CumulusRDF: Linked Data Management on Nested Key-Value Stores

Günter Ladwig and Andreas Harth

Institute AIFB, Karlsruhe Institute of Technology
76128 Karlsruhe, Germany
{guenter.ladwig,harth}@kit.edu

Abstract. Publishers of Linked Data require scalable storage and retrieval infrastructure due to the size of datasets and potentially high rate of lookups on popular sites. In this paper we investigate the feasibility of using a distributed nested key-value store as an underlying storage component for a Linked Data server which provides functionality for serving Linked Data via HTTP lookups and in addition offers single triple pattern lookups. We devise two storage schemes for our CumulusRDF system implemented on Apache Cassandra, an open-source nested key-value store. We compare the schemes on a subset of DBpedia and both synthetic workloads and workloads obtained from DBpedia's access logs. Results on a cluster of up to 8 machines indicate that CumulusRDF is competitive to state-of-the-art distributed RDF stores.

1 Introduction

Linked Data refers to graph-structured data encoded in RDF (Resource Description Framework[1]) and accessible via HTTP (Hypertext Transfer Protocol)[3]. Linked Data leverages the general web architecture for data publishing and has become increasingly popular for exposing data on the web. The Linking Open Data (LOD) cloud [2] lists over 200 datasets and comprises billions of RDF triples. A basic variant for publishing Linked Data is making an RDF file accessible via HTTP, e.g., by putting the file on a standard web server.

Many datasets on the Linked Data web cover descriptions of millions of entities. DBpedia [2], for example, describes 3.5m things with 670m RDF triples, and LinkedGeoData [13] describes around 380m locations with over 2bn RDF triples. Standard file systems are ill-equipped for dealing with these large amounts of files (each description of a thing would amount to one file). Thus, data publishers typically use full-fledged RDF triple stores for exposing their datasets online. Although the systems are in principle capable of processing complex (and hence expensive) queries, the offered query processing services are curtailed to guarantee continuous service[3].

[1] http://www.w3.org/RDF/

[2] http://lod-cloud.net/

[3] Restrictions on query expressivity are common in other large-scale data services, for example in the Google Datastore. See http://code.google.com/appengine/docs/python/datastore/overview.html\#Queries_and_Indexes.

At the same time, there is a trend towards specialised data management systems tailored to specific use cases [16]. Distributed key-value stores, such as Google Bigtable [5], Apache Cassandra [10] or Amazon Dynamo [6], sacrifice functionality for simplicity and scalability. These stores operate on a key-value data model and provide basic key lookup functionality only. Some key-value stores (such as Apache Cassandra and Googles Bigtable) also provide additional layers of key-value pairs, i.e. keys can be nested. We call these stores *nested* key-value stores. As we will see, nesting is crucial to efficiently store RDF data. Joins, the operation underlying complex queries, either have to be performed outside the database or are made redundant by de-normalising the storage schema. Given the different scope, key-value stores can be optimised for the requirements involving web scenarios, for example, high availability and scalability in distributed setups, possibly replicated over geographically dispersed data centres. Common to many key-value stores is that they are designed to work in a distributed setup and provide replication for fail-safety. The assumption is that system failures occur often in large data centres and replication ensures that components can fail and be replaced without impacting operation.

The goal of our work is to investigate the applicability of key-value stores for managing large quantities of Linked Data. We review Linked Data in Section 2 before we present our main contributions:

- We devise two RDF storage schemes on Google BigTable-like (nested) key-value stores supporting Linked Data lookups and atomic triple pattern lookups (Section 3).
- We compare the performance of the two storage schemes implemented on Apache Cassandra [10], using a subset of the DBpedia dataset with a synthetic and a real-world workload obtained from DBpedia's access log (Section 4).

We discuss related work in Section 5 and conclude with a summary and an outlook to future work in Section 6.

