses the same flags mechanism as in Tokyo Cabinet. Other parameters may be appended to the file path with the “#” symbol, followed by a list of `key=value` pairs. Pairs are separated with the “#” symbol. The Kyoto Cabinet files have a `.kct` suffix.

A cursor for the data structure is created with `kdbcursor(KCDB* kcdb)`, which returns a pointer to a `KCCUR` struct. The API is equivalent to the Tokyo Cabinet API.

²<http://fallabs.com/tokyocabinet/spex-en.html>

The method `kccurjumpkey(KCDB* kcdb, void* key, int key_size)` jumps to a key prefix, and `kccurstep(KCDB* kcdb)` jumps to the next record.

The mechanisms on how to translate the index and cursor interfaces to Kyoto Cabinet API calls is similar to the Tokyo Cabinet API. The Kyoto Cabinet API documentation³ provides more details on the API usage.

4.3.3 Kyoto Cabinet On Memory

The Kyoto Cabinet On Memory backend can be found in the `rdfbbox.storage.kyotocabinet.omindex` package. The index implementation is `OMKCIndexImpl` and inherits from the Kyoto Cabinet `KCIndexImpl`. The index implementation shares all interface implementations, but overloads the `open()` and `close()` methods to use an on-memory tree data structure. Instead of using an actual file path for the `kcdbopen(...)` method, the file path is `"%"`. The percentage symbol indicates the on-memory tree data structure. Additional parameters may be added with the `"#"`, followed by `key=value` pairs, equal to the Kyoto Cabinet file-based usage.

The on-memory tree structure needs to be persisted nevertheless. Kyoto Cabinet allows the on-memory based data structures to be saved as *snapshots*. These snapshots may be loaded and saved to files. Therefore, in the `open()` method, the snapshot is loaded into memory. If the file does not exist, the loading request is ignored by Kyoto Cabinet. In the `close()` operation, the snapshot is persisted to disk. The snapshot files have a `.kcbdump` suffix.

Other than loading and saving snapshots and the special file path, the Kyoto Cabinet file-based and on-memory index backend are identical. Thus, the on-memory implementation inherits from the file-based implementation and only overloads the `open()` and `close()` method. The cursor implementation is identical.

4.3.4 LevelDB

The LevelDB index backend can be found in the `rdfbbox.storage.leveldb.index` package. The implementation is `LVLDBIndexImpl` and uses the LevelDB C API. The index is persisted to a directory and may be opened with `leveldb_open(leveldb_options_t* options, char* path, char** error_message)`. The first parameter is a struct that defines options and parameters for the LevelDB index. The second parameter is the index directory path and the third parameter is a string pointer to an error message, if an error occurs. Otherwise the pointer is `NULL`. The open method returns a pointer to a `leveldb_t` struct. The index directories have a `-leveldb` suffix.

The cursor concept is called iterator and created with `leveldb_create_iterator(leveldb_t* db, leveldb_readoptions_t* options)`. Each iterator is configured with its own "read options". The method returns a pointer to a `leveldb_iterator_t` struct. The method `leveldb_iter_seek(leveldb_iterator_t* iterator, char* key, size_t key_size)` jumps to a key prefix. The method `leveldb_iter_next(leveldb_iter`

³http://fallabs.com/kyotocabinet/api/kclangc_8h.html

index level	key	value (list)	list index tuples	concatenated keys
SPO level 0	spo_subj	[101, 102, 103]		
SPO level 1	101	[201, 202]	(0, 0)	101201
	102	[201, 202]	(0, 1)	101202
	103	[201, 202]	(1, 0)	102201
SPO level 2	101201	[102, 103, 104]	(1, 1)	102202
	101202	[150]	(2, 0)	103201
	102201	[103, 104]	(2, 1)	103202
	102202	[160]	(A, B)	A = list index, level 0 B = list index, level 1
	103201	[104]		
	103202	[150, 160]		

Table 4.1: A Redis SPO index.**Table 4.2:** Iterating over the SPO level 1 index in Redis. The tuple denotes the list index for the current key.

`ator_t* iterator)` jumps to the next record. Iterators have to be destroyed with `leveldb_iter_destroy(leveldb_iterator_t* iterator)`, otherwise closing the index fails.

Unfortunately, LevelDB does not offer a mechanism to count the records, which is part of the **Index** interface. To keep track of the number of records, a naive approach would keep a list of all keys stored in the index, which would almost double the index size and is therefore not feasible. Cardinality counters provide a solution to this problem. All keys added to the index are also added to the cardinality counter, which returns an estimate of the number of keys added. Cardinality counters trade memory size with estimate accuracy. A state of the art cardinality counter is the HyperLogLog[17] counter. In an index with 160 million 24-bytes keys, the serialized HyperLogLog counter takes 7 kilobytes, which is negligible compared to the index size of 2.1 gigabytes.

4.4 Problems with Index Backends

Unfortunately, not all index backends could be successfully added to Rdfbox. The Redis backend turned out to be unsuitable, mainly because Redis does not allow to iterate over records sorted by key. Tokyo Cabinet also offers an in-memory tree data structure with a similar API to the file-based tree, but did not allow for multiple cursors on the same tree data structure.

In the following, the issues with these backends are examined.

4.4.1 Redis

Applying the Hexastore concept to a Redis key-value store has proven difficult. Using Redis in Rdfbox has two main disadvantages.

First, Redis is built as a server, and to store records, a connection to the Redis server has to be established. This connection is either via TCP or Unix socket. Although Unix sockets enable a connection with less overhead compared to TCP sockets, performance is still poor compared to a solution that runs in-process. Unfortunately, Redis is not embeddable and communication is limited to TCP or Unix sockets.

The second main disadvantage is that records in a Redis database may not be iterated sorted by key, which is essential for index cursors. Because the iteration over key ranges and consequently, jumping to a key prefix is not supported, the Hexastore indexes have to be mapped differently to a Redis key-value store.

As explained in Section 3.1, Redis allows character or binary strings as keys, and character or binary strings as values. Values may be data structures as well, such as lists. These lists are used to map Hexastore indexes to the Redis key-value paradigm. In table 4.1, the example triples from table 3.2 are demonstrated in a Redis SPO index. In the SPO index level zero, only one records exists, which is at the same time the entry point to the SPO index itself. In this record, which uses a well-known, configurable key (in this case, “spo_subj”), all the subjects are stored. Remember that subjects are **Type1** in the SPO index. In this record, all SPO **Type1** identifiers are stored. Each of these subject identifiers is a key in the SPO level 1. There, each subject identifier key stores a list of **Type2** identifiers (in this case predicates) that belong to this **Type1** subject. In level 2, for each combination of subject identifier and predicate identifier, a concatenated key is stored in the SPO index level 2. The value is a list of **Type3** identifiers (objects) that belong to this **Type1** and **Type2** element, i.e. the object identifiers for the given subject and predicate. To summarize, for the SPO index, in level 0 all the subject identifiers are stored in a list with a well-known, configurable key. In level 1, all the subject identifiers are keys, and each key stores a list of predicate identifiers. In level 2, the keys are a concatenation of subject and predicate identifiers. They each store a list of object identifiers. Of course, the same principle applies for all other indexes.

For example, the triple <Emma, likes, Football> the identifiers are <102, 202, 160>. The subject can be found in the SPO level 0 values. In level 1, each subject identifier has its own key with the predicate identifiers list as their value. The subject 102 has the predicates 201 (“knows”) and 202 (“likes”). In level 2, The concatenation of subject identifier and predicate identifier (102202) serves as the key, with a list of object identifiers as its value. In this case, the only object is 160 (“Football”).

To offer a cursor to the index level 0 is relatively straightforward. Redis offers a command to return a list value by index. This enables iterating over the level 0 index by requesting list values by index. First, the list size for the key “spo_subj” has to be determined. In the example in table 4.1, the list size is 3.

To iterate over all subjects, the list index is saved in the cursor, and for the `next()` operation, this list index is incremented until the list index reaches the list size, in which case the end of level 0 is reached and `next()` returns *False*. Therefore to iterate over all

level 0 elements, the list size has to be determined, which is a round-trip to the server and then for each `next()` a round-trip to the server is used to receive the next `Type1` identifier.

For the index level 1, the procedure becomes more complex. Now the subject list from level 0 has to be combined with the records from level 1. To iterate over all level 1 records, the list size for “spo_subj” (level 0) has to be determined. Next, the first identifier from this list is retrieved. This identifier serves as the key for the list in level 1. Then, the size of this level 1 list is determined and the first identifier from the list is retrieved as well. These two identifiers are then concatenated and form the first element for the level 1 index.

In table 4.2, the iteration over the SPO index level 1 is illustrated. Note that, compared to level 0, in level 1 two list indexes have to be maintained by the cursor. These list indexes are depicted as tuples. The first tuple element (“A”) is the list index from level 0 (“spo_subj”), the second tuple (“B”) element is the list index for level 1. In this table, the complete level 1 iteration is depicted, with the resulting key concatenations that are returned by the cursor `get_idkey()` operation. Note that the first tuple element runs over the values in the “spo_subj” list until the end reaches. The second tuple element runs over the three level 1 lists, which are each of size 2. For every list overflow, the size for the next list has to be determined. To iterate over all elements in this example, there are 4 round-trips to determine the list sizes plus 6 round-trips to retrieve the identifiers that have to be concatenated.

For the level 2 index, the same principle applies, but the list index tuple has three elements. To iterate over all the SPO index level 2, 10 round-trips are necessary to determine list sizes, and 19 round-trips to retrieve identifiers to be concatenated.

A Redis index was built on a small dataset containing 481 triples and a simple query, that executed in split seconds on a Tokyo Cabinet index took about 20 seconds with the Redis index. Optimizations that tried to pre-fetch batches of identifiers from identifier lists improved the performance significantly to the range of several seconds, but was still orders of magnitude worse compared to Tokyo Cabinet.

