NoSQL Databases for RDF: An Empirical Evaluation

Philippe Cudré-Mauroux[1], Iliya Enchev[1], Sever Fundatureanu[2], Paul Groth[2], Albert Haque[3], Andreas Harth[4], Felix Leif Keppmann[4], Daniel P. Miranker[3], Juan F. Sequeda[3], and Marcin Wylot[1,⋆]

[1] University of Fribourg
{phil,iliya.enchev,marcin}@exascale.info
[2] VU University Amsterdam
{s.fundatureanu,p.t.groth}@vu.nl
[3] University of Texas at Austin
{ahaque,miranker,jsequeda}@cs.utexas.edu
[4] Karlsruhe Institute of Technology
{harth,felix.leif.keppmann}@kit.edu

Abstract. Processing large volumes of RDF data requires sophisticated tools. In recent years, much effort was spent on optimizing native RDF stores and on repurposing relational query engines for large-scale RDF processing. Concurrently, a number of new data management systems—regrouped under the NoSQL (for "not only SQL") umbrella—rapidly rose to prominence and represent today a popular alternative to classical databases. Though NoSQL systems are increasingly used to manage RDF data, it is still difficult to grasp their key advantages and drawbacks in this context. This work is, to the best of our knowledge, the first systematic attempt at characterizing and comparing NoSQL stores for RDF processing. In the following, we describe four different NoSQL stores and compare their key characteristics when running standard RDF benchmarks on a popular cloud infrastructure using both single-machine and distributed deployments.

1 Introduction

A number of RDF data management and data analysis problems merit the use of big data infrastructure. These for example include: large-scale caching of linked open data, entity name servers, and the application of data mining techniques to automatically create linked data mappings.

NoSQL data management systems have emerged as a commonly used infrastructure for handling big data outside the RDF space. We view the NoSQL moniker broadly to refer to non-relational databases that generally sacrifice query complexity and/or ACID properties for performance. Given the success of NoSQL systems [14], a number of authors have developed RDF data management systems based on these technologies (e.g. [5, 13, 18, 21]). However, to date,

⋆ Authors are listed in alphabetical order.

H. Alani et al. (Eds.): ISWC 2013, Part II, LNCS 8219, pp. 310–325, 2013.

there has not been a systematic comparative evaluation of the use of NoSQL systems for RDF.

This work tries to address this gap by measuring the performance of four approaches to adapting NoSQL systems to RDF. This adaptation takes the form of storing RDF and implementing SPARQL over NoSQL databases including HBase[1], Couchbase[2] and Cassandra[3]. To act as a reference point, we also measure the performance of 4store, a native triple store. The goal of this evaluation is not to define which approach is "best", as all the implementations described here are still in their infancy, but instead to understand the current state of these systems. In particular, we are interested in: (i) determining if there are commonalities across the performance profiles of these systems in multiple configurations (data size, cluster size, query characteristics, etc.), (ii) characterizing the difference between NoSQL systems and native triple stores, (iii) providing guidance on where researchers and developers interested in RDF and NoSQL should focus their efforts, and (iv) providing an environment for replicable evaluation.

To ensure that the evaluation is indeed replicable, two openly available benchmarks were used (Berlin SPARQL Benchmark and DBpedia SPARQL Benchmark). All measurements were made on the Amazon EC2 cloud platform. More details of the environment are given in Section 3. We note that the results presented here are the product of a cooperation between four separate research groups spread internationally, thus helping to ensure that the described environment is indeed reusable. In addition, all systems, parameters, and results have been published online.

The remainder of this paper is organized as follows. We begin with a presentation of each of the implemented systems in Section 2. In Section 3, we describe the evaluation environment. We then present the results of the evaluation itself in Section 4, looking at multiple classes of queries, different data sizes, and different cluster configurations. Section 4, then discusses the lessons we learned during this process and identifies commonalities across the systems. Finally, we briefly review related work before concluding in Sections 5 and 6.

2 Systems

We now turn to brief descriptions of the five systems used in our tests, focusing on the modifications and additions needed to support RDF. Our choice of systems was based on two factors: (i) Developing and optimizing a full-fledged RDF data management layer on top of a NoSQL system is a very time-consuming task. For our work, we selected systems that were already in development; and (ii) we chose systems that represent a variety of NoSQL system types: document databases (CouchDB), key-value/column stores (Cassandra, HBase), and query compilation for Hadoop (Hive). In addition, we also provide results for 4store, which is a well-known and native RDF store. We use the notation (*spo*) or SPO

[1] `http://hbase.apache.org/`
[2] `http://www.couchbase.com/`
[3] `http://cassandra.apache.org/`

to refer to the subject, predicate, and object of the RDF data model. Question marks denote variables.

2.1 4store

We use 4store[4] as a baseline, native, and distributed RDF DBMS. 4store stores RDF data as quads of (model, subject, predicate, object), where a model is analogous to a SPARQL graph. URIs, literals, and blank nodes are all encoded using a cryptographic hash function. The system defines two types of computational nodes in a distributed setting: (i) storage nodes, which store the actual data, and (ii) processing nodes, which are responsible for parsing the incoming queries and handling all distributed communications with the storage nodes during query processing. 4store partitions the data into non-overlapping segments and distributes the quads based on a hash-partitioning of their subject.