2 Linked Data

We first review basic Linked Data concepts. RDF is a data format for graph-structured data encoded as ($subject$, $predicate$,$object$) triples. These triples are composed of unique identifiers (URI references), literals (e.g., strings or other data values), and local identifiers called blank nodes as follows:

Definition 1. *(RDF Triple, RDF Term, RDF Graph) Given a set of URI references $\mathcal{U}$, a set of blank nodes $\mathcal{B}$, and a set of literals $\mathcal{L}$, a triple (s, p, o) $\in (\mathcal{U} \cup \mathcal{B}) \times \mathcal{U} \times (\mathcal{U} \cup \mathcal{B} \cup \mathcal{L})$ is called an* RDF triple*. We call elements of $\mathcal{U} \cup \mathcal{B} \cup \mathcal{L}$* RDF terms*. Sets of RDF triples are called* RDF graphs*.*

The notion of graph stems from the fact that RDF triples may be viewed as labelled edges connecting subjects and objects.

There are various serialisation formats for RDF, for example normative RDF/XML and Notation3 (N3)[4]. We use the human-readable N3 in this paper. In N3, namespaces can be introduced to abbreviate full URIs as $namespaceprefix$:$localname$, e.g., a

parser expands `foaf:name` in combination with the syntactic definition of the namespace prefix to `http://xmlns.com/foaf/0.1/name`.

The N-Triples syntax[4] is a subset of N3 (without namespaces declarations) where RDF triples are written as whitespace separated RDF terms with a trailing '.'. Brackets (<>) denote URIs and quotes (`""`) denote literals. Blank node identifiers start with '`_:`'.

Definition 2. *(Triple Pattern) A* triple pattern *is an RDF triple that may contain variables (prefixed with '?') instead of RDF terms in any position.*

Triple patterns can be used to express basic RDF queries. For example, a triple pattern `?s foaf:name ?n .` matches all triples with a `foaf:name` predicate. In total there are eight possible patterns for RDF triples (where s,p,o denote a constant and ? a variable): (spo), $(sp?)$, $(?po)$, $(s?o)$, $(?p?)$, $(s??)$, $(??o)$, $(???)$. Queries in SPARQL[5] can be translated to query plans involving triple pattern lookups. However, we exclude full SPARQL query processing capabilities as our goal is to provide scalable Linked Data access.

Linked Data refers to principles [3] that mandate that things (entities or concepts) are identified via HTTP URIs. When dereferencing the URI t denoting a thing, the web server shall return an RDF graph g describing t; g should contain links to other URIs to enable decentralised discovery of resources.

There is a correspondence (in URI syntax or via HTTP redirects) between t and the information resource s which represents that data source of a graph. For an example of a syntactic correspondence consider the resource URI `http://harth.org/andreas/foaf#ah` which is described at the information resource URI `http://harth.org/andreas/foaf`. The former denotes a person and the latter denotes the physical data source which contains the RDF graph. HTTP redirects are another way for ensuring the correspondence; a user agent performs a lookup on URI t, and the web server answers with a new location for the data source s. User agents specify their preferred content format via HTTP's `Accept` header for content negotiation. For example, a lookup on `http://xmlns.com/foaf/0.1/knows` redirects to a HTML file `http://xmlns.com/foaf/spec/index.html` when dereferenced via a Web browser, or to an RDF/XML file `http://xmlns.com/foaf/spec/index.rdf` when accessed via an RDF-aware client.

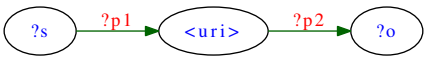

Fig. 1. Illustration of object-subject lookups with `<uri>` as constant.