4.4.2 Tokyo Cabinet In Memory

Tokyo Cabinet offers an on-memory B+ tree data structure that can be persisted to disk. The iteration over records, however, is solved differently to the file-based API. Instead of returning a C struct for the cursor, the `TCNDB` struct, which is the tree struct itself, is used. The state of the cursor is therefore directly stored in the tree data structure itself. Therefore only one cursor may be used simultaneously, which makes this data structure unusable for the purpose, as `Rdfbox` may use more than one cursor for an index simultaneously.

A work-around which copied the `TCNDB` struct failed, because a deep-copy including all records would have been necessary. This would defeat the purpose, since copying a complete index level for a cursor is not feasible.

Therefore support for a Tokyo Cabinet in-memory backend had to be dropped.

4.5 Unit Testing Indexes

The index backends were all unit tested in isolation and written in Python. To test index backends in isolation, however, it was not possible to directly manipulate the `IndexImpl` objects, because the methods were written to be accessed from Cython code only for performance reasons. Therefore, a layer between the Cython index objects and the Python unit tests had to be written. The solution was to create proxy objects that are written in Cython. The proxy objects implement the same `Index` interface, but provide these methods in an API that is accessible to Python code as well. These proxy objects are not as fast as the index objects themselves, but performance was not a criterion in the unit tests.

As a first step, the existing Tokyo Cabinet index backend was taken as a template and was unit tested. These tests were later used for the other backends as well. Following this method, it was possible to capture the index and cursor behavior of Tokyo Cabinet and therefore to ensure that other backends followed the same behavior.

In the unit tests, three index objects are instantiated, that represent level 0, 1 and 2 for the SPO index. The test data is the same as presented in table 3.2. The data involves only ten triples, which is extremely little, but has a significant advantage: the amount of triples is small and can easily be memorized. Therefore the index integrity can be examined easily in unit tests. Even though the test data is small, the triples cover essential features of RDF data, such as more than one predicate per subject and objects that are reused as subjects. Further, some objects only appear as such and thus resemble literals.

The unit tests reuse the identifiers assigned in table 3.2. The identifiers are therefore three-digit numbers. Subjects are assigned a number between 101 and 104, predicates use a number that starts with the digit 2 and objects that are not reused as subjects use a number that is higher or equal to 150 and lower than 200 to not be confused with predicates. This numbering scheme enables to memorize RDF elements easily while still providing a fixed-size identifier.

These triple identifiers are tested in two versions. The first version uses ASCII characters as identifiers, i.e. identifiers such as “201” are stored as number characters. The second version uses binary identifiers, i.e. identifiers such as “104” are stored as three-byte identifiers, in which every digit’s value is represented as a byte. Because most of the identifiers have a zero-byte in the middle, these identifiers should catch errors where the middle zero-byte is interpreted as the end of a string by accident. This is an important aspect, because the indexes are implemented in Cython code that makes heavy use of C methods and APIs, where the distinction between character strings and byte strings is a common source of errors. Rdfbox uses 8-bytes integer identifiers by default, which may have zero-bytes as well.

All unit tests are therefore performed twice, first on the ASCII identifiers and then on the binary identifiers. The unit tests covered basic index operations, such as putting records into the index and retrieving them, but focus on the cursor. The unit tests cover each index level’s cursor thoroughly, including jumping to full keys and key prefixes. This ensures that all index operations works correctly, i.e. retrieving `Type2` elements

for a given **Type1** and retrieving **Type3** elements for given **Type1** and **Type2** elements. Further, cardinalities on each level are checked for integrity.

The unit tests are designed to be easily extensible for new indexing backends. New indexing objects need to provide a proxy object. These are easily created by inheriting from a proxy base class that provides an implementation for most of the index methods and may be customized where needed. New index unit tests inherit from `RdfboxTestHelper` and instantiate all three index level proxy objects and provide them via the `get_indexN()` methods.

This unit test design ensures that the tested index operations are the same for any index backend and that new backends may easily be checked for conformity with existing index backends.

4.6 Loading Triples

In this section, the process of encoding, sorting and inserting triples into the indexes is discussed.

This process is divided into two phases: phase one is encoding and sorting, and phase two is merging and loading into index.

The triples are loaded into the index in encoded form, i.e. triple elements (subject, predicate or object) are not in their URI or literal form, but encoded as 8-bytes integer identifiers. The encoding was discussed in Section 2.5.4. 8-bytes integers is the default identifier size, which may be configured. The triples are sorted because the Tokyo Cabinet and Kyoto Cabinet are B+ tree data structures, which perform best with sorted input. The LevelDB backend uses a log-structured merge tree. The performance implications of sorted and unsorted inputs for LevelDB are discussed in chapter 5.

4.6.1 Encoding and Sorting

Figure 4.1 provides an overview to phase 1. The triples are stored in their standard SPO form and `Rdfbox` reads them in and parses them. All RDF elements are then encoded to their integer identifiers. At the end of this step, the triples are in their encoded SPO form. In the next step, the encoded triples are permuted to all six permutations. In figure 4.1, the permutation to the OPS form is depicted. After the triples are permuted, each triple is concatenated. Since the triples are in their encoded form, the concatenated triple is 24 bytes long, given that RDF elements are encoded with 8-bytes identifiers. This list of concatenated triples is then sorted and stored to a file. In the sorted file, the triples are stored continuously, without any extra characters that separate them. It is assumed that the identifier size is known, as well as the triple permutation, which is encoded in the file name.

The work-flow in figure 4.1 is, however, just an overview to the general process in phase 1. Figure 4.2 depicts the phase in its actual form. The triples may span multiple input files. If there are multiple files, they are treated as one input source and split into chunks. The file chunk size may be configured. These file chunks are then put

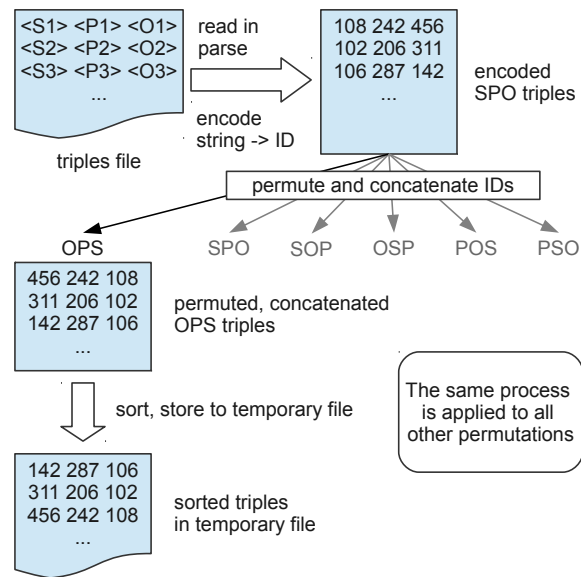


Figure 4.1: An overview to the sorted loading work-flow. It starts on the top left.

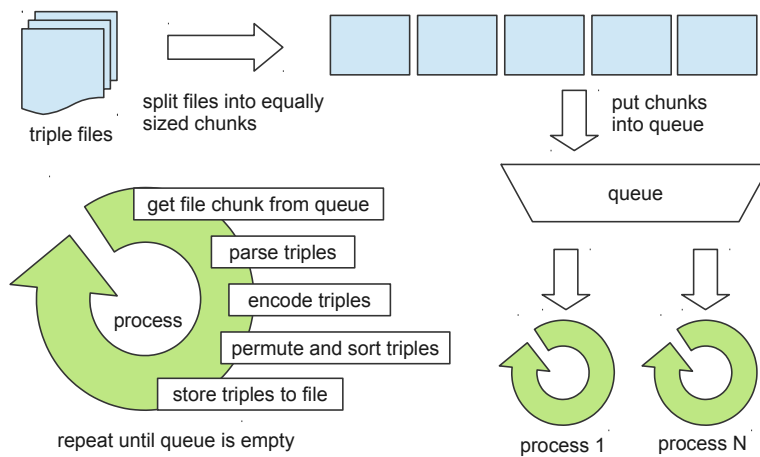


Figure 4.2: The sorted loading work-flow, executed in parallel by a process pool. The process is the same as in figure 4.1, but executed by many processes in chunks.

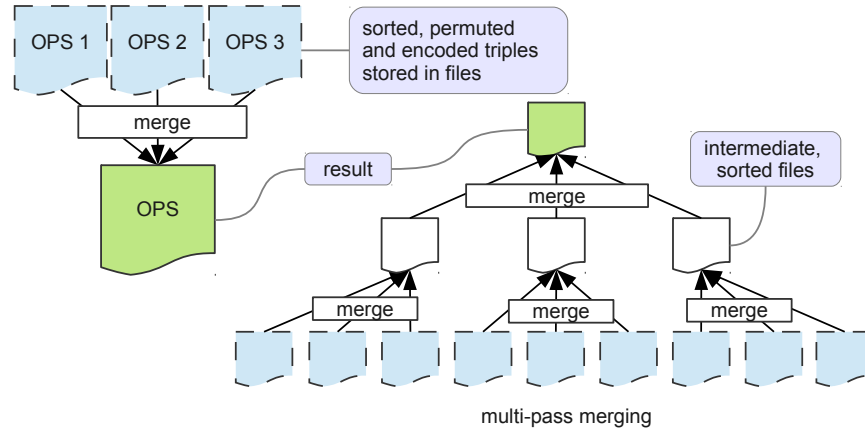


Figure 4.3: The second part of the sorted-loading process: merging all sorted, temporary files to 6 resulting files for each index.

into a queue. Note that they, however, consist of the file path and a range of bytes that represent the file chunk, not the actual content of the file. After that, a pool of processes is started which begins to process the queue. The number of processes may be configured as well, but is by default equal to the number of CPUs the host system provides. Each process performs the steps discussed in figure 4.1. Therefore, for each file chunk in the queue, six sorted, encoded, temporary triple files are produced. These temporary files are the input for the next phase.