Schema. Data in 4store is organized as property tables [7]. Two radix-tree indices (called P indices) are created for each predicate: one based on the subject of the quad and one based on the object. These indices can be used to efficiently select all quads having a given predicate and subject/object (they hence can be seen as traditional P:OS and P:SO indices). In case the predicate is unknown, the system defaults to looking up all predicate indices for a given subject/object.

4store considers two auxiliary indices in addition to P indices: the lexical index, called R index, stores for each encoded hash value its corresponding lexical (string) representation, while the M index gives the list of triples corresponding to each model. Further details can be found in [7].

Querying. 4store's query tool (4s-query) was used to run the benchmark queries.

2.2 Jena+HBase

Apache HBase[5] is an open source, horizontally scalable, row consistent, low latency, random access data store inspired by Google's BigTable [3]. It relies on the Hadoop Filesystem (HDFS)[6] as a storage back-end and on Apache Zookeeper[7] to provide support for coordination tasks and fault tolerance. Its data model is a column oriented, sparse, multi-dimensional sorted map. *Columns* are grouped into *column families* and timestamps add an additional dimension to each cell. A key distinction is that *column families* have to be specified at schema design time, while columns can be dynamically added.

There are a number of benefits in using HBase for storing RDF. First, HBase has a proven track-record for scaling out to clusters containing roughly 1000 nodes.[8] Second, it provides considerable flexibility in schema design. Finally,

[4] `http://4store.org/`
[5] `http://hbase.apache.org/`
[6] `http://hadoop.apache.org/hdfs`
[7] `http://zookeeper.apache.org/`
[8] See e.g.,`http://www.youtube.com/watch?v=byXGqhz2N5M`

HBase is well integrated with Hadoop, a large scale MapReduce computational framework. This can be leveraged for efficiently bulk-loading data into the system and for running large-scale inference algorithms [20].

Schema. The HBase schema employed is based on the optimized index structure for quads presented by Harth et al. [8] and is described in detail in [4]. In this evaluation, we use only triples so we build 3 index tables: *SPO*, *POS* and *OSP*. We map RDF URIs and most literals to 8-byte ids and use the same table structure for all indices: the row key is built from the concatenation of the 8-byte ids, while column qualifiers and cell values are left empty. This schema leverages lexicographical sorting of the row keys, covering multiple triple patterns with the same table. For example, the table *SPO* can be used to cover the two triple patterns: subject position bound i.e. *(s ? ?)*, subject and predicate positions bound i.e. *(s p ?)*. Additionally, this compact representation reduces network and disk I/O, so it has the potential for fast joins. As an optimization, we do not map numerical literals, instead we use a number's Java representation directly in the index. This can be leveraged by pushing down SPARQL filters and reading only the targeted information from the index. Two dictionary tables are used to keep the mappings to and from 8-byte ids.

Querying. We use Jena as the SPARQL query engine over HBase. Jena represents a query plan through a tree of iterators. The iterators, corresponding to the tree's leafs, use our HBase data layer for resolving triple patterns e.g. *(s ? ?)*, which make up a Basic Graph Pattern (BGP). For joins, we use the strategy provided by Jena, which is indexed nested loop joins. As optimizations, we pushed down simple numerical SPARQL filters i.e. filters which compare a variable with a number, translating them into HBase prefix filters on the index tables. We used these filters, together with selectivity heuristics [19], to reorder subqueries within a BGP. In addition, we enabled joins based on ids, leaving the materialization of ids after the evaluation of a BGP. Finally, we added a mini LRU cache in Jena's engine, to prevent the problem of redundantly resolving the same triple pattern against HBase. We were careful to disable this mini-cache in benchmarks with fixed queries i.e. DBpedia, so that HBase is accessed even after the warmup runs.

2.3 Hive+HBase

The second HBase implementation uses Apache Hive[9], a SQL-like data warehousing tool that allows for querying using MapReduce.

Schema. A property table is employed as the HBase schema. For each row, the RDF subject is compressed and used as the row key. Each column is a predicate and all columns reside in a single HBase column family. The RDF object value is stored in the matching row and column. Property tables are known to have several issues when storing RDF data [1]. However, these issues do not arise in our HBase implementation. We distinguish multi-valued attributes from one

[9] `http://hive.apache.org/query`

another by their HBase timestamp. These multi-valued attributes are accessed via Hive's array data type.

Querying. At the query layer, we use Jena ARQ to parse and convert a SPARQL query into HiveQL. The process consists of four steps. Firstly, an initial pass of the SPARQL query identifies unique subjects in the query's BGP. Each unique subject is then mapped onto its requested predicates. For each unique subject, a Hive table is temporarily created. It is important to note that an additional Hive table does not duplicate the data on disk. It simply provides a mapping from Hive to HBase columns. Then, the join conditions are identified. A join condition is defined by two triple patterns in the SPARQL `WHERE` clause, $(s_1\ p_1\ s_2)$ and $(s_2\ p_2\ s_3)$, where $s_1 \neq s_2$. This requires two Hive tables to be joined. Finally, the SPARQL query is converted into a Hive query based on the subject-predicate mapping from the first step and executed using MapReduce.