What should be included in the graph g is only lightly specified[6]. A method which can be regarded as current best practice is employed by DBpedia, which, given a URI,

[4] `http://www.w3.org/TR/rdf-testcases/#ntriples`
[5] `http://www.w3.org/TR/rdf-sparql-query/`
[6] [3] advises to "provide useful information"

returns (i) all triples with the given URI (or its associated information URI) as subject and (ii) some triples with the given URI as object are returned (Figure 1). Such a lookup translates to a union of two triple pattern lookups (or four when including the associated information URI lookups): (i) a $(s??)$ lookup on the SPO index and (ii) a $(??o)$ lookup.

Another method called "Concise Bounded Description" [15] proposes to return all triples which contain the given URI on the subject position with provisions to ensure that connected blank nodes and reified statements are returned. How to include the handling of blank nodes and reified statements in our method is subject to further work.

3 Storage Layouts

In the following we describe indexing schemes for RDF triples on top of nested key-value stores. The goals for the index scheme are:

- To cover all six possible RDF triple pattern with indices to allow for answering single triple patterns directly from the index. In other words, we aim to provide a complete index on RDF triples [8].
- To employ prefix lookups so that one index covers multiple patterns; such a scheme reduces the number of indices and thus index maintenance time and amount of space required.

Ultimately, the index should efficiently support basic triple pattern lookups and (as a union of those) Linked Data lookups which we evaluate in Section 4.

3.1 Nested Key-Value Storage Model

Since Google's paper on Bigtable [5] in 2006, a number of systems have been developed that mimic Bigtable's mode of operation. Bigtable pioneered the use of the key-value model in distributed storage systems and inspired systems such as Amazon's Dynamo [6] and Apache's Cassandra [10]. We illustrate the model in Figure 2.

We use the notation { `key:value` } to denote a key-value pair. We denote concatenated element with, e.g., `sp`, constant entries with `'constant'` and an empty entry with −. The storage model thus looks like the following:

```
{ row key : { column key : value } }
```

Systems may use different strategies to distribute data and organise lookups. Cassandra uses a distributed hashtable structure for network access: storage nodes receive a hash value; row keys are hashed and the row is stored on a node which is closest in numerical space to the row key's hash value. Only hash lookups thus can be performed. Although there is the possibility for configuring an order preserving partitioner for row keys, we dismiss that option, as skewed data distribution easily can lead to hot-spots among the storage nodes.

Another restriction is that entire rows are stored on a single storage node - data with the same row key always ends up on the same machine. Columns, on the other hand, are stored in order of their column keys, which means that the index allows for range scans and therefore prefix lookups.

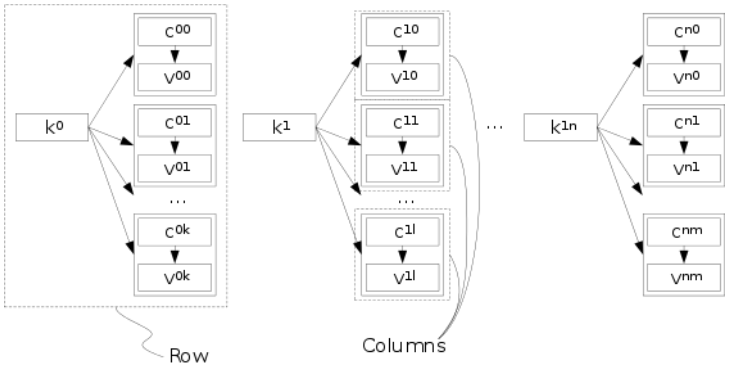

Fig. 2. Illustration of key-value storage model comprising rows and columns. A lookup on the row key (k) returns columns, on which a lookup on column keys (c) returns values (v).

Please note that keys have to be unique; both row keys and column keys can only exist in the index once, which has implications on how we can store RDF triples.

Cassandra has two further features which our index schemes use: supercolumns and secondary indices. Supercolumns add an additional layer of key-value pairs; a storage model with supercolumns looks like the following:

```
{ row key : { supercolumn key : { column key : value }}}
```

Supercolumns can be either stored based on the hash value of the supercolumn key or in sorted order. In addition, supercolumns can be further nested.

