RDF on Cloud Number Nine

Raffael Stein[1] and Valentin Zacharias[2]

[1] Karlsruhe Institute of Technology (KIT), Karlsruhe Service Research Institute (KSRI), Englerstr. 11, 76131 Karlsruhe, Germany
`raffael.stein@kit.edu`
[2] Forschungszentrum Informatik (FZI), Haid-und-Neustr. 10-14, 76131, Germany
`zacharias@fzi.de`

Abstract. We examine whether the existing 'Database in the Cloud' service SimpleDB can be used as a back end to quickly and reliably store RDF data for massive parallel access. Towards this end we have implemented 'Stratustore', an RDF store which acts as a back end for the Jena Semantic Web framework and stores its data within the SimpleDB. We used the Berlin SPARQL Benchmark to evaluate our solution and compare it to state of the art triple stores. Our results show that for certain simple queries and many parallel accesses such a solution can have a higher throughput than state of the art triple stores. However, due to the very limited expressiveness of SimpleDB's query language, more complex queries run multiple orders of magnitude slower than the state of the art and would require special indexes. Our results point to the need for more complex database services as well as the need for robust, possible query dependent index techniques for RDF.

1 Introduction

The current trend of Cloud Computing promises to make complex IT solutions available as service to customers in a way that is elastically scalable and priced with a utility pricing model that enables customers to only pay for what they need. For the customer cloud services are expected to bring the following advantages: 1) The customer does not need to worry about the complexities of running the respective IT solution 2) Large cloud computing companies might be able to realize scale effects and hence offer a solution at a price lower than would be possible at a smaller scale 3) The customer does not need to provision infrastructure for an estimated future peak demand because he/she can scale the solution on demand and pay only for what is needed.

Many different kinds of cloud computing services are currently offered, ranging from infrastructure services (that allow to rent virtual computers) to complex software service [1]. Of particular interest for this paper are "database as a service" offerings - and here in particular the probably oldest and best established one: SimpleDB by Amazon [2]. This service offers access to a distributed, scalable and redundant database management system with a utility pricing model. For the user they hold the promise of not having to worry about the problematic issues of a large scale database development; of not having to worry about redundancy, backups, and scaling to large data sets and many concurrent database clients. This paper examines whether these promises can be realized for the storage of RDF; whether the SimpleDB service allows to build a scalable, “worry free” triple store.

We have examined this question by creating Stratustore, a system that extends the Jena Semantic Web Framework to store the data in the SimpleDB. We have evaluated its performance using the Berlin SPARQL Benchmark.

After a short discussion of related work this paper starts with a description of the most important properties of the SimpleDB service. Next, the architecture and design of the Stratustore is described. Then the evaluation setup and results are detailed before a conclusion and discussion.

This paper is a summary presentation of the diploma thesis by Raffael Stein which has been published in German under the title “Entwicklung eines skalierbaren Semantic Web Datanspeichers in der Cloud”.

2 Related Work

To the authors best knowledge no comparable work that uses and evaluates a database cloud service as a back end for a triple store exist. There is, however, considerable work showing great promise for the use of column stores - the database technology also underlying the SimpleDB service - for the storage and querying of RDF data [3–5]. Under the name of “Connected Commons” [6], Talis is offering specifically a triple store as a service. This service is surely an important offer if only RDF data needs to be processed, however, it lacks the maturity and entire ecosystem of different services to be found with Amazon.

Amazon Web Services

Amazon was one of the first large companies to provide cloud computing services and by now offers a large variation of cloud services. For the realization and evaluation of the Stratustore we used the Simple Storage Service (S3), the Elastic Compute Cloud (EC2) and - most importantly - the SimpleDB. Each of these service will be introduced below.

The **Simple Storage Service (S3)** is a distributed key-value store that scales to very large data sets and massive parallel access. The use of the service is charged based on traffic and the amount of data stored. Within the context of this work it was used mostly to store the machine images that were then executed with EC2 (see below). The S3 is also needed for large CLOBS that go beyond the limits of SimpleDB (see below).

The **Elastic Compute Cloud (EC2)** is a service that offers virtual computer instances that can be rented on an hourly basis. These instances are started with an operating system and applications based on an machine image stored in the S3. EC2 is charged based on the properties of the virtual computer (the customer can choose between different numbers of cores, sizes of RAM and hard disks), the time it is used and the data transferred into and out of Amazon's data centers. In the context of this work EC2 was used to run all tests and evaluations - indeed the assumption behind Stratustore is, that it would be used mostly by applications that are already executed within Amazons cloud.