The first phase of loading triples is processed by a pool of processes. Processes are used instead of threads to scale to multiple CPUs, as Python uses the Global Interpreter Lock, which limits threads to run on a single CPU[18]. Access to the module that uses dictionary-encoding is guarded by a lock. This is necessary, because dictionary encoding has to provide the same identifier for the same resources. If two or more processes access the dictionary encoding module, it can not be guaranteed that two resource URIs are assigned the same identifiers. Therefore, access to the dictionary encoding is locked.

4.6.2 Merging and Loading into Index

The first phase produces for each tuple permutation at least one file with sorted, encoded triples. These triple files have to be merged into one, large triple file for each index, which is part of phase 2. The second phase is depicted in figure 4.3. At the end of phase two, six files with permuted, sorted and encoded triples are ready to be loaded into the indexes.

If the first phase produced a lot of temporary triple files, multi-pass merging is necessary. This multi-pass merging is depicted in figure 4.3 at the bottom. The multi-pass merging works as follows. The number of files N to merge in one iteration has to be defined. In the figure, the number of files to merge is $N=3$. In practice, this number is higher. Then, all temporary files that belong to the same permutation have to be found. These temporary input files are defined to be in level 0. From these, the first

N are merged and produce a merged file in level 1. The next N are again merged and produce yet another file in level 1, and so on. After all initial files are merged in batches of size N , a number of merged, temporary files are in level 1. If the number of files in level 1 is equal or less than N , these files are merged to produce the final, sorted triples file. If there are more than N files in level 1, the same procedure continues until a level is reached where there are less than (or equal to) N files.

The optimal number N is not automatically calculated at the moment. It is, among other factors, certainly determined by the maximum number of files that may be opened per process defined by the underlying operating system. An optimal solution is still to be done.

After the multi-pass merging, all the initial temporary files and all intermediate-level temporary files can be deleted. In fact, the temporary files may be deleted as soon as they are merged in to a higher-level temporary file.

4.6.3 Iterative Loading

The two phases described above are necessary for the sorted input into indexes which are based on B+ trees. For other indexes, a sorted input may not be necessary. In this case, iterative loading is possible. The process is similar to the process in figure 4.2. The triple input is divided into chunks and put into a queue. A process pool starts to process the chunks, i.e. reading the triples in and parsing them. After that, they are encoded and permuted. After permutation, however, triples are not sorted and stored to a file, but directly put into the index.

Phase two is completely skipped, as there is no sorting involved and therefore no temporary files are written. However, the dictionary encoding problem remains. The triple encoding is still guarded by a lock, to avoid that the same resources are assigned to different identifiers.

5

Evaluation

To evaluate the index backends, two aspects were considered. First, the performance when loading triples. Second, the performance when executing queries.

The Tokyo Cabinet und Kyoto Cabinet backends are built using B+ trees. Therefore, the triples loaded into the index have to be sorted. For each of the six indexes, a file with encoded, permuted and sorted triples is loaded into these backends. In the evaluation, the default identifier size of 8 bytes was chosen. This allows for $2^{64} \approx 1.8 * 10^{19}$ different RDF element identifiers.

Three differently sized data sets were chosen to evaluate the performance. These data sets were loaded into the indexes. For each of these data sets, a set of queries were executed to compare their querying performance.

5.1 Datasets and Queries

The data sets were taken from the FedBench[19] research project. They are listed in table 5.1.

The goal of FedBench was actually to provide a benchmark for federated queries. The idea is to test query engines that execute queries over distributed data sets. The FedBench data sets are therefore designed to be distributed to many triple store instances and to evaluate distributed query execution strategies. The data sets and queries are, however, suited for centralized queries as well. Because the FedBench data sets and queries are widely used, they were chosen for this evaluation.

The FedBench data sets consist of many different, collected data sets. The individual data sets span a variety of domains. These individual data sets are grouped to form larger data sets.

The smallest group is a single data set from the SP²Bench SPARQL Benchmark[20]. SP²Bench provides a data generator which can produce arbitrary large data sets. A generated data set with 10 million triples is part of FedBench and forms the first data set group, which is referred to as “sp2bench” or “sp2b” in this work.

Dataset Group	Dataset	Domain	Triples (mill.)
SP2Bench	SP2B-10M	Bibliographic (synthetic)	10
Life Science 52.8 M triples	DBpedia subset	Generic	43.6
	KEGG	Chemicals	1.09
	Drugbank	Drugs	0.767
	ChEBI	Compounds	7.33
Cross Domain 159.2 M triples	DBpedia subset	Generic	43.6
	NY Times	News about people	0.335
	LinkedMDB	Movies	6.15
	Jamendo	Music	1.05
	Geonames	Geography	108
	SW Dog Food	SW Conf. and publ.	0.104

Table 5.1: The FedBench data sets. Each Data set group consists of one or more individual data sets. The groups are named after their data domain. The smallest group consists of 10 million triples, the largest of almost 160 million triples.

The second group consists of data sets from the life science domain. These are real-world, non-synthetic data from the chemistry and biology domain. Added to this group is also a subset of the DBpedia data set. All data sets from this group combined consist of 52.8 million triples. This data set is referred to as “life-science”.

The third and largest group is the cross domain data set. It includes the same DBpedia subset used in the life-science group and adds five other data sets: a New York times data set (news about people), LinkedMDB (movies), Jamendo (music), GeoNames (geographical and spatial data), and Semantic Web Dog Food, a small data set on Semantic Web conferences and publications.

The two largest individual data set are DBpedia with 43.6 million triples and GeoNames with 108 million triples. The DBpedia subset is used in the cross-domain group as well as the life-science group.

To summarize, FedBench provides three data set groups. The smallest group is *sp2bench* with 10 million triples. The second group is *life-science* with 52.8 million triples and the largest group is *cross-domain* with 159.2 million triples.

FedBench provides a set of SPARQL queries for each of these groups. A complete listing of queries may be found in the appendix A.1. The queries used are, however, not all equal to the FedBench queries, because Rdfbox does not support all SPARQL features yet. The queries were, however, rewritten to equivalents where possible or simplified.

The queries are named after their groups. For *sp2bench* data set, the queries are S-Q1 to S-Q4, for the life-science data set, the queries are L-Q1 to L-Q11 and for the cross-domain data set, the queries are C-Q1 to C-Q11.

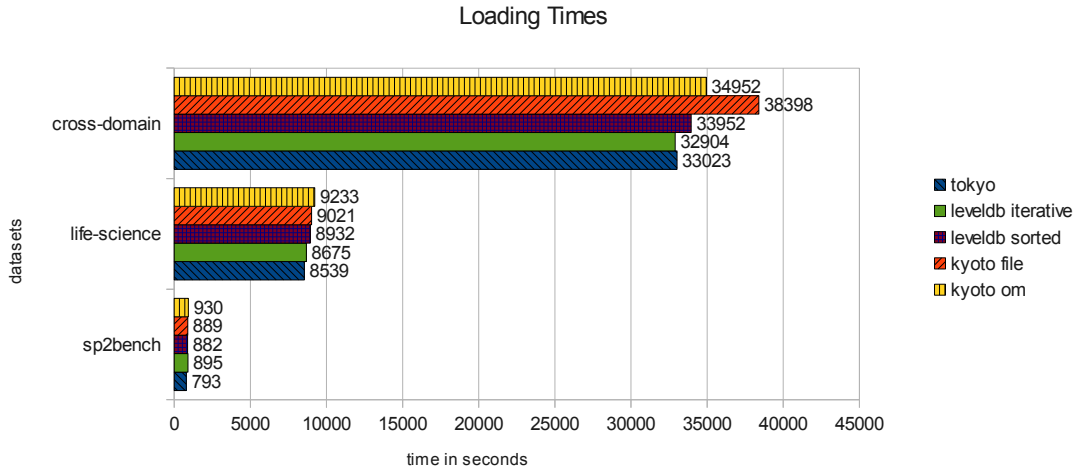


Figure 5.1: The loading time in seconds for the three data sets, plotted for each backend. LevelDB is loaded twice, with sorted input and iteratively.

5.2 Loading Data

Test Setup The evaluation was performed on an 8-core 2.933 GHz Intel machine with 72 GB of main memory running Linux. It was setup with Python 2.7.2, Cython 0.17.1, and GCC 4.4.3.

Loading FedBench Data Loading the data was performed with sorted data input on all four index backends. Additionally, the triples were loaded iteratively for the LevelDB backend. Figure 5.1 shows the results for all three FedBench data sets.

For the sp2bench and life-science data sets, Tokyo Cabinet outperforms all other backends. For sp2bench, Tokyo Cabinet takes 10% less time than the second-fastest backend (LevelDB, sorted input) and 14% less time than the slowest backend (Kyoto Cabinet on-memory).

For the life-science data set, Tokyo Cabinet takes 1.5% less time than the second-fastest backend (LevelDB, iterative input) and 7.5% less time than the slowest backend (Kyoto Cabinet on-memory).

For the cross-domain data set, the figures changed. The iteratively loaded LevelDB backend wins by a small margin. The second-fastest is Tokyo Cabinet, followed by LevelDB with sorted input and Kyoto Cabinet on-memory. The slowest backend was the file-based Kyoto Cabinet, which takes 14% more time than Tokyo Cabinet.