Secondary indices, another feature of Cassandra we use for one of the storage layouts, allow to map column values to row keys:

```
{ value : row key }
```

Applications could use the regular key-value layout to also index values to row keys, however, secondary indices are "shortcuts" for such indices. In addition, secondary indices are built in the background without requiring additional maintenance code.

For an introduction to more Cassandra-specific notation we refer the interested reader to the Cassandra web site[7].

We identify two feasible storage layouts: and one based on supercolumns ("Hierarchical Layout"), and one based on a standard key-value storage model ("Flat Layout"), but requiring a secondary index given restrictions in Cassandra.

3.2 Hierarchical Layout

Our first layout scheme builds on supercolumns.

[7] `http://wiki.apache.org/cassandra/DataModel`

The first index is constructed by inserting (s, p, o) triples directly into a supercolumn three-way index, with each RDF term occupying key, supercolumn and column positions respectively, and an empty value. We refer to that index as SPO, which looks like the following:

```
{ s : { p : { o : - } }
```

For each unique s as row key, there are multiple supercolumns, one for each unique p. For each unique p as supercolumn key, there are multiple columns, one for each o as column key. The column value is left empty.

Given that layout, we can perform (hash-base) lookups on s, (sorted) lookups on p and (sorted) lookups on o.

We construct the POS and OSP indices analogously. We use three indices to satisfy all six RDF triple patterns as listed in Table 1. The patterns (spo) and $(???)$ can be satisfied by any of the three indices.

Triple Pattern	Index
(spo)	SPO, POS, OSP
$(sp?)$	SPO
$(?po)$	POS
$(s?o)$	OSP
$(?p?)$	POS
$(s??)$	SPO
$(??o)$	OSP
$(???)$	SPO, POS, OSP

Table 1. Triple patterns and respective usable indices to satisfy triple pattern

3.3 Flat Layout

We base our second storage layout on the standard key-value data model. As columns are stored in a sorted fashion, we can perform range scans and therefore prefix lookups on column keys. We thus store (s, p, o) triples as

```
{ s : { po : - } }
```

where s occupies the row-key position, p the column-key position and o the value position.

We use a concatenated po as column key as column keys have to be unique. Consider using { s : { p : { o } } } as layout, which p as column key and o as value. In RDF, the same predicate can be attached to a subject multiple times, which violates the column key uniqueness requirement. We would like to delegate all low-level index management to Cassandra, hence we do not consider maintaining lists of o's manually.

The SPO index satisfies the triple patterns $(s??)$, $(sp?)$ and (spo) with direct lookups. To perform a $(sp?)$ lookup on the SPO index, we look for the column keys matching p (via prefix lookup on the column key `po`) on the row with row key s.

We also need to cover the other triple patterns. Thus, two more indices are constructed by inserting (p, o, s) and (o, s, p) re-orderings of triples, leading to POS and OSP indices. In other words, to store the SPO, POS and OSP indices in a key-value store, we create a column family for each index and insert each triple three times: once into each column family in different order of the RDF terms.

There is a complication with the POS index though, because RDF data is skewed: many triples may share the same predicate [9]. A typical case are triples with `rdf:type` as predicate, which may represent a significant fraction of the entire dataset. Thus, having P as the row key will result in a very uneven distribution with a few very large rows. As Cassandra is not able to split rows among several nodes, large rows lead to problems (at least a very skewed distribution, out of memory exceptions in the worst case). The row size may exceed node capacity; the uneven distribution makes load balancing problematic. We note that skewed distribution may also occur on other heavily used predicates (and possibly objects, for example in case of heavily used class URIs), depending on the dataset. For an in-depth discussion of uneven distribution in RDF datasets see [7].

To alleviate the issue, we take advantage of Cassandra's secondary indexes.