The **SimpleDB** is a distributed database that is accessed through a simple REST or SOAP interface. The use of the SimpleDB is charged based on the amount of data stored (on a much higher rate than for the S3) and based on the computing time needed for query processing.

The SimpleDB service works without a user defined schema and organizes data in the following way (see also figure 1):

– At the highest level data is organized in *domains*, which can be thought of as akin to a collection or a table.
– Within domains data is stored in *items*, which is the basic unit of data for the SimpleDB.
– Each item then has a number of *attributes* each of which can have one or more *values*.

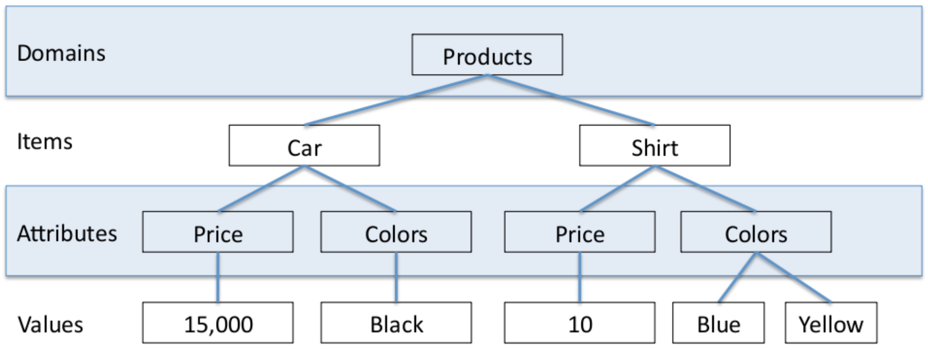

Fig. 1. Data organization inside the SimpleDB

A number of limitations exist that restrict the flexibility of this data organization: A domain must not be larger than 10GB, an item must not have more than 256 attribute-value pairs and an attribute value may not be larger than 1024 bytes. The SimpleDB also uses strings as the only data type - meaning that all comparisons have to be done lexically and that numbers have to be padded with zeros to make this possible.

The SimpleDB keeps multiple copies of a user's data to ensure reliability and security of the data. However, the way this is done means that it supports only "eventual consistency" [7] where the propagation of changes can take a few milliseconds and a user query may hit an out-of-date copy. It is not even guaranteed that a query after a write by the same client will reflect that write operation. SimpleDB also lacks any support for normalized data.

The biggest drawback is the limited expressiveness of SimpleDB's query language. In syntax this language is modeled after SQL select queries, it is, however, much less expressive. In particular it does not allow for comparisons between stored entities; it does not allow for any kind of join operation. That means queries containing dependencies must be split into several parts. Imagine asking the database for all work colleagues of Bob. We do not know the company name, so a first query has to find out the company Bob works at. A second query can then find everyone who works at that company. Without an index for that specific case, it is not possible to answer a query like this in a single run.

3 Stratustore

We have implemented the Stratustore that extends the Jena Semantic Web Framework such that it can use Amazon's SimpleDB as back end (in addition to relational database and main memory back ends already provided). A sketch of its overall architecture is shown in figure 2.

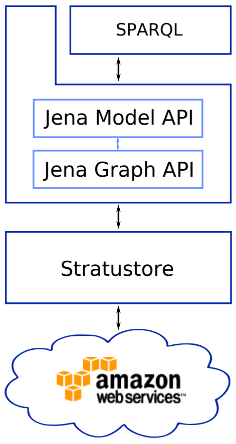

Fig. 2. Architectural overview of the Stratustore

The Stratustore is realized as an implementation of the Jena Graph API which supports a simple interface organized around triples. A detailed description of the layers of Jena can be found in [8]. Adding and removing is done on a per triple basis and queries are realized based on TripleMatch objects; objects that specify the expected values for some or all of the parameters of a triple. The data models of RDF and of the SimpleDB differ considerably and there are many different ways to map the one to the other.

The easiest mapping is a triple oriented mapping - one triple is mapped to one item, with the attributes subject, predicate and object. This simple mapping avoids SimpleDB's restrictions with respect to the maximum attribute/value count. However, this mapping also means that - because of the missing join functionality in SimpleDB's query language - only a very small part of SPARQL query processing can be done by the SimpleDB. Most of the processing for queries needs to be done on the client resulting in a large amount of data that needs to be transferred over the network and very bad performance.