Loading GeoNames Data To further analyze how loading times change with increasing data sizes, the GeoNames data set was split up into 10 million triple chunks. Data was then loaded into indexes in 10 million triple increments. The results are plotted

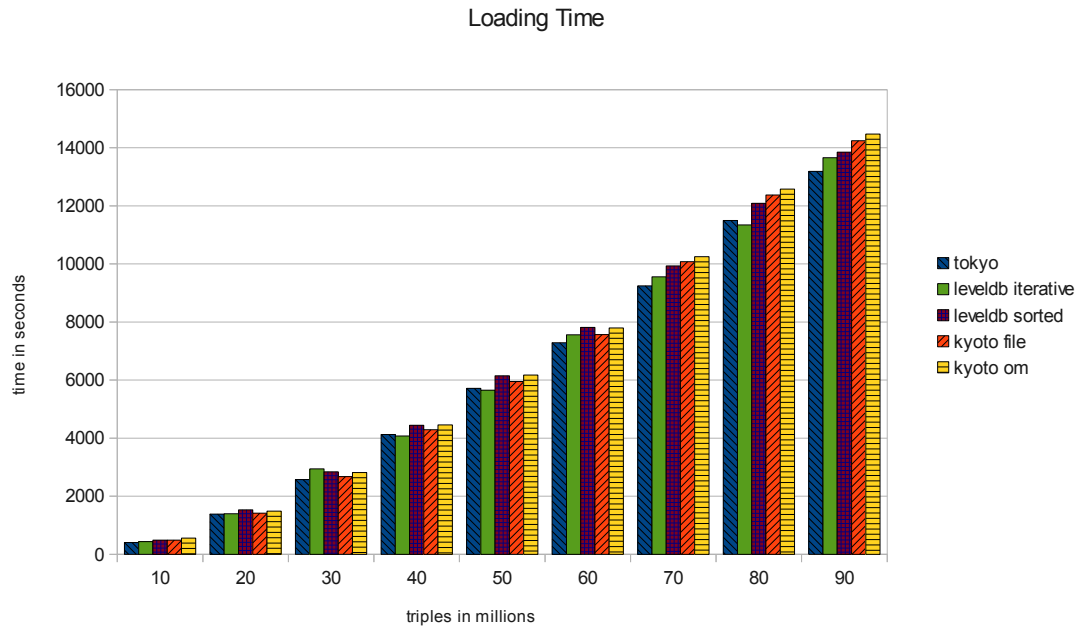


Figure 5.2: The loading time for the GeoNames data set. The data is split in 10 million triple chunks to analyze the backend behavior with an increasing number of triples.

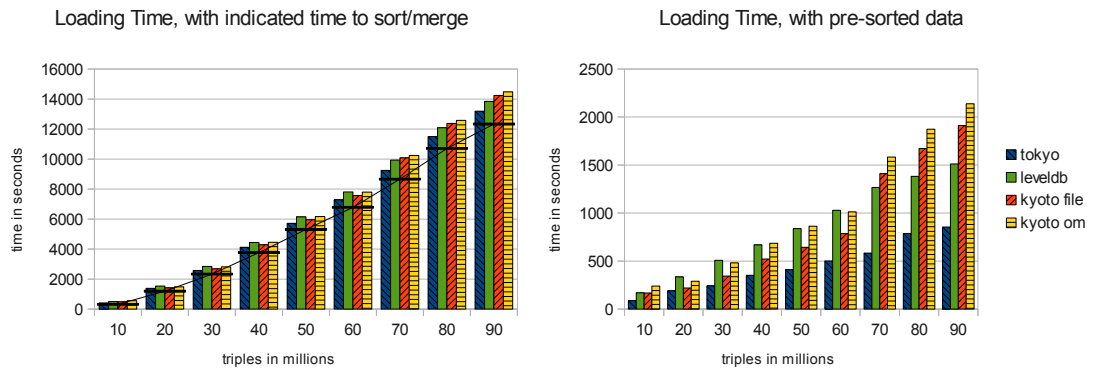


Figure 5.3: On the left: The loading time for the GeoNames data set with an indicated time for the sort-merge work-flow before the data is ready to be inserted into the indexing backends. On the right: The loading time without this process for pre-sorted data.

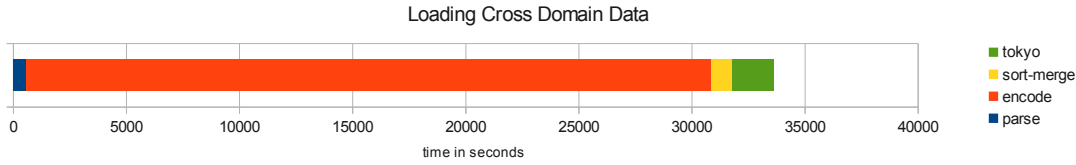


Figure 5.4: Loading time for cross-domain data set, split in individual steps. The encoding step takes 90% of the overall time.

in figure 5.2. Tokyo Cabinet is fastest for smaller chunks and is only second-fastest for 40 million and 50 million triples. At 90 million triples, Tokyo Cabinet is fastest again.

The iteratively loaded LevelDB backend is second-fastest for bigger data sets and even fastest for 80 million triples.

In figure 5.3, the loading time is compared to the time the sorting and merging takes. On the left the black lines show the time to sort and merge the triple batches. On the right, the net loading times with pre-sorted data are plotted. This chart shows that LevelDB performs bad for small inputs but is second-best with 90 million triples.

Bottlenecks In figure 5.3, it is clearly visible that most of the time is spent in parsing, encoding, sorting and merging, and not much time loading the triples into the index backends.

In figure 5.4, the different steps are split up. The parsing and sorting/merging steps do not take much time. The actual dictionary encoding is the major part in the whole process. When comparing the overall time including loading the data into the Tokyo Cabinet backend, 90% of the time is spent dictionary-encoding the triples. Therefore, dictionary encoding is a clear bottleneck in the loading process. This affects the iterative loading in LevelDB as well.

Index Sizes In table 5.2, the index sizes for all three data sets are listed. Figure 5.5 and 5.6 show complementary charts.

A general observation is that LevelDB produces the smallest index sizes. This is due to the Snappy compression that is enabled by default. For the cross-domain data set, LevelDB indexes are 42% the size of the Kyoto Cabinet on-memory index sizes. The Kyoto Cabinet on-memory indexes are in most scenarios the second-smallest indexes, followed by the Tokyo Cabinet indexes. The biggest indexes are produced by the Kyoto Cabinet file-based backend.

The pie chart on the right in figure 5.6 shows the index sizes for all indexes combined. This shows again that LevelDB indexes are smaller in size compared to the other backends.

As explained in Section 4.3, the Kyoto Cabinet on-memory backend stores its data as snapshots to a file. Loading these snapshots takes a significant amount of time: for the cross-domain data set, loading all indexes takes 1152 seconds (≈ 19 min). This is the time to open all index levels serially. Opening the indexes could be parallelized, but



Figure 5.5: The index sizes for each individual index, plotted for all backends for the life-science (left) and sp2bench (right) data sets.

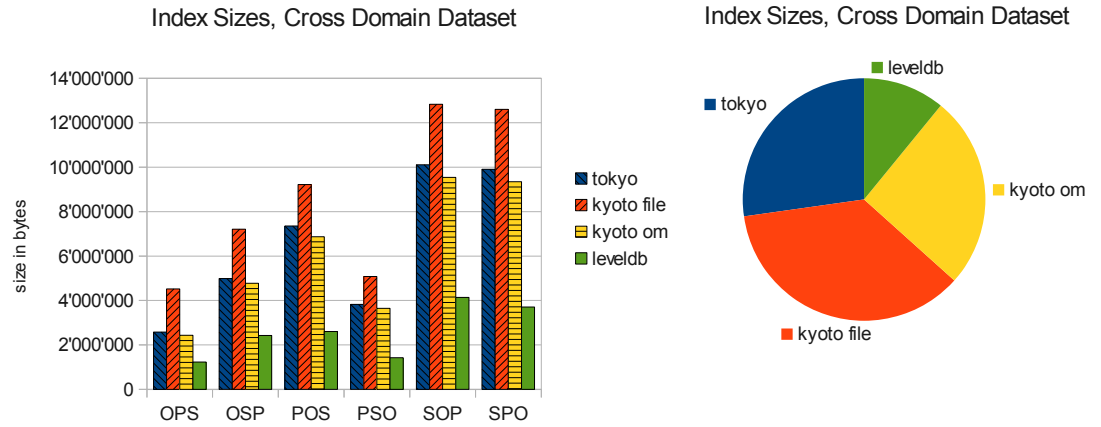


Figure 5.6: On the left: the index sizes for each individual index, plotted for all backends for the cross-domain data set. On the right: the overall index size for the cross-domain data set for all backends relative to each other.

dataset	backend	OPS	OSP	POS	PSO	SOP	SPO	total
CD	tokyo	2,576.0	4,990.0	7,352.2	3,828.4	10,110.8	9,906.6	38,764.0
	k-file	4,522.5	7,205.6	9,218.3	5,083.7	12,829.8	12,602.2	51,462.1
	k-om	2,434.6	4,771.1	6,872.7	3,643.9	9,543.8	9,346.1	36,612.2
	leveldb	1,231.4	2,430.7	2,605.7	1,427.0	4,142.3	3,704.7	15,541.9
LS	tokyo	821.8	1,602.3	2,318.4	819.6	3,282.3	2,793.0	11,637.4
	k-file	1,431.7	2,217.9	2,861.9	1,110.4	4,152.5	3,653.8	15,428.2
	k-om	776.5	1,531.9	1,422.3	773.5	3,099.8	2,626.2	10,230.1
	leveldb	383.3	784.4	823.5	298.9	1,402.0	1,106.3	4,798.5
SP2B	tokyo	253.2	377.6	516.0	246.7	673.4	644.7	2,711.6
	k-file	355.8	479.0	611.4	298.0	796.4	768.7	3,309.4
	k-om	241.6	362.0	484.1	234.9	636.5	608.8	2,567.9
	leveldb	113.4	183.6	183.3	86.7	263.6	231.4	1,062.1

Table 5.2: The index sizes for the three data sets, for each index individually and in total. All sizes are in megabytes.

would most probably still take a few minutes. This is a big disadvantage, as all other backends take less than a second to open all indexes for the cross-domain data set.