2.4 CumulusRDF: Cassandra+Sesame

CumulusRDF[10] is an RDF store which provides triple pattern lookups, a linked data server and proxy capabilities, bulk loading, and querying via SPARQL. The storage back-end of CumulusRDF is Apache Cassandra, a NoSQL database management system originally developed by Facebook [10]. Cassandra provides decentralized data storage and failure tolerance based on replication and failover.

Schema. Cassandra's data model consists of nestable distributed hash tables. Each hash in the table is the hashed key of a row and every node in a Cassandra cluster is responsible for the storage of rows in a particular range of hash keys. The data model provides two more features used by CumulusRDF: super columns, which act as a layer between row keys and column keys, and secondary indices that provide value-key mappings for columns.

The index schema of CumulusRDF consists of four indices (SPO, PSO, OSP, CSPO) to support a complete index on triples and lookups on named graphs (contexts). Only the three triple indices are used for the benchmarks. The indices provide fast lockup for all variants of RDF triple patterns. The indices are stored in a "flat layout" utilizing the standard key-value model of Cassandra [9]. CumulusRDF does not use dictionaries to map RDF terms but instead stores the original data as column keys and values. Thereby, each index provides a hash based lookup of the row key, a sorted lookup on column keys and values, thus enabling prefix lookups.

Querying. CumulusRDF uses the Sesame query processor[11] to provide SPARQL query functionality. A stock Sesame query processor translates SPARQL queries to index lookups on the distributed Cassandra indices; Sesame processes joins and filter operations on a dedicated query node.

[10] `http://code.google.com/p/cumulusrdf/`
[11] `http://www.openrdf.org/`

2.5 Couchbase

Couchbase is a document-oriented, schema-less distributed NoSQL database system, with native support for JSON documents. Couchbase is intended to run in-memory mostly, and on as many nodes as needed to hold the whole dataset in RAM. It has a built-in object-managed cache to speed-up random reads and writes. Updates to documents are first made in the in-memory cache, and are only later persisted to disk using an eventual consistency paradigm.

Schema. We tried to follow the document-oriented philosophy of Couchbase when implementing our approach. To load RDF data into the system, we map RDF triples onto JSON documents. For the primary copy of the data, we put all triples sharing the same subject in one document (i.e., creating RDF molecules), and use the subject as the key of that document. The document consists of two JSON arrays containing the predicates and objects. To load RDF data, we parse the incoming triples one by one and create new documents or append triples to existing documents based on the triples' subject.

Querying. For distributed querying, Couchbase provides MapReduce views on top of the stored JSON documents. The JavaScript Map function runs for every stored document and produces 0, 1 or more key-value pairs, where the values can be null (if there is no need for further aggregation). The reduce function aggregates the values provided by the Map function to produce results. Our query execution implementation is based on the Jena SPARQL engine to create triple indices similar to the HBase approach described above. We implement Jena's Graph interface to execute queries and hence provide methods to retrieve results based on triple patterns. We cover all triple pattern possibilities with only three Couchbase views, on (?p?) (??o) and (?po). For every pattern that includes the subject, we retrieve the entire JSON document (molecule), parse it, and provide results at the Java layer. For query optimization, similar to the HBase approach above, selectivity heuristics are used.

3 Experimental Setting

We now describe the benchmarks, computational environment, and system setting used in our evaluation.

3.1 Benchmarks

Berlin SPARQL Benchmark (BSBM). The Berlin SPARQL Benchmark [2] is built around an e-commerce use-case in which a set of products is offered by different vendors and consumers are posting reviews about products. The benchmark query mix emulates the search and navigation patterns of a consumer looking for a given product. Three datasets were generated for this benchmark:

- 10 million: 10,225,034 triples (Scale Factor: 28,850)
- 100 million: 100,000,748 triples (Scale Factor: 284,826)
- 1 billion: 1,008,396,956 triples (Scale Factor: 2,878,260)

DBpedia SPARQL Benchmark (DBPSB). The DBpedia SPARQL Benchmark [11] is based on queries that were actually issued by humans and applications against DBpedia. We used an existing dataset provided on the benchmark website.[12] The dataset was generated from the original DBpedia 3.5.1 with a scale factor of 100% and consisted of 153,737,783 triples.

3.2 Computational Environment

All experiments were performed on the Amazon EC2 Elastic Compute Cloud infrastructure[13]. For the instance type, we used m1.large instances with 7.5 GiB of memory, 4 EC2 Compute Units (2 virtual cores with 2 EC2 Compute Units each), 850 GB of local instance storage, and 64-bit platforms.

To aid in reproducibility and comparability, we ran Hadoop's TeraSort [12] on a cluster consisting of 16 m1.large EC2 nodes (17 including the master). Using TeraGen, 1 TB of data was generated in 3,933 seconds (1.09 hours). The data consisted of 10 billion, 100 byte records. The TeraSort benchmark completed in 11,234 seconds (3.12 hours).

Our basic scenario was to test each system against benchmarks on environments composed of 1, 2, 4, 8 and 16 nodes. In addition, one master node was set up as a zookeeper/coordinator to run the benchmark. The loading timeout was set to 24 hours and the individual query execution timeout was set to 1 hour. Systems that were unable to load data within the 24 hour timeout limit were not allowed to run the benchmark on that cluster configuration.