Other mappings define sets of triples that are jointly represented by one item. For example all triples that share the same subject and predicate can be represented in one item. Or all triples that share the same predicate could be mapped to one item. These mappings increase the portion of queries that can be answered directly within SimpleDB, however, at the same time they increase the risk of hitting the limits of the SimpleDB data model.

For the Stratustore we settled for an entity oriented mapping: here one item represents the data known about one subject, i.e. one item for entity "s" contains the data from all triples that contain s as a subject. Each item has an attribute "S" that has the URI of the subject represented by the current item. The other attributes then each represent one predicate defined for this subject and the attribute values represent the objects (recall that multiple values for one attribute are allowed in SimpleDB's data model). Taking figure 1 as an example, that means that a shirt is stored as a item with the attributes "S" containing the subject URI, the attribute "price" containing the price and the attribute "colors" containing the available colors.

The entity oriented mapping allows to push a larger proportion of SPARQL queries into the SimpleDB, for many cases entire SPARQL queries can now be translated into SimpleDB queries. However, it runs afoul of SimpleDB's limitations on the number of attribute-value pairs: Whenever an entity appears as subject in more than 255 triples there will need to be more than one item and queries will have to be split. Simple techniques can restrict the runtime impact occasional split items have; nevertheless it adds considerable additional complexity. Another problem for the entity oriented mapping is the fact that SimpleDB does not allows to query for attribute names, and that hence with the entity oriented mapping variables in predicate position pose a challenge. This second problem can be solved simply by creating a second item for each entity, this time with attribute names representing object and attribute values representing predicates, however, this leads to a doubling of storage requirements and update times. The final problem (affecting all possible mappings) is caused by the size limitation on attribute values which have no correspondence in RDF. This means that long values have to be stored separately in the S3 and that a separate text index is necessary. The code to handle these restrictions of the SimpleDB is not currently implemented in Stratustore and so it too is bound by these restrictions. The Berlin SPARQL Benchmark does not break these restrictions, so this was (almost) no problem for the evaluation.

3.1 Uploading Data

By the time this work was performed (early 2009), SimpleDB did not support any bulk uploading and hence naive uploading (one triple at a time) took a very long time. We used 1) triples collected by subject, 2) sorted inputs and 3) multi-threading to help tackle this problem.

1. Since multiple data changes to one item can be bundled into one request and since one item represents all the triples known about one subject URI, the Stratustore merges consecutive triple update requests with the same URI. This means that it holds back and combines consecutive update requests for triples that have the same subject. The item thus constructed is sent to the SimpleDB only when an update request with a different URI arrives or the the update process is finished.
2. To make best use of this bundling based on consecutive triples we pre-sorted the input triples by subject for larger updates.
3. Finally we used 10 threads running in parallel. More than 10 threads and distributing the uploading over more than one machine brought no additional benefit - 10 threads on one computer were already already able to saturate the write capacity of one SimpleDB domain.

Recently the SimpleDB interface was changed to allow bulk uploads of up to 25 items at one time. With this new functionality we would expect the upload speed to increase by up to one order of magnitude, the structure of our implementation would not need to change fundamentally.

3.2 Querying Data

The Stratustore supports querying through the Jena Graph API and the SPARQL language. SPARQL is the W3C standardized query language for RDF graphs. Its specification can be found at [9]. The triple oriented query operations of the Graph API can be translated directly into SimpleDB SELECT queries. To answer SPARQL queries, a mapping from SPARQL to the much simpler SELECT language is needed. The SPARQL query strings are broken into parts which are then transformed into SELECT queries and evaluated separately by SimpleDB. The results of all single queries are merged again and returned to the user. For this the pipeline shown in figure 3 is used.

The SPARQL query is first received and parsed by the Jena Semantic Web framework. Sets of triple patterns are created from the query and handed (via the Graph API) to the Stratustore. The Stratustore then groups the triple patterns by subject. The grouping is done to take advantage of the entity oriented schema, which is used to store the triples inside SimpleDB. Each pattern group can be combined if the single patterns ask about the same subject. One SELECT query is created for each group of triple patterns and all of these are posed in parallel to the SimpleDB. An example of the conversion of a SPARQL to a SELECT query is given in figures 4 and 5. The patterns which have the same subject form a SELECT query. The results of both SELECT queries have to be matched