Iterative Loading Rdfbox allows for customized identifier sizes. By default, 8-bytes identifiers are used. When choosing between 1 and 8 bytes for the identifiers, a dictionary encoding is applied; i.e. parsed RDF elements are searched in a dictionary that maps RDF elements to their identifier. If the RDF element is found, it was inserted into the indexes before and its identifier is returned. If the element is not found, it is “new” and a new identifier is assigned to it and added to the dictionary. In addition to that, the reverse pair is added to a second dictionary. This dictionary returns for a given identifier the original RDF element. It is used when returning results from the index. Both dictionaries are essential. The first dictionary ensures that equal RDF elements receive equal identifiers, and the second dictionary ensures that results returned by the indexes can be translated back to the URIs or literals.

The first dictionary, however, can be replaced with a hash function. A hash function would return for a given RDF element an identifier that can be computed without consulting a dictionary, which is potentially faster for large data sets. The disadvantage is that hash functions return bigger identifiers. The SHA-1 hash function returns a 20-bytes identifier, which is more than double the size of the dictionary-encoded 8-bytes identifier.

Because the LevelDB indexes were smaller by a large margin, an additional experiment was started, where the first dictionary was replaced with the SHA-1 function. Because the encoding takes 90% of the overall loading time (as shown in figure 5.4), the hypothesis was that loading time could be reduced by replacing the first part of dictionary encoding with SHA-1 hashing. This resulted in bigger indexes, because identifiers were more than twice the size. Instead of 8, 16 and 24 bytes keys in index levels, keys were 20, 40 and

60 bytes long. This was, however, justified with the Snappy compression that would compress the indexes to a size comparable to the other indexes with 8-bytes identifiers.

The hypothesis, however, turned out to be wrong. Iteratively loading the cross-domain data set into the index took 39387 seconds ($\approx 11.9\text{h}$) and was slower compared to the 8-bytes identifier variant, which took 32904 seconds ($\approx 9.1\text{h}$). The 20-bytes identifier experiment took therefore 19.7% more time. Experiments with the smaller data sets showed even worse numbers. The life-science data set group took 50.2% more time and the sp2bench data set group took 32.6% more time.

The overall index size grew to roughly 41 gigabytes, which is a bit larger than the Tokyo Cabinet 8-bytes identifier index. Therefore Snappy compression was as effective as predicted.

5.3 Queries

The second part of the evaluation compared query execution times for all backends. The execution time for each query is measured with cold cache and warm cache. The queries were repeated 3 times for the warm cache figures.

The actual measurements were taken for non-materialized results, i.e. results were the identifiers return by the indexes themselves. When materializing results, the dictionary encoded identifiers have to be translated to their original RDF elements. This step is identical for all backends and is therefore excluded to avoid this potentially disruptive influence.

Note that all query charts are plotted on a *logarithmic scale*, therefore each grid line marks a ten-fold increase. The full listing of SPARQL queries can be found in the appendix A.1.

In the following, the backends are abbreviated: TC for the Tokyo Cabinet, KCF for the Kyoto Cabinet file-based backend, KCOM for the Kyoto Cabinet on-memory backend and LDB for LevelDB.

Cross Domain Queries The cold-cache cross-domain queries in figure 5.7 show that query execution times vary greatly. TC performs generally bad, except for query C-Q2. KCF is equally bad, except for query C-Q8. KCOM and LDB generally perform best, although they balance each other.

For the warm-cache cross-domain queries in figure 5.8, both Kyoto Cabinet backends perform equally well and are comparable to the performance of TC, except for the queries C-Q8 and C-Q10, where TC and LDB perform significantly worse. In general, LDB performs worse or equally at best compared to the other backends. Performance is significantly worse for the queries C-Q3, C-Q5, C-Q7 and C-Q11.

The query execution times for cold and warm caches are listed in table 5.3.

Life Science Queries For the cold-cache life-science queries, the figures are less clear compared to the cross-domain queries. LDB performs generally worse or equal to the other backends, except for query L-Q2, where it is best. KCOM generally performs best,

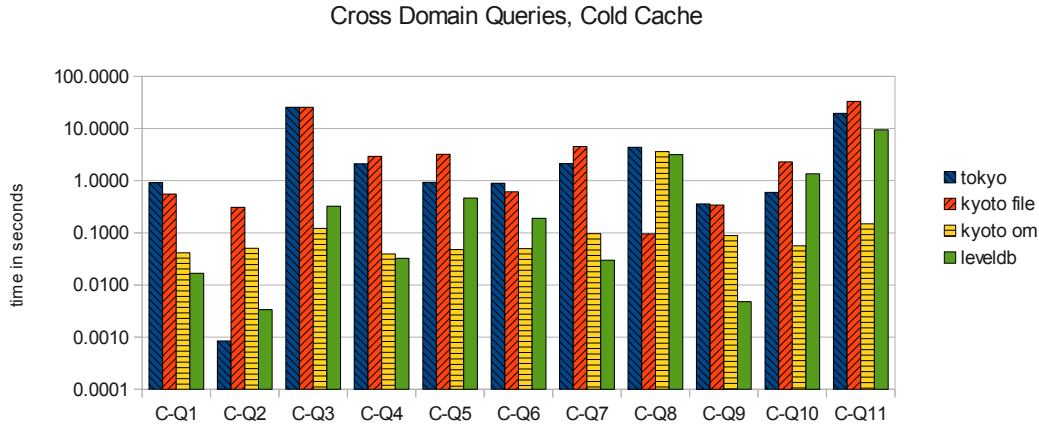


Figure 5.7: The cross-domain queries, measured in seconds with cold caches.

for some queries even orders of magnitude faster, such as query ranges L-Q3 to L-Q8. TC and KCF perform equally well.

For the warm-cache queries, the difference is less distinct. TC and KCF again perform equally well, except for L-Q9, where TC performs best. In general, LDB performs worst; in some queries even orders of magnitude (L-Q1, L-Q4, L-Q6, L-Q7, L-Q9, L-Q10). KCOM is comparable to KCF, except for the queries L-Q7 and L-Q8, where it performs significantly better than all others.

The query execution times for cold and warm caches are listed in table 5.3.

SP²Bench Queries The sp2bench queries are listed in figure 5.11. For the cold-cache queries, all queries perform equally well on all backends, except for two outliers: LDB performs significantly worse in S-Q1 and KCF does so in query S-Q4.

The warm-cache queries perform equally well on all backends, except for LDB which performs worse in S-Q2 and S-Q4.

The query execution times for cold and warm caches are listed in table 5.4.

	cold cache				warm cache			
query	tokyo	k-file	k-om	leveldb	tokyo	k-file	k-om	leveldb
C-Q1	0.9126	0.5560	0.0415	0.0167	0.0017	0.0029	0.0021	0.0028
C-Q2	0.0008	0.3085	0.0505	0.0034	0.0007	0.0007	0.0008	0.0008
C-Q3	25.62	25.58	0.1222	0.3231	0.0175	0.0145	0.0275	0.3018
C-Q4	2.1091	2.9294	0.0393	0.0324	0.0040	0.0045	0.0060	0.0286
C-Q5	0.9264	3.2173	0.0480	0.4653	0.0081	0.0142	0.0255	0.4640
C-Q6	0.8913	0.6119	0.0498	0.1899	0.0346	0.0169	0.0254	0.1875
C-Q7	2.1286	4.5266	0.0963	0.0298	0.0016	0.0020	0.0027	0.0252
C-Q8	4.3695	0.0958	3.6086	3.1467	1.8828	0.0008	0.0012	2.5854
C-Q9	0.3587	0.3399	0.0885	0.0048	0.0007	0.0008	0.0008	0.0008
C-Q10	0.5953	2.3056	0.0566	1.3586	0.0872	0.0018	0.0019	0.0549
C-Q11	19.58	33.17	0.1484	9.4357	0.0425	0.0432	0.0614	0.7379

Table 5.3: A listing with the measured query execution times for the cross-domain queries, cold and warm cache.

	cold cache				warm cache			
query	tokyo	k-file	k-om	leveldb	tokyo	k-file	k-om	leveldb
L-Q1	0.0718	0.3150	0.0511	0.4882	0.0233	0.0276	0.0295	0.4877
L-Q2	0.0114	0.0086	0.0009	0.0006	0.0006	0.0006	0.0006	0.0006
L-Q3	0.3759	0.4643	0.0113	0.3769	0.0083	0.0098	0.0059	0.0061
L-Q4	862.02	162.46	4.5808	5179.9	25.17	1.8559	1.4106	2356.2
L-Q5	0.8189	0.9581	0.0014	0.5990	0.0011	0.0012	0.0009	0.0086
L-Q6	25.04	69.85	0.3392	9.1710	0.2096	0.1728	0.0506	8.7319
L-Q7	1.5483	7.9158	0.0012	1.5888	0.0051	0.0106	0.0007	0.1040
L-Q8	261.83	222.60	0.0012	6250.4	265.46	101.28	0.0011	2305.5
L-Q9	0.1607	12.71	0.7323	12.54	0.0010	0.5489	0.2818	12.01
L-Q10	1.2487	1.1730	1.9244	3.4531	1.2248	0.5806	1.9413	26.52
L-Q11	0.4755	0.4297	0.1994	0.4133	0.1585	0.1705	0.3067	0.8758

Table 5.4: A listing with the measured query execution times for the life-science queries, cold and warm cache.

	cold cache				warm cache			
query	tokyo	k-file	k-om	leveldb	tokyo	k-file	k-om	leveldb
S-Q1	0.0016	0.0018	0.0015	0.0276	0.0007	0.0007	0.0007	0.0008
S-Q2	6.4822	14.93	8.9029	2.1385	0.0956	0.1106	0.1039	0.8001
S-Q3	0.0014	0.0014	0.0013	0.0014	0.0006	0.0008	0.0007	0.0008
S-Q4	0.0043	0.2080	0.0048	0.0063	0.0037	0.0044	0.0044	0.0880

Table 5.5: A listing with the measured query execution times for the sp2bench queries, cold and warm cache.