For each test, we performed two warm-up runs and ten workload runs. We considered two key metrics: the arithmetic mean and the geometric mean. The former is sensitive to outliers whereas the effect of outliers is dampened in the latter.

3.3 System Settings

4store. We used 4store revision v1.1.4. To set the number of segments, we followed the rule of thumb proposed by the authors, i.e., power of 2 close to twice as many segments as there are physical CPU cores on the system. This led to four segments per node. To benchmark against BSBM, we used the SPARQL endpoint server provided by 4store, and disabled its soft limit. For the DBpedia benchmark, we used the standard 4store client (4s-query), also with the soft limit disabled. 4store uses an Avahi daemon to discover nodes, which requires network multicasting. As multicasts is not supported in EC2, we built a virtual network between the nodes by running an openvpn infrastructure for node discovery.

HBase. We used Hadoop 1.0.3, HBase 0.92, and Hive 0.8.1. One zookeeper instance was running on the master for all cases. We provided 5GB of RAM for the region servers, while the rest was given to Hadoop. All nodes were located in the North Virginia and North Oregon region. The parameters used for HBase are available on our website which is listed in Section 4.

[12] `http://aksw.org/Projects/DBPSB.html`
[13] `http://aws.amazon.com/`

Jena+HBase. When configuring each HBase table, we took into account the access patterns. As a result, for the two dictionary tables with random reads, we used an 8 KB block size so that lookups are faster. For indices, we use the default 64 KB block size such that range scans are more efficient. We enable block caching for all tables, but we favor caching of the *Id2Value* table by enabling the *in-memory* option. We also enable compression for the *Id2Value* table in order to reduce I/O when transferring the verbose RDF data.

For loading data into this system, we first run two MapReduce jobs which generate the 8-byte ids and convert numerical literals to binary representations. Then, for each table, we run a MapReduce job which sorts the elements by row key and outputs files in the format expected by HBase. Finally, we run the HBase bulk-loader tool which actually adopts the previously generated files into the store.

Hive+HBase. Before creating the HBase table, we identify the split keys such that the dataset is roughly balanced when stored across the cluster. This is done using Hadoop's `InputSampler.RandomSampler`. We use a frequency of 0.1, the number of samples as 1 million, and the maximum sampled splits as 50% the number of original dataset partitions on HDFS. Once the HBase table has been generated, we run a MapReduce job to convert the input file into the HFile format. We likewise run the HBase bulk-loader to load the data in the store. Jena 2.7.4 was used for the query layer.

CumulusRDF (Cassandra+Sesame). For CumulusRDF, we ran Ubuntu 13.04 loaded from Amazon's Official Machine Image. The cluster consisted of one node running Apache Tomcat with CumulusRDF and a set of nodes with Cassandra instances that were configured as one distributed Cassandra cluster. Depending on the particular benchmark settings, the size of the Cassandra cluster varied.

Cassandra nodes were equipped with Apache Cassandra 1.2.4 and a slightly modified configuration: a uniform cluster name and appropriate IP configuration were set per node, the location of directories for data, commit logs, and caches were moved to the local instance storage. All Cassandra instances equally held the maximum of 256 index tokens since all nodes ran on the same hardware configuration. The configuration of CumulusRDF was adjusted to fit the Cassandra cluster and keyspace depending on the particular benchmark settings. CumulusRDF's bulk loader was used to load the benchmark data into the system. A SPARQL endpoint of the local CumulusRDF instances was used to run the benchmark.

Couchbase. Couchbase Enterprise Edition 64 bit 2.0.1 was used with default settings and 6.28 GB allocated per node. The Couchbase java client version was 1.1.0. The NxParser version 1.2.3 was used to parse N-Triples and json-simple 1.1 to parse JSON. The Jena ARQ version was 2.9.4.

4 Performance Evaluation

Figure 1 and 2 show a selected set of evaluation results for the various systems. Query execution times were computed using a geometric average. For a more detailed list of all cluster, dataset, and system configurations, we refer the reader to our website.[14] This website contains all results, as well as our source code, how-to guides, and EC2 images to rerun our experiments. We now discuss the results with respect to each system and then make broader statements about the overall experiment in the conclusion.

Table 1 shows a comparison between the total costs incurred on Amazon for loading and running the benchmark for the BSBM 100 million, 8 nodes configuration. The costs are computed using the formula:

$$(1+8)nodes * \$0.240/hour * (loading\_time + benchmark\_time)$$

where the loading and benchmark time are in hours. All values are in U.S. dollars and prices are listed as of May 2013. Exact costs may vary due to hourly pricing of the EC2 instances.

Table 1. Total Cost – BSBM 100 million on 8 nodes

4store	Jena+HBase	Hive+Hbase	CumulusRDF	Couchbase
$1.16	$35.80	$81.55	$105.15	$86.44

4.1 4store

4store achieved sub-second response times for BSBM queries on 4, 8, and 16 nodes with 10 and 100 million triples. The notable exception is Q5, which touches a lot of data and contains a complex FILTER clause. Results for BSBM 1 billion are close to 1 second, except again for Q5 which takes between 6 seconds (16 nodes) and 53 seconds (4 nodes). Overall, the system scales for BSBM as query response times steadily decrease as the number of machines grow. Loading takes a few minutes, except for the 1 billion dataset which took 5.5 hours on 16 nodes and 14.9 hours on 8 nodes. Note: 4store actually times out when loading 1 billion for 4 nodes but we still include the results to have a coherent baseline.