First, we use PO (and not P) as a row key for the POS index which results in smaller rows and also better distribution, as less triples share predicate and object than just the predicate. The index thus looks like the following:

```
{ po : { s : - } }
```

In each row there are is also a special column 'p' which has P as value:

```
{ po : { 'p' : p } }
```

Second, we use a secondary index which maps column values to row keys, resulting in an index which allows for retrieving all PO row keys for a given P. Thus, to perform a $(?p?)$ lookup, the secondary index is used to retrieve all row keys that contain that property. For $(?po)$ lookups, the corresponding row is simply retrieved.

In effect, we achieve better distribution at the cost of a secondary index and an additional redirection for $(?p?)$ lookups.

4 Evaluation

We now describe the experimental setup and the results of the experiments. We perform two set of experiments to evaluate our approach:

- first, we measure triple pattern lookups to compare the hierarchical (labelled Hier) and the flat storage (labelled Flat) layout; and
- second, we select the best storage layout to examine the influence of output format on overall performance.

4.1 Setting

We conducted all benchmarks on four nodes in a virtualised infrastructure (provided via OpenNebula[8], a cloud infrastructure management system similar to EC2). Each virtualised node has 2 CPUs, 4 GB of main memory and 40 GB of disk space connected to a networked storage server (SAN) via GPFS[9]. The nodes run Ubuntu Linux. We used version 1.6 of Sun's Java Virtual Machine and version 0.8.1 of Cassandra. The Cassandra heap was set to 2GB, leaving the rest for operating system buffers.. We deactivated the Cassandra row cache and set the key cache to 100k. The Tomcat[10] server was run on one of the cluster nodes to implement the HTTP interface.

Apache JMeter[11] was used to simulate multiple concurrent clients.

4.2 Dataset and Queries

For the evaluation we used the DBpedia 3.6 dataset[12] excluding pagelinks; characteristics of that DBpedia subset are listed in Table 2. We obtained DBpedia query logs from 2009-06-30 to 2009-10-25 consisting of 87,203,310 log entries.

Name	Value
distinct triples	120,436,315
distinct subject	18,324,688
distinct predicates	42,004
distinct objects	40,531,020

Table 2. Dataset characteristics

We constructed two query sets:

- for testing single triple pattern lookups, we sampled 1 million S, SP, SPO, SO, O patterns from the dataset;
- for Linked Data lookups we randomly selected 2 million resource lookup log entries from the DBpedia logs (which, due to duplicate lookups included in the logs, amount to 1,241,812 unique lookups).

The triple pattern lookups were executed on CumulusRDF using its HTTP interface. The output of such a lookup is the set of triples matching the pattern, serialised as RDF/XML.

The output of a Linked Data lookup on URI u is the union of two triple pattern lookups: 1) all triples matching pattern $(u??)$ and 2) a maximum of 10k triples matching $(??u)$. The number of results for object patterns is limited in order to deal with the

[8] http://opennebula.org/
[9] http://www-03.ibm.com/systems/software/gpfs/
[10] http://tomcat.apache.org/
[11] http://jakarta.apache.org/jmeter/
[12] http://wiki.dbpedia.org/Downloads36

skewed distribution of objects, which may lead to very large result sets. A similar limitation is used by the official DBpedia server (where a maximum of 2k triples in total are returned).

4.3 Results: Storage Layout

Table 3 shows the size and distribution of the indices on the four machines. The row size of a Cassandra column family is the amount of data stored for a single row-key. For example, for the OSP index, this is the number of triples that share a particular object.

Index	Node 1	Node 2	Node 3	Node 4	Std.Dev.	Max. Row
SPO Hier	4.41	4.40	4.41	4.41	0.01	0.0002
SPO Flat	4.36	4.36	4.36	4.36	0.00	0.0004
OSP Hier	5.86	6.00	5.75	6.96	0.56	1.16
OSP Flat	5.66	5.77	5.54	6.61	0.49	0.96
POS Hier	4.43	3.68	4.69	1.08	1.65	2.40
POS Sec	7.35	7.43	7.38	8.05	0.33	0.56
POS Flat	-	-	-	-	-	-

Table 3. Index size per node in GB. POS Flat is not feasible due to skewed distribution of predicates. The size for POS Sec includes the secondary index.