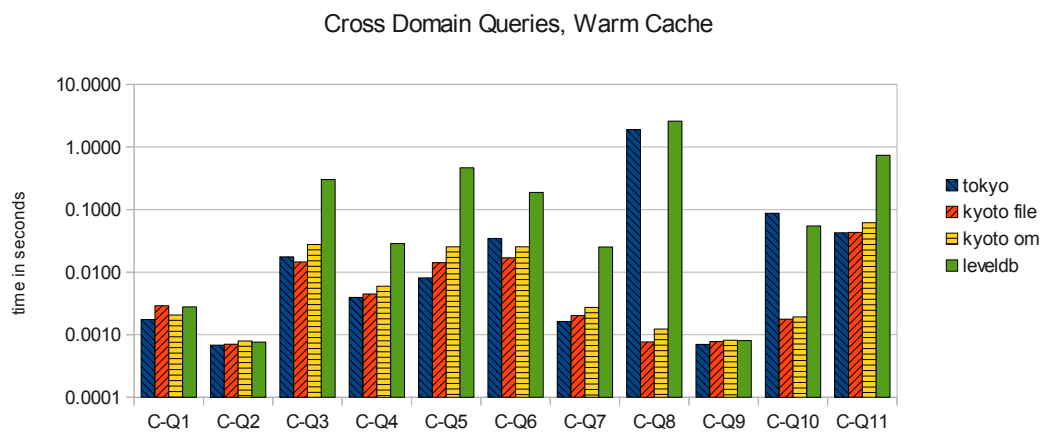


Figure 5.8: The cross-domain queries, measured in seconds with warm caches.

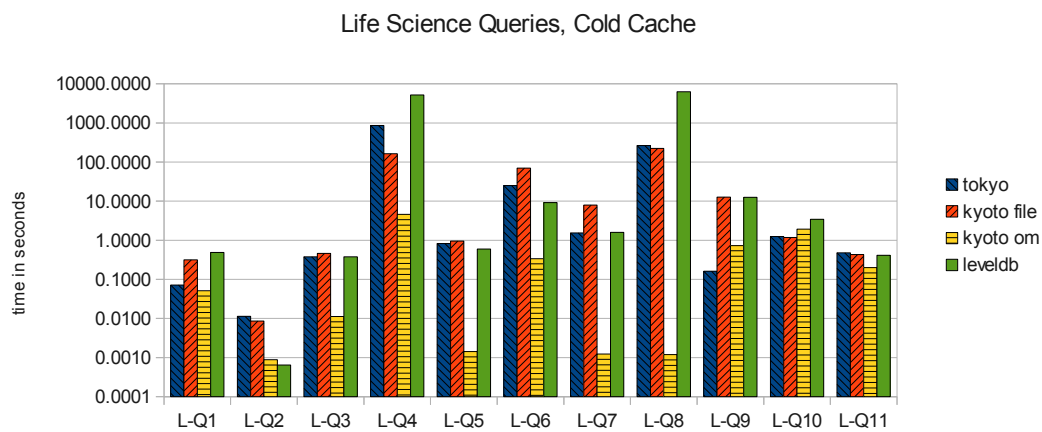


Figure 5.9: The life-science queries, measured in seconds with cold caches.

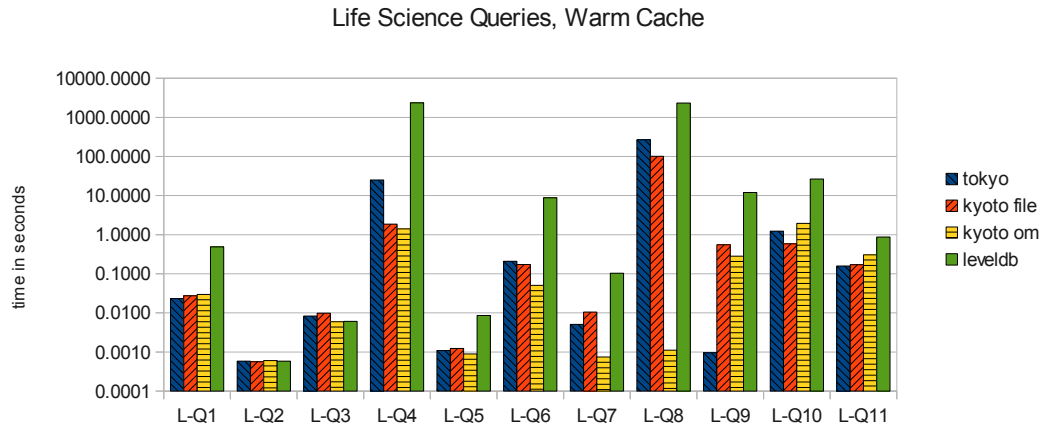


Figure 5.10: The life-science queries, measured in seconds with warm caches.

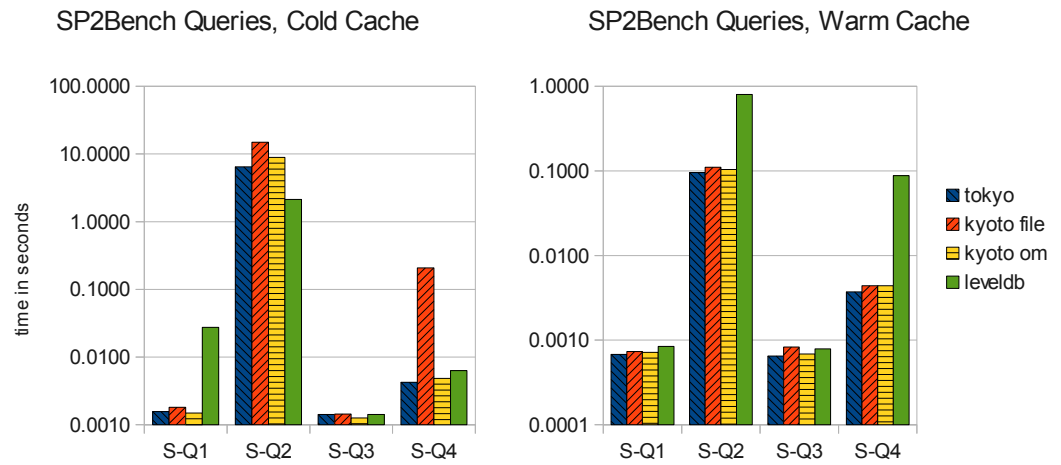


Figure 5.11: The sp2bench queries, measured in seconds with cold and warm caches.

6

Discussion and Conclusion

6.1 Discussion

Evaluation showed that for loading data, the Tokyo Cabinet and LevelDB backends were superior to the other backends, but not by a large margin. Loading the GeoNames data showed that, when gradually increasing the number of triples, LevelDB performs better in relation to the Kyoto Cabinet backends, but is still outperformed by Tokyo Cabinet. For the cross-domain data set, iteratively loading the triples into the LevelDB index was already marginally faster than Tokyo Cabinet. For larger data sets, LevelDB's advantage could be more distinct. Therefore experiments with larger data sets are necessary.

Both Kyoto Cabinet backends performed worst for loading the cross-domain data set. Performance will probably be even worse for larger data sets, as the GeoNames trend suggests. Moreover, Kyoto Cabinet on-memory index loading time (from file snapshots) will aggravate for larger data sets. Its utility will depend on the use case: for long-running setups where the index is loaded once at the beginning, the loading time will amortize. In use cases where the time to load an index matters, other backends are better suited.

An important benefit of LevelDB is its small index size. Indexes are significantly smaller compared to the indexes produced by all other backends due to the Snappy compression used by default. This advantage is of importance for Rdfbox as a whole, since the Hexastore indexing scheme produces larger databases than competing triple stores. LevelDB's compressed indexes help improve this drawback, even more so because performance is still competitive.

Evaluation also showed, however, that the overall loading time is dominated by dictionary encoding. The dictionary encoding module has to maintain two dictionaries. The first translates from RDF element to identifier, the second from identifier to RDF element. When replacing the first dictionary with a hash function, a lookup is not necessary which potentially speeds up the encoding. The disadvantage to hashing is its larger identifiers. However, this can be compensated with compressed LevelDB indexes. An experiment with the cross-domain data set, however, showed that this approach is 20%

slower. This could have several reasons. First, larger identifiers also produce more data that has to be persisted to disk, as index level keys are larger. Second, computing the hash might be slower than looking up the identifier in a dictionary. Experiments with larger data sets could verify these theories.

One issue with the current dictionary encoding module is, however, that both steps — assigning an identifier and adding the identifier with its RDF element mapping to the second dictionary — are coupled. When loading triples, these steps could be decoupled. The hashed identifier can be computed in parallel and the second step (store identifier to RDF element mapping) could be omitted altogether for just building the indexes. Even when omitting the second step, the index can be built, as the second dictionary is needed for materializing results only. Therefore, the second step can be decoupled and be executed independently.

If these steps are independent, all processes in the loading work-flow can compute identifiers independently. The computed identifiers with their RDF elements could be queued and then processed separately. A separate process could add all the mappings to the dictionary. This process could then be improved independently, for example with an efficient mechanism to decide if an identifier was already added to the dictionary to quickly discard previously added RDF elements.