Results for the DBpedia SPARQL Benchmark are all in the same ballpark, with a median around 11 seconds when running on 4 nodes, 19 seconds when running on 8, and 32 seconds when running on 16 nodes. We observe that the system is *not* scalable in this case, probably due to the high complexity of the dataset and an increase in network delays caused by the excessive fragmentation of DBpedia data stored as property tables on multiple machines.

[14] `http://ribs.csres.utexas.edu/nosqlrdf`

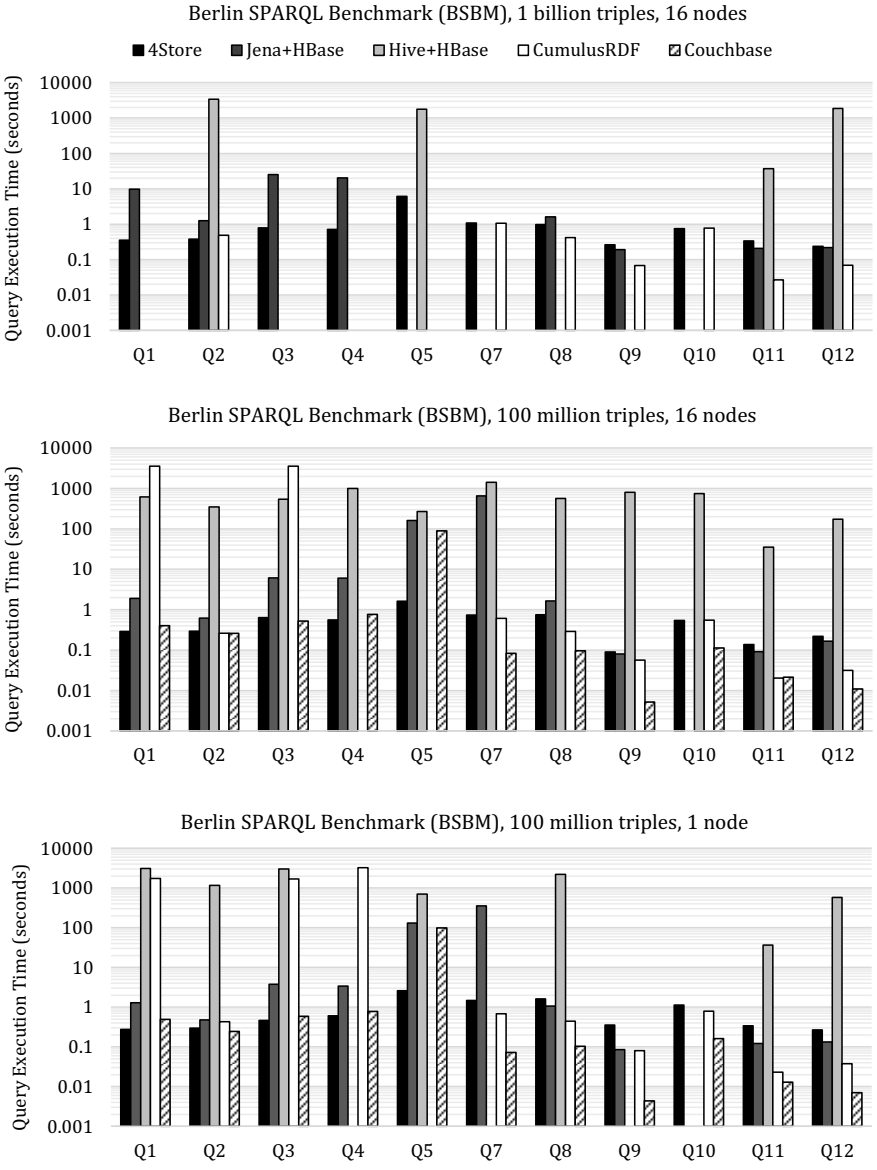

Fig. 1. Results for BSBM showing 100 million and 1 billion triple datasets run on a 16 node cluster. Results for the 100 million dataset on a single node are also shown to illustrate the effect of the cluster size.