The load distribution shows several typical characteristics of RDF data and directly relates to the dataset statistics from Table 2. The small maximum row size for SPO shows that there are very few triples that share a subject. The large maximum row size of the OSP index indicates that there are a few objects that appear in a large amount of triples, i.e., the distribution is much more skewed (this is usually due the small number of classes that appear as objects in the many rdf:type triples).

Here, we can also see the difference between the hierarchical and secondary POS indices. The hierarchical POS index only uses the predicate as key, leading to a very skewed distribution (indicated by the maximum row size) and a very uneven load distribution among the cluster nodes (indicated by the high standard deviation). The POS index using a secondary index fares much better as it uses the predicate and object as row key. This validates the choice of using secondary index for POS over the hierarchical layout.

4.4 Results: Queries

The performance evaluation consists of two parts: first, we use triple pattern lookups to compare two CumulusRDF storage layouts and, second, we examine the influence of output formats using Linked Data lookups.

Triple Pattern Lookups Fig. 3 shows the average number of requests/s for the two CumulusRDF storage layouts (flat and hierarchical) for 2, 4, 8, 16 and 32 concurrent

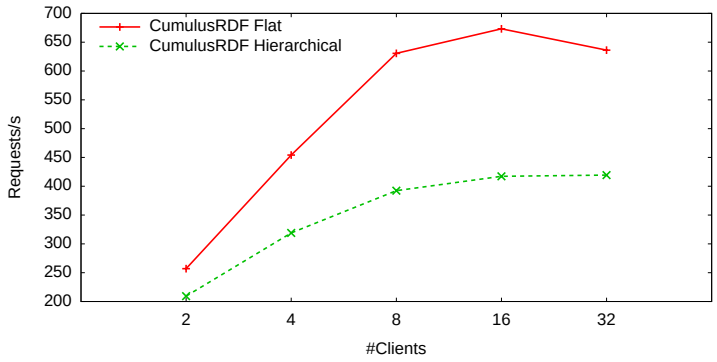

Fig. 3. Requests per second for triple pattern lookups with varying number of clients.

clients. Overall, the flat layout outperforms the hierarchical layout. For 8 concurrent clients, the flat layout delivers 1.6 times as many requests per second as the hierarchical layout. For both layouts the number of requests per second does not increase for more than 8 concurrent clients, indicating a performance limit. This may be due to the bottleneck of using a single HTTP server (as we will see in the next section).

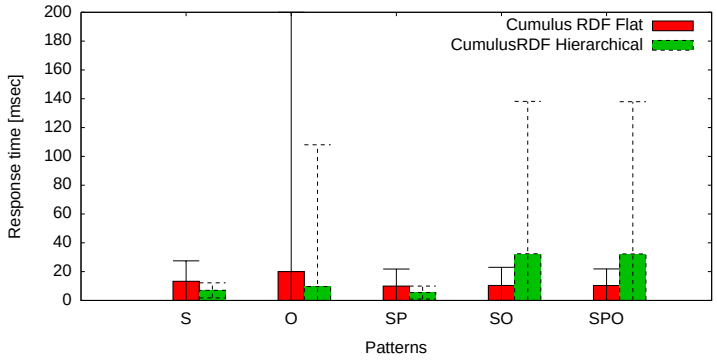

Fig. 4. Average response times per triple pattern (for 8 concurrent clients). Please note we omitted patterns involving the POS index (P and PO). Error bars indicate the standard deviation.