This is, of course, only one proposal to solve the dictionary-encoding bottleneck. Other solutions which preserve the original two-dictionaries strategy could be developed. Unfortunately, time constraints prevented a thorough exploration for improvements and future work needs to address this issue.

When examining the query performance for the most efficient backends in the loading evaluation — Tokyo Cabinet and LevelDB — LevelDB excels for the cold cache scenarios. In the warm cache scenarios, the opposite holds true.

Query performance for the Kyoto Cabinet on-memory backend is, contrary to expectations, not significantly better than other (file-based) backends. Although it performs better than Tokyo Cabinet and Kyoto Cabinet (file-based) for cold-cache queries, this advantage is lost for warm-cache queries. Moreover, LevelDB performs better for half of the cold-cache queries in the cross-domain data set.

Overall, Tokyo Cabinet and LevelDB are the most profitable backends. Both store triples faster than the Kyoto Cabinet backends. LevelDB performs better for cold-cache queries and Tokyo Cabinet is favorable for warm-cache queries. LevelDB's smaller indexes may, however, be a decisive factor, especially for large data sets.

6.2 Conclusion

The goal of this work was to extend Rdfbox — a native triple store that implements the Hexastore indexing scheme — with alternative indexing backends to the already implemented Tokyo Cabinet backend. Tokyo Cabinet is a key-value store, therefore other key-value stores were evaluated and integrated into Rdfbox.

One of them was Kyoto Cabinet, which is a successor to Tokyo Cabinet. Due to their relatedness, the integration was seamless, as the Kyoto Cabinet C API is very similar to

the one of Tokyo Cabinet. Contrary to Tokyo Cabinet, the Kyoto Cabinet API provides a polymorphic key-value data structure that offers a uniform API to all data structures with varying storage mechanisms provided by Kyoto Cabinet.

This uniform API allowed to provide two Kyoto Cabinet backends, one being file-based and the other being in-memory, with shared code bases for most parts.

The third backend was LevelDB, a key-value store that uses a log-structured merge tree to store records instead of the B+ trees used by Tokyo Cabinet and Kyoto Cabinet.

Redis, another key-value store, proved difficult to integrate. Because iterating over records sorted by key is not supported by Redis, an alternative mapping of Hexastore vectors to key-value records had to be implemented. This turned out to be technically possible, but not feasible due to many round-trips that are necessary for iterating over records. Efficient iterating over records and the possibility to jump to a key prefix are key requirements for Rdfbox backends. The fact that Redis does not support iteration by key and the fact that it is based on a client-server model and not embeddable made it unusable for its purpose.

Efforts to use the Tokyo Cabinet in-memory tree data structure as an index failed because of Tokyo Cabinet design decisions. Iterating over a Tokyo Cabinet in-memory tree *simultaneously* is not possible, because the iterating state is stored in the tree data structure itself. Therefore the Tokyo Cabinet in-memory backend was abandoned.

Evaluation showed that triple loading was fastest with the Tokyo Cabinet and LevelDB backends, although not by a large margin to the other backends. In the triple loading work-flow, dictionary-encoding the triples turned out to be a major bottleneck. Parsing, sorting the triples and storing them in the indexes only have a small proportion of the total loading run-time. Experiments to tackle this bottleneck with hashed identifiers were attempted, but did not improve loading time. A solution to this problem could be part of future work.

In general, the Tokyo Cabinet and LevelDB backends turned out to be most performant. Both are faster when loading the triples into the index than the Kyoto Cabinet backends. For query performance, LevelDB performs better in cold-cache scenarios, and Tokyo Cabinet is faster in warm-cache scenarios. LevelDB's indexes are, however, significantly smaller because of the Snappy compression used by default.

An evaluation with larger data sets to determine which backend performs best might be considered for future work.

References

- [1] Berners-Lee, Tim, James Hendler, and Ora Lassila. *The semantic web*. Scientific american 284.5 (2001): 28-37.
- [2] Heath, Tom, and Christian Bizer. *Linked data: Evolving the web into a global data space*. Synthesis Lectures on the Semantic Web: Theory and Technology 1.1 (2011): 1-136.
- [3] Manola, Frank, Eric Miller, and Brian McBride. *RDF primer*. W3C recommendation 10 (2004): 1-107.
- [4] Shadbolt, Nigel, Wendy Hall, and Tim Berners-Lee. *The semantic web revisited*. Intelligent Systems, IEEE 21.3 (2006): 96-101.
- [5] Weiss, Cathrin, Panagiotis Karras, and Abraham Bernstein. *Hexastore: sextuple indexing for semantic web data management*. Proceedings of the VLDB Endowment 1.1, 1008-1019. 2008.
- [6] Abadi, Daniel J., et al., *Scalable semantic web data management using vertical partitioning*. Proceedings of the 33rd international conference on Very large data bases. VLDB Endowment, 2007.
- [7] Weiss, Cathrin, and Abraham Bernstein. *On-disk storage techniques for Semantic Web data-Are B-Trees always the optimal solution?*. The 5th International Workshop on Scalable Semantic Web Knowledge Base Systems. 2009.
- [8] Brickley, Dan, and Libby Miller. *FOAF vocabulary specification 0.98*. Namespace Document 9 (2010).
- [9] Neumann, Thomas, and Gerhard Weikum. *The RDF-3X engine for scalable management of RDF data*. The VLDB Journal–The International Journal on Very Large Data Bases 19.1 (2010): 91-113.
- [10] Chang, Fay, et al. *Bigtable: A distributed storage system for structured data*. ACM Transactions on Computer Systems (TOCS) 26.2 (2008): 4.
- [11] O’Neil, Patrick, et al. *The log-structured merge-tree (LSM-tree)*. Acta Informatica 33.4 (1996): 351-385.

- [12] Behnel, Stefan, et al. *Cython: The best of both worlds*. Computing in Science & Engineering 13.2 (2011): 31-39.
- [13] Tokyo Cabinet <http://www.fallabs.com/tokyocabinet> Last visited: January 2013.
- [14] Kyoto Cabinet <http://www.fallabs.com/kyotocabinet> Last visited: January 2013.
- [15] LevelDB <http://code.google.com/p/leveldb/> Last visited: January 2013.
- [16] Redis <http://redis.io/> Last visited: January 2013.
- [17] Flajolet, Philippe, et al. *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*. 2007 Conference on Analysis of Algorithms, AofA. Vol. 7. 2007.
- [18] Beazley, David M. *Python essential reference*. Addison-Wesley Professional, 2009, Fourth edition, p. 414
- [19] Schmidt, Michael, et al. *Fedbench: A benchmark suite for federated semantic data query processing*. The Semantic Web–ISWC 2011 (2011): 585-600.
- [20] Schmidt, Michael, et al. *SP²Bench: a SPARQL performance benchmark*. Data Engineering, 2009. ICDE'09. IEEE 25th International Conference on. IEEE, 2009.
- [21] Stocker, Markus, et al. *SPARQL basic graph pattern optimization using selectivity estimation*. Proceedings of the 17th international conference on World Wide Web. ACM, 2008.

A

Appendix

A.1 SPARQL Queries

Listing A.1: Cross Domain SPARQL Queries

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX swc: <http://data.semanticweb.org/ns/swc/ontology#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
PREFIX drugbank: <http://www4.wiwiiss.fu-berlin.de/drugbank/resource/drugbank/>
PREFIX drugbankcat: <http://www4.wiwiiss.fu-berlin.de/drugbank/resource/drugcategory/>

# C-Q1
SELECT ?predicate ?object WHERE {
  :Barack.Obama ?predicate ?object .
}

# C-Q2
SELECT ?party ?page WHERE {
  :Barack.Obama dbo:party ?party .
  ?x <http://data.nytimes.com/elements/topicPage> ?page .
  ?x owl:sameAs :Barack.Obama .
}

# C-Q3
SELECT ?paper ?p ?n WHERE {
  ?paper swc:isPartOf <http://data.semanticweb.org/conference/iswc/2008/poster.demo.proceedings> .
  ?paper swrc:author ?p .
```

```
?p rdfs:label ?n .
}

# C-Q4
SELECT ?proceedings ?paper ?p WHERE {
  ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010> .
  ?paper swc:isPartOf ?proceedings .
  ?paper swrc:author ?p .
}

# C-Q5
SELECT ?paper ?p ?x ?n WHERE {
  ?paper swc:isPartOf <http://data.semanticweb.org/conference/iswc/2008/poster.demo.proceedings> .
  ?paper swrc:author ?p .
  ?p owl:sameAs ?x .
  ?p rdfs:label ?n .
}

# C-Q6
SELECT ?role ?p ?paper ?proceedings WHERE {
  ?role swc:isRoleAt <http://data.semanticweb.org/conference/eswc/2010> .
  ?role swc:heldBy ?p .
  ?paper swrc:author ?p .
  ?paper swc:isPartOf ?proceedings .
  ?proceedings swc:relatedToEvent <http://data.semanticweb.org/conference/eswc/2010> .
}

# C-Q7
SELECT ?a ?n WHERE {
  ?a dbo:artist :Michael.Jackson .
  ?a rdf:type dbo:Album .
  ?a foaf:name ?n .
}

# C-Q8
SELECT ?drug ?id ?s ?o ?sub WHERE {
```

```

?drug drugbank:drugCategory drugbankcat:↵
micronutrient .
?drug drugbank:casRegistryNumber ?id .
?drug owl:sameAs ?s .
?s foaf:name ?o .
?s skos:subject ?sub .
}

# C-Q9

SELECT ?x ?p WHERE {
  ?x skos:subject <http://dbpedia.org/↵
resource/Category:FIFA_World_Cup↵
winning_countries> .
  ?p dbo:managerClub ?x .
  ?p foaf:name "Luiz_Felipe_Scolari" .
}