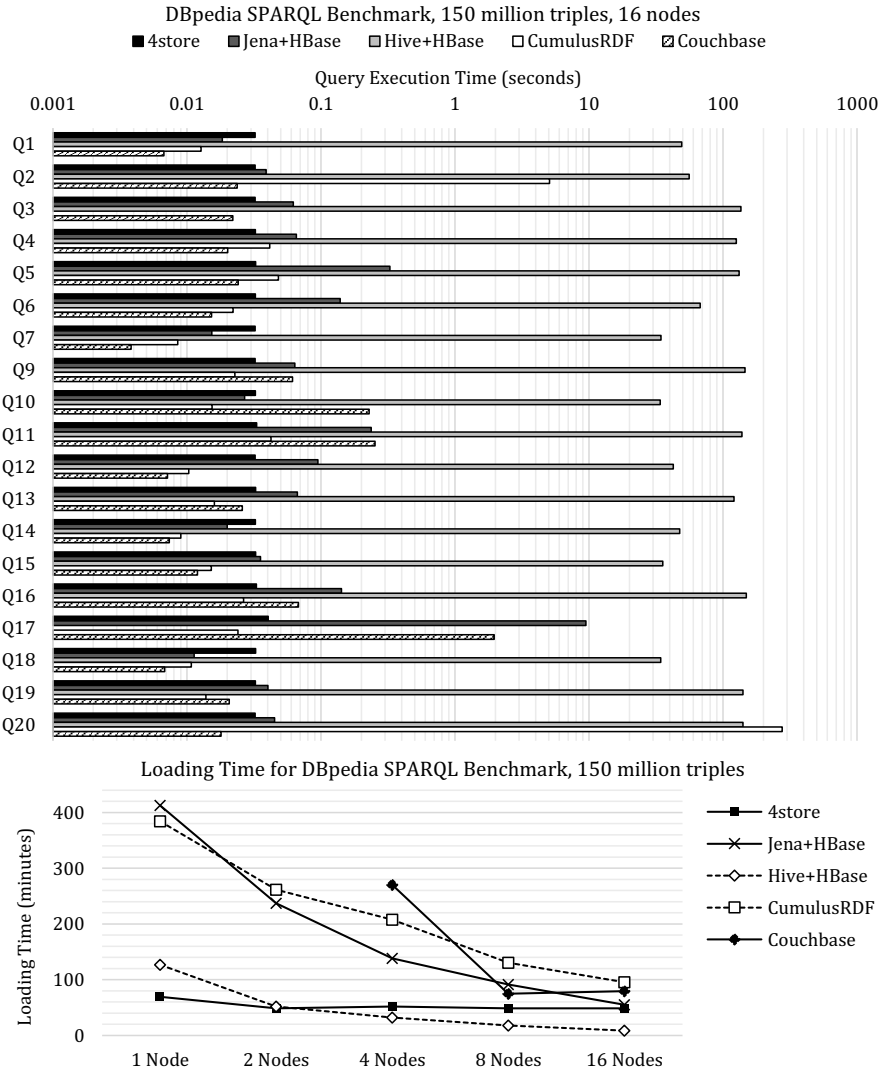

Fig. 2. Results for the DBpedia SPARQL Benchmark and loading times

4.2 Jena+HBase

This implementation achieves sub-second query response times up to a billion triples on a majority of the highly selective query mixes for BSBM (Q2, Q8, Q9, Q11, and Q12). This includes those queries that contain a variable predicate. It is not relevant whether we have an inner or an outer join, instead results are greatly influenced by selectivity. For low selectivity queries (Q1, Q3, Q10), we see that leveraging HBase features is critical to even answer queries. For Q1 and Q3, we provide results for all dataset sizes. These two queries make use of numerical

SPARQL filters which are pushed down as HBase prefix filters, whereas with Q10 we are unable to return results as it contains a date comparison filter which has not been pushed down. The importance of optimizing filters is also shown when a query touches a lot of data such as Q5, Q7 – both of which contain complex or date specific filters.

In terms of the DBpedia SPARQL Benchmark, we see sub-second response times for almost all queries. One reason is that our loading process eliminates duplicates, which resulted in 60 million triples from the initial dataset being stored. In addition, the queries tend to be much simpler than the BSBM queries. The one slower query (Q17) is again due to SPARQL filters on strings that could were not implemented as HBase filters. With filters pushed into HBase, the system approaches the performance of specially designed triple stores on a majority of queries. Still, we believe there is space for improvement in the join strategies as currently the "off-the-shelf" Jena strategy is used.

4.3 Hive+HBase

The implementation of Hive atop HBase introduces various sources of overhead for query execution. As a result, query execution times are in the minute range. However, as more nodes are added to the cluster, query times are reduced. Additionally, initial load times tend to be fast.

For most queries, the MapReduce shuffle stage dominates the running time. Q7 is the slowest because it contains a 5-way join and requires 4 MapReduce passes from Hive. Currently, the system does not implement a query optimizer and it uses the naive Hive join algorithm[15]. For the DBpedia SPARQL Benchmark, we observed faster query execution times than on BSBM. The dataset itself is more sparse than BSBM and the DBpedia queries are simpler; most queries do not involve a join. Due to the sparse nature of the dataset, bloom filters allow us to scan less HDFS blocks. The simplicity of the queries reduce network traffic and also reduce time spent in the shuffle and reduce phases. However, queries with language filters (e.g., Q3, Q9, Q11) perform slower since Hive performs a substring search for the requested SPARQL language identifier).

4.4 CumulusRDF: Cassandra+Sesame

For this implementation, the BSBM 1 billion dataset was only run on a 16 node cluster. The loading time was 22 hours. All other cluster configurations exceeded the loading timeout. For the 100 million dataset, the 2 node cluster began throwing exceptions midway in the benchmark. We observed that the system not only slowed down as more data was added but also as the cluster size increased. This could be attributed to the increased network communication required by Cassandra in larger clusters. With parameter tuning, it may be possible to reduce this. As expected, the loading time decreased as the cluster size increased.

[15] `https://cwiki.apache.org/Hive/languagemanual-joins.html`

BSBM queries 1, 3, 4, and 5 were challenging for the system. Some queries exceeded the one hour time limit. For the 1 billion dataset, these same queries timed out while most other queries executed in the sub-second range.