Fig. 4 shows the average response times from the same experiment, broken down by pattern type (S, O, SP, SO and SPO). This shows the differences between the two CumulusRDF storage layouts. While the hierarchical layout performs better for S, O and SP patterns, it performs worse for SO and SPO patterns. The worse performance for SO and SPO is probably due to inefficiencies of Cassandra super columns. For example, Cassandra is only able to de-serialise super columns as a whole, which means

for SPO lookups *all* triples matching a particular subject and predicate are loaded. This is not the case for the flat layout (which does not use super columns): here, Cassandra is able to load only a single column, leading to much better response times.

Linked Data Lookups Fig. 5 shows the average number of requests per second of the flat layout of CumulusRDF with two different output formats: RDF/XML and N-Triples. As CumulusRDF stores RDF data in N3 notation the N-Triples output is cheaper to generate (for example, it does not require the escaping that RDF/XML does). This is reflected in the higher number of lookups per second that were performed using N-Triples output. The difference between the two output format is more pronounced for a higher number of concurrent clients: for 4 clients N-Triples is 12% faster than RDF/XML, whereas for 8 clients the difference is 26%. This indicates that the performance is mainly limited by the Tomcat server, whose webapp performs the output formatting.

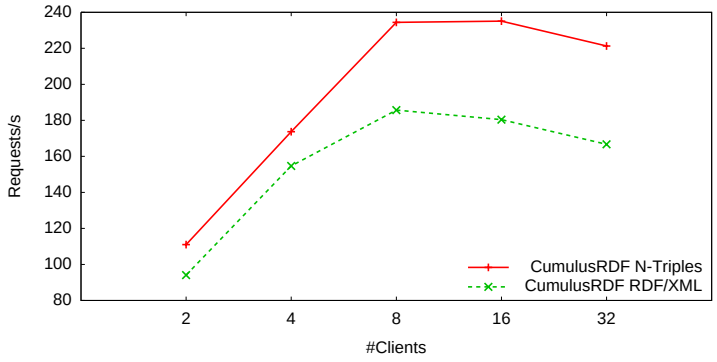

Fig. 5. Requests per second Linked Data lookups (based on flat storage layout) with varying number of clients (including measure of effect of serialisation).

4.5 Conclusion

Overall, the evaluation shows that the flat layout outperforms the hierarchical layout. We also see that Apache Cassandra is a suitable storage solution for RDF, as indicated by the preliminary results. As the different layouts of CumulusRDF perform differently for different types of lookups, the choice of storage layout should be made considering the expected workload. From the experiments we can also see that the output format also has a large impact on performance. It may be beneficial to store the RDF data in an already escaped form if mainly RDF/XML output is desired.

5 Related Work

A number of RDF indexing schemes have been devised. YARS [8] described the idea of "complete" indices on RDF with context (quads)[13], with an index for each possible triple pattern. Similar indices are used by RDF-3X [11] and Hexastore [17]. All of these systems are built to work on single machines. In our work we use complete indices for RDF triples, implemented in three indices over a distributed key-value store.

Abadi et al. introduced an indexing scheme on C-Store for RDF [1] called "vertical partitioning", however, with only an index on the predicate while trying to optimise access on subject or object via sorting of rows. The approach is sub-optimal on datasets with many predicates as subject or object lookups without specified predicate require a lookup on each of the predicate indices. For a follow-up discussion on C-Store we refer the interested reader to [12].

The skewed distribution for predicates on RDF datasets and the associated issues in a distributed storage setting have been noted in [9].

Stratustore is an RDF store implemented over Amazon's SimpleDB [14]. In contrast to Stratusstore, which only makes use of one index on subject (and similarly the RDF adaptor for Ruby[14]), we index all triple patterns. In addition, Cassandra has to be set up on a cluster while SimpleDB is a service offering, accessible via defined interfaces.