# C-Q10

```

```

SELECT ?n ?p2 ?u WHERE {
  ?n skos:subject <http://dbpedia.org/↵
resource/Category:↵
Chancellors_of_Germany> .
  ?n owl:sameAs ?p2 .
  ?p2 <http://data.nytimes.com/elements/↵
latest_use> ?u .
}

# C-Q11

SELECT ?x ?y ?d ?p ?l WHERE {
  ?x dbo:team :Eintracht_Frankfurt .
  ?x rdfs:label ?y .
  ?x dbo:birthDate ?d .
  ?x dbo:birthPlace ?p .
  ?p rdfs:label ?l .
}

```

Listing A.2: Life Science SPARQL Queries

```

PREFIX owl: <http://www.w3.org/2002/07/owl↵
#>
PREFIX rdfs: <http://www.w3.org/2000/01/↵
rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-↵
rdf-syntax-ns#>
PREFIX dc: <http://purl.org/dc/elements↵
/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX drugbank: <http://www4.wiwiss.fu↵
berlin.de/drugbank/resource/drugbank/>
PREFIX drugbankcat: <http://www4.wiwiss.fu↵
-berlin.de/drugbank/resource/↵
drugcategory/>
PREFIX kegg: <http://bio2rdf.org/ns/kegg#>
PREFIX disease: <http://www4.wiwiss.fu↵
berlin.de/diseasome/resource/diseases↵
/>
PREFIX bio2rdf: <http://bio2rdf.org/ns/↵
bio2rdf#>

# L-Q1

SELECT ?drug ?melt WHERE {
  ?drug drugbank:meltingPoint ?melt .
}

# L-Q2

SELECT ?drug ?melt WHERE {
  ?drug <http://dbpedia.org/ontology/drug/↵
meltingPoint> ?melt .
}

# L-Q3

SELECT ?predicate ?object WHERE {
  <http://www4.wiwiss.fu-berlin.de/↵
drugbank/resource/drugs/DB00201> ?↵
predicate ?object .
}

# L-Q4

SELECT ?Drug ?IntDrug ?IntEffect WHERE {
  ?Drug rdf:type dbo:Drug .
  ?y owl:sameAs ?Drug .
  ?Int drugbank:interactionDrug1 ?y .
  ?Int drugbank:interactionDrug2 ?IntDrug ↵
.
  ?Int drugbank:text ?IntEffect .
}

# L-Q5

SELECT ?drugDesc ?cpd ?equation WHERE {
  ?drug drugbank:drugCategory drugbankcat:↵
cathartics .
  ?drug drugbank:keggCompoundId ?cpd .
}

```

```

?drug drugbank:description ?drugDesc .
?enzyme kegg:xSubstrate ?cpd .
?enzyme rdfs:type kegg:Enzyme .
?reaction kegg:xEnzyme ?enzyme .
?reaction kegg:equation ?equation .
}

# L-Q6

SELECT ?drug ?keggUrl ?chebiImage WHERE {
  ?drug rdf:type drugbank:drugs .
  ?drug drugbank:keggCompoundId ?keggDrug ↵
.
  ?keggDrug bio2rdf:url ?keggUrl .
  ?drug drugbank:genericName ?drugBankName↵
.
  ?chebiDrug dc:title ?drugBankName .
  ?chebiDrug bio2rdf:image ?chebiImage .
}

# L-Q7

SELECT ?drug ?title WHERE {
  ?drug drugbank:drugCategory drugbankcat:↵
micronutrient .
  ?drug drugbank:casRegistryNumber ?id .
  ?keggDrug rdf:type kegg:Drug .
  ?keggDrug bio2rdf:xRef ?id .
  ?keggDrug dc:title ?title .
}

# L-Q8

SELECT ?drug ?enzyme ?reaction WHERE {
  ?drug1 drugbank:drugCategory drugbankcat:↵
:antibiotics .
  ?drug2 drugbank:drugCategory drugbankcat:↵
:antiviralAgents .
  ?drug3 drugbank:drugCategory drugbankcat:↵
: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 dbo:Drug .
  ?drug drugbank:keggCompoundId ?cpd .
  ?enzyme kegg:xSubstrate ?cpd .
  ?enzyme rdfs:type kegg:Enzyme .
  ?reaction kegg:xEnzyme ?enzyme .
  ?reaction kegg:equation ?equation .
}

# L-Q9

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 .
}
# L-Q10

SELECT ?d ?drug5 ?cpd ?enzyme ?equation ←
WHERE {
?drug1 drugbank:possibleDiseaseTarget ←
disease:261 .
?I1 drugbank:interactionDrug2 ?drug1 .
?I1 drugbank:interactionDrug1 ?drug .
?drug drugbank:possibleDiseaseTarget ?d←
.
?drug owl:sameAs ?drug5 .
?drug5 rdf:type dbo:Drug .
?drug drugbank:keggCompoundId ?cpd .
?enzyme kegg:xSubstrate ?cpd .
?enzyme rdf:type kegg:Enzyme .
?reaction kegg:xEnzyme ?enzyme .
?reaction kegg:equation ?equation .
}
# L-Q11

SELECT ?drug5 ?drug6 WHERE {
?drug1 drugbank:possibleDiseaseTarget ←
disease:319 .
?drug1 drugbank:possibleDiseaseTarget ←
disease:270 .
?I1 drugbank:interactionDrug1 ?drug1 .
?I1 drugbank:interactionDrug2 ?drug .
?drug1 owl:sameAs ?drug5 .
?drug owl:sameAs ?drug6 .
}

```

Listing A.3: SP2Bench SPARQL Queries

```

PREFIX owl: <http://www.w3.org/2002/07/owl←
#>
PREFIX xsd: <http://www.w3.org/2001/←
XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/←
rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-←
rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements←
/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms←
/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/←
property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/←
skos/core#>
PREFIX swrc: <http://swrc.ontoware.org/←
ontology#>
PREFIX sp2b: <http://localhost/vocabulary/←
bench/>

# S-Q1

SELECT ?yr WHERE {
?journal rdf:type sp2b:Journal .
?journal dc:title "Journal_1_(1940)" .
?journal dcterms:issued ?yr
}

# S-Q2

SELECT ?inproc ?author ?booktitle ?title ?←
proc ?ee ?page ?url ?yr WHERE {
?inproc rdf:type sp2b:Inproceedings .
?inproc dc:creator ?author .
?inproc sp2b: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
}

# S-Q3

SELECT DISTINCT ?person ?name WHERE {
?article rdf:type sp2b:Article .
?article dc:creator ?person .
?inproc rdf:type sp2b:Inproceedings .
?inproc dc:creator ?person .
?person foaf:name ?name
}

# S-Q4

SELECT ?subject ?predicate WHERE {
?subject ?predicate <http://localhost/←
persons/Paul.Erdoes>
}

```

List of Figures

2.1	A directed graph representation of the triple collection. Resources (blue ellipses) and literals (green boxes) are nodes, predicates are edges (arrows).	4
2.2	Linking Open Data cloud diagram, by Richard Cyganiak and Anja Jentzsch. http://lod-cloud.net/	6
2.3	A graphical representation of the Hexastore index. The OSP and SOP indexes are missing. Each index has only one heading RDF element to avoid an overcrowded figure.	12
3.1	An SPO index, graphically represented as a Hexastore index (top) and mapped to a key-value store in Rdfbox (bottom).	20
3.2	A stripped-down UML diagram of the Rdfbox indexing module with its backends.	24
4.1	An overview to the sorted loading work-flow. It starts on the top left.	37
4.2	The sorted loading work-flow, executed in parallel by a process pool. The process is the same as in figure 4.1, but executed by many processes in chunks.	37
4.3	The second part of the sorted-loading process: merging all sorted, temporary files to 6 resulting files for each index.	38
5.1	The loading time in seconds for the three data sets, plotted for each backend. LevelDB is loaded twice, with sorted input and iteratively.	43
5.2	The loading time for the GeoNames data set. The data is split in 10 million triple chunks to analyze the backend behavior with an increasing number of triples.	44
5.3	On the left: The loading time for the GeoNames data set with an indicated time for the sort-merge work-flow before the data is ready to be inserted into the indexing backends. On the right: The loading time without this process for pre-sorted data.	44
5.4	Loading time for cross-domain data set, split in individual steps. The encoding step takes 90% of the overall time.	45

5.5	The index sizes for each individual index, plotted for all backends for the life-science (left) and sp2bench (right) data sets.	46
5.6	On the left: the index sizes for each individual index, plotted for all backends for the cross-domain data set. On the right: the overall index size for the cross-domain data set for all backends relative to each other. .	46
5.7	The cross-domain queries, measured in seconds with cold caches.	49
5.8	The cross-domain queries, measured in seconds with warm caches.	51
5.9	The life-science queries, measured in seconds with cold caches.	51
5.10	The life-science queries, measured in seconds with warm caches.	52
5.11	The sp2bench queries, measured in seconds with cold and warm caches. .	52

List of Tables

2.1	A collection of ten example triples. Elements are simplified to a name instead of a full URI.	11
2.2	The query patterns and the equivalent index operations.	13
3.1	The used key-value stores and their properties.	18
3.2	The original ten triples, with made-up three-digit identifiers.	19
4.1	A Redis SPO index.	32
4.2	Iterating over the SPO level 1 index in Redis. The tuple denotes the list index for the current key.	32
5.1	The FedBench data sets. Each Data set group consists of one or more individual data sets. The groups are named after their data domain. The smallest group consists of 10 million triples, the largest of almost 160 million triples.	42
5.2	The index sizes for the three data sets, for each index individually and in total. All sizes are in megabytes.	47
5.3	A listing with the measured query execution times for the cross-domain queries, cold and warm cache.	50
5.4	A listing with the measured query execution times for the life-science queries, cold and warm cache.	50
5.5	A listing with the measured query execution times for the sp2bench queries, cold and warm cache.	50