The DBpedia SPARQL Benchmark revealed three outliers: Q2, Q3, and Q20. Query 3 timed out for all cluster sizes. As the cluster size increased, the execution time of query 2 and 20 increased as well. All other queries executed in the lower millisecond range with minor increases in execution time as the cluster size increased.

One hypothesis for the slow performance of the above queries is that, as opposed to other systems, CumulusRDF does not use a dictionary encoding for RDF constants. Therefore, joins require equality comparisons which are more expensive than numerical identifiers.

4.5 Couchbase

Couchbase encountered problems while loading the largest dataset, BSBM 1 billion, which timed out on all cluster sizes. While the loading time for 8 and 16 nodes is close to 24 hours, index generation in this case is very slow, hampered by frequent node failures. Generating indices during loading was considerably slower with smaller cluster sizes, where only part of the data can be held in main memory (the index generation process took from less than an hour up to more than 10 hours). The other two BSBM data sets, 10 million and 100 million, were loaded on every cluster configuration with times ranging from 12 minutes (BSBM 10 million, 8 nodes) to over 3.5 hours (100 million, 1 node). In the case of DBpedia, there are many duplicate triples spread across the data set, which cause many read operations and thus slower loading times. Also, the uneven size of molecules and triples caused frequent node failures during index creation. For this reason, we only report results with 4 clusters and more, where the loading times range from 74 min for 8 nodes to 4.5 hours for 4 nodes.

Overall, query execution is relatively fast. Queries take between 4 ms (Q9) and 104 seconds (Q5) for BSBM 100 million. As noted above, Q5 touches a lot of data and contains a complex FILTER clause, which leads to a very high number of database accesses in our case, since none of the triple patterns of the query is very restrictive. As the cluster size increases, the query execution times remain relatively constant. Results for DBpedia benchmark exhibit similar trends.

5 Related Work

Benchmarking has been a core topic of RDF data management research. In addition to BSBM and the DBpedia benchmark, SP^{2}Bench [16] and LUBM [6] are widely used benchmarks. Our paper aims to use these benchmarks to investigate NoSQL stores that were not originally intended for RDF. Indexing and storage schemas have a large impact on the performance of a database. Sidirourgos et al. [17] use a single system to evaluate the performance impact of different approaches. Again, our paper looks at multiple, widely used, and currently deployed NoSQL systems.

Of the NoSQL systems available, HBase has been the most widely used. Jianling Sun [18] adopted the Hexastore [23] schema for HBase by storing verbose RDF data. Papailiou et. al. [13] developed *H2RDF*, a distributed triple-store also based on HBase. Khadilkar et al. [22] developed several versions of a triple store that combines the Jena framework with the storage provided by HBase. Przyjaciel-Zablocki et al. [15] created a scalable technique for doing indexed nested loop joins which combines the power of the MapReduce paradigm with the random access pattern provided by HBase.

Another important category for NoSQL RDF storage is the concept of graph databases. Most triple stores (i.e. 4store) can be seen as an implementation of a graph database. Popular products include Virtuoso[16], Neo4j[17], AllegroGraph[18], and Bigdata[19].

An interesting direction forward for NoSQL stores is presented by Tsialiamanis et. al [19]. Using the MonetDB column-store, they show heuristics for efficient SPARQL query planning without using statistics which add overhead in terms of maintainability, storage space, and query execution.

6 Conclusions

This paper represents, to the best of our knowledge, the first systematic attempt at characterizing and comparing NoSQL stores for RDF processing. The systems we have evaluated above all exhibit their own strengths and weaknesses. Overall, we can make a number of key observations:

1. Distributed NoSQL systems can be competitive against distributed and native RDF stores (such as 4store) with respect to query times. Relatively simple SPARQL queries such as distributed lookups, in particular, can be executed very efficiently on such systems, even for larger clusters and datasets. For example, on BSBM 1 billion triples, 16 nodes, Q9, Q11, and Q12 are processed more efficiently on Cassandra and Jena+HBase than on 4store.
2. Loading times for RDF data varies depending on the NoSQL system and indexing approach used. However, we observe that most of the NoSQL systems scale more gracefully than the native RDF store when loading data in parallel.
3. More complex SPARQL queries involving several joins, touching a lot of data, or containing complex filters perform, generally-speaking, poorly on NoSQL systems. Take the following queries for example: BSBM Q3 contains a negation, Q4 contains a union, and Q5 touches a lot of data and has a complex filter. These queries run considerably slower on NoSQL systems than on 4store.

[16] `http://virtuoso.openlinksw.com/`
[17] `http://www.neo4j.org/`
[18] `http://www.franz.com/agraph/allegrograph/`
[19] `http://www.systap.com/bigdata.htm`

4. Following the comment above, we observe that classical query optimization techniques borrowed from relational databases generally work well on NoSQL RDF systems. Jena+HBase, for example, performs better than other systems on many join and filter queries since it reorders the joins, pushes down the selections and filters in its query plans.
5. MapReduce-like operations introduce a higher latency for distributed view maintenance and query execution times. For instance, Hive+HBase and Couchbase (on larger clusters) introduce large amounts of overhead resulting in slower runtimes.