6 Conclusion and Future Work

We have presented and evaluated two index regimen for RDF on nested key-value stores to support Linked Data lookups and basic triple pattern lookups. The flat indexing scheme has given best results; in general, our implementation of the flat indexing scheme on Apache Cassandra can be seen as a viable alternative to full-fledged RDF stores in scenarios where large amounts of small lookups are required. We are working on packaging the current version of CumulusRDF for publication as open source at `http://code.google.com/p/cumulusrdf/`. We would like to add functionality for automatically generating and maintaining dataset statistics to aid dataset discovery or distributed query processors. Future experiments include measures on workloads involving inserts and updates.

Acknowledgements

We thank Knud Möller and Chris Bizer for providing access to the dbpedia.org logs and Aidan Hogan and Andreas Wagner for comments. We gratefully acknowledge the support of the Open Cirrus team at KIT's Steinbuch Centre for Computing which provided the necessary infrastructure for conducting the experiments. The authors acknowledge the support of the European Commission's Seventh Framework Programme FP7/2007-2013 (PlanetData, Grant 257641).

[13] See also `http://kowari.sourceforge.net/`.
[14] `http://rdf.rubyforge.org/cassandra/`

References

1. D. J. Abadi, A. Marcus, S. R. Madden, and K. Hollenbach. Scalable semantic web data management using vertical partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 411–422. VLDB Endowment, 2007.
2. S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. G. Ives. Dbpedia: A nucleus for a web of open data. In *Proceedings of the 6th International Semantic Web Conference*, ISWC '07, pages 722–735, 2007.
3. T. Berners-Lee. Linked Data - Design Issues, year = 2006. `http://www.w3.org/DesignIssues/LinkedData`.
4. T. Berners-Lee. Notation3 (N3) A readable RDF syntax year = 2000. `http://www.w3.org/DesignIssues/Notation3`.
5. F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, OSDI '06, pages 15–15, 2006.
6. G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, pages 205–220. ACM, 2007.
7. S. Duan, A. Kementsietsidis, K. Srinivas, and O. Udrea. Apples and oranges: a comparison of rdf benchmarks and real rdf datasets. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 145–156. ACM, 2011.
8. A. Harth and S. Decker. Optimized Index Structures for Querying RDF from the Web. In *Proceedings of the 3rd Latin American Web Congress*, pages 71–80. IEEE Computer Society, 2005.
9. A. Harth, J. Umbrich, A. Hogan, and S. Decker. Yars2: A federated repository for querying graph structured data from the web. In *6th International Semantic Web Conference*, ISWC '07, pages 211–224, 2007.
10. A. Lakshman and P. Malik. Cassandra: a structured storage system on a p2p network. In *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures*, SPAA '09, pages 47–47. ACM, 2009.
11. T. Neumann and G. Weikum. RDF-3X: a RISC-style Engine for RDF. *Proceedings of the VLDB Endowment*, 1(1):647–659, 2008.
12. L. Sidirourgos, R. Goncalves, M. Kersten, N. Nes, and S. Manegold. Column-store support for rdf data management: not all swans are white. *Proceedings of the VLDB Endowment*, 1(2):1553–1563, 2008.
13. C. Stadler, J. Lehmann, K. Höffner, and S. Auer. Linkedgeodata: A core for a web of spatial open data, 2011. Under review, preliminary version at `http://www.semantic-web-journal.net/sites/default/files/swj173_1.pdf`.
14. R. Stein and V. Zacharias. Rdf on cloud number nine. In *Workshop on NeFoRS: New Forms of Reasoning for the Semantic Web: Scalable & Dynamic*, 2010.
15. P. Stickler. CBD - concise bounded description, June 2005. W3C Member Submission, `http://www.w3.org/Submission/CBD/`.
16. M. Stonebraker and U. Cetintemel. "one size fits all": An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11. IEEE Computer Society, 2005.
17. C. Weiss, P. Karras, and A. Bernstein. Hexastore: sextuple indexing for semantic web data management. *Proceedings of the VLDB Endowment*, 1(1):1008–1019, 2008.