In conclusion, NoSQL systems represent a compelling alternative to distributed and native RDF stores for simple workloads. Considering the encouraging results from this study, the very large user base of NoSQL systems, and the fact that there is still ample room for query optimization techniques, we are confident that NoSQL databases will present an ever growing opportunity to store and manage RDF data in the cloud.

Acknowledgements. The work on Jena+HBase was supported by the Innovative Medicines Initiative Joint Undertaking under grant agreement number 115191. The work on Hive+HBase was supported by the United States National Science Foundation Grant IIS-1018554. The work on CumulusRDF was partially supported by the European Commission's Seventh Framework Programme (PlanetData NoE, grant 257641). The work on Couchbase was supported by the Swiss NSF under grant number PP00P2_128459.

References

1. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB 2007, pp. 411–422 (2007)
2. Bizer, C., Schultz, A.: The berlin sparql benchmark. International Journal on Semantic Web and Information Systems (IJSWIS) 5(2), 1–24 (2009)
3. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Trans. Comput. Syst. 26(2), 4:1–4:26 (2008)
4. Fundatureanu, S.: A Scalable RDF Store Based on HBase. Master's thesis, Vrije University (2012), `http://archive.org/details/ScalableRDFStoreOverHBase`
5. Gueret, C., Kotoulas, S., Groth, P.: Triplecloud: An infrastructure for exploratory querying over web-scale RDF data. WI-IAT (2011)
6. Guo, Y., Pan, Z., Heflin, J.: Lubm: A benchmark for owl knowledge base systems. Web Semantics: Science, Services and Agents on the World Wide Web (2005)
7. Harris, S., Lamb, N., Shadbolt, N.: 4store: The design and implementation of a clustered rdf store. In: 5th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2009), pp. 94–109 (2009)
8. Harth, A., Decker, S.: Optimized Index Structures for Querying RDF from the Web. In: IEEE LA-WEB, pp. 71–80 (2005)

9. Ladwig, G., Harth, A.: CumulusRDF: Linked data management on nested key-value stores. In: The 7th International Workshop on Scalable Semantic Web Knowledge Base Systems (SSWS 2011), p. 30 (2011)
10. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. 44(2), 35–40 (2010)
11. Morsey, M., Lehmann, J., Auer, S., Ngomo, A.-C.N.: DBpedia SPARQL benchmark – performance assessment with real queries on real data. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 454–469. Springer, Heidelberg (2011)
12. O'Malley, O.: Terabyte sort on apache hadoop (2008), `http://sortbenchmark.org/YahooHadoop.pdf`
13. Papailiou, N., Konstantinou, I., Tsoumakos, D., Koziris, N.: H2rdf: adaptive query processing on rdf data in the cloud. In: WWW (Companion Volume)
14. Pokorny, J.: Nosql databases: a step to database scalability in web environment. In: Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services, iiWAS 2011, pp. 278–283. ACM, New York (2011)
15. Przyjaciel-Zablocki, M., Schätzle, A., Hornung, T., Dorner, C., Lausen, G.: Cascading map-side joins over hbase for scalable join processing. CoRR (2012)
16. Schmidt, M., Hornung, T., Küchlin, N., Lausen, G., Pinkel, C.: An experimental comparison of rdf data management approaches in a SPARQL benchmark scenario. In: Sheth, A.P., Staab, S., Dean, M., Paolucci, M., Maynard, D., Finin, T., Thirunarayan, K. (eds.) ISWC 2008. LNCS, vol. 5318, pp. 82–97. Springer, Heidelberg (2008)
17. Sidirourgos, L., Goncalves, R., Kersten, M., Nes, N., Manegold, S.: Column-store support for rdf data management: not all swans are white. Proc. of the VLDB Endow. 1(2), 1553–1563 (2008)
18. Sun, J.: Scalable rdf store based on hbase and mapreduce. In: 2010 3rd International Conference Advanced Computer Theory and Engineering, ICACTE (2010)
19. Tsialiamanis, P., Sidirourgos, L., Fundulaki, I., Christophides, V., Boncz, P.: Heuristics-based query optimisation for SPARQL. In: Proceedings of the 15th International Conference on Extending Database Technology
20. Urbani, J., Kotoulas, S., Maassen, J., Drost, N., Seinstra, F., Harmelen, F.V., Bal, H.: Webpie: A web-scale parallel inference engine. In: Third IEEE International Scalable Computing Challenge (SCALE2010), held in Conjunction with the 10th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGrid (2010)
21. Urbani, J., van Harmelen, F., Schlobach, S., Bal, H.: Querypie: Backward reasoning for owl horst over very large knowledge bases. In: Aroyo, L., Welty, C., Alani, H., Taylor, J., Bernstein, A., Kagal, L., Noy, N., Blomqvist, E. (eds.) ISWC 2011, Part I. LNCS, vol. 7031, pp. 730–745. Springer, Heidelberg (2011)
22. Khadilkar, V., Murat Kantarcioglu, B.T., Castagna, P.: Jena-hbase: A distributed, scalable and efficient rdf triple store. In: Proceedings of the ISWC 2012 Posters & Demonstrations Track (2012)
23. Weiss, C., Karras, P., Bernstein, A.: Hexastore: sextuple indexing for semantic web data management. Proc. of the VLDB Endow. (2008)