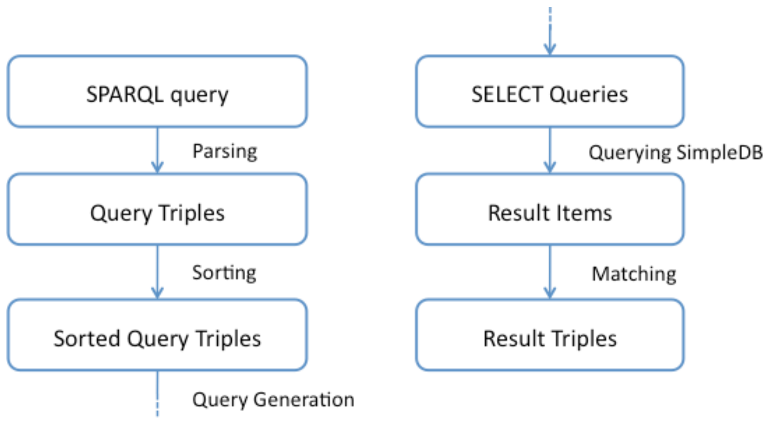

Fig. 3. Processing of a SPARQL query inside the Stratustore

against each other after they have been retrieved. It is not possible to compare two entries inside the SimpleDB. Therefore, two seperate queries are necessary. Query marshaling, cryptographic signing and remote access are handled by the Typica Library [10].

In response to these queries the SimpleDB returns the list of items matching the SELECT queries. In many cases multiple requests must be posed to retrieve all results for a SELECT query, since the size of responses is limited both in the number of items returned as well as the overall size of the message. Stratustore then re-constructs the triples based on the items and performs any necessary joins to find the variable assignments for the triple patterns and returns these to the broader Jena Framework.

Finally the Jena Framework applies any needed additional processing - this includes applying FILTER statements as well as combining graph group patterns, particular to realize the UNION statements. After this processing the result to the SPARQL query is returned to the user.

```
@prefix example: <http://example.org/> .
@prefix country: <http://downlode.org/rdf/iso-3166/countries#>

SELECT ?s WHERE {
  ?s rdf:type example:shirt .
  ?s example:producedBy ?m .
  ?m rdf:type example:Producer .
  ?m example:producesIn country:DE .
}
```

Fig. 4. Pattern Grouping inside the Stratustore

```
SELECT s, example:producedBy FROM products WHERE
  rdf:type = "example:shirt"

SELECT s FROM products WHERE
  rdf:type = "example:Producer"
  INTERSECTION
  example:producesIn = "country:DE"
```

Fig. 5. The aforementioned SPARQL query transformed into SimpleDB SELECT queries

4 Evaluation

We chose to use the Berlin SPARQL Benchmark (BSBM) [11] for evaluation. This gives us the possibility to compare the figures to other Semantic Web stores already tested with BSBM.

The BSBM is a benchmark system which allows the evaluation of semantic data stores that provide a SPARQL endpoint. It is based on an e-commerce use case which is centered around a set of products that are offered by different vendors. For the products, there are reviews written by consumers. To query the generated triples, the BSBM provides a sequence of 12 different SPARQL queries which simulate a user interaction with this e-commerce scenario.

The benchmark consists of a data generator and a query driver. The generator can be used to create sets of connected triples of any size. The query driver constructs queries which fit to the generated triples and executes them on the SPARQL endpoint. The queries and their constellation in the original benchmark can be found on the BSBM website at [12]. Response times are measured and logged.

The complete evaluation system was set up on Amazon's Elastic Compute Cloud (EC2) and is sketched in figure 6. One test system is always running on one EC2 instance (for the evaluation we used the default "Small Instance", see [13]). On each test system we ran both a BSBM test driver and the Stratustore.

To make the Stratustore accessible for the BSBM benchmark diver, Joseki[3] is used. Joseki is an HTTP server which offers a SPARQL endpoint and accesses RDF data via its interface to the Jena model. Behind the Jena model stands the Stratustore which in turn accesses the (remote) SimpleDB. This setup allowed to easily test the scalability in terms of parallel accesses of multiple users by starting up multiple instances of the test system. During evaluation it quickly became apparent, that two BSBM queries (2 and 5) have extremely long and useless runtimes and for time reasons we excluded them from most evaluation runs. A third query (query 11) had to be excluded, because it is not yet supported by

[3] `http://www.joseki.org/`

our implementation. We shortly detail the problem with each of these queries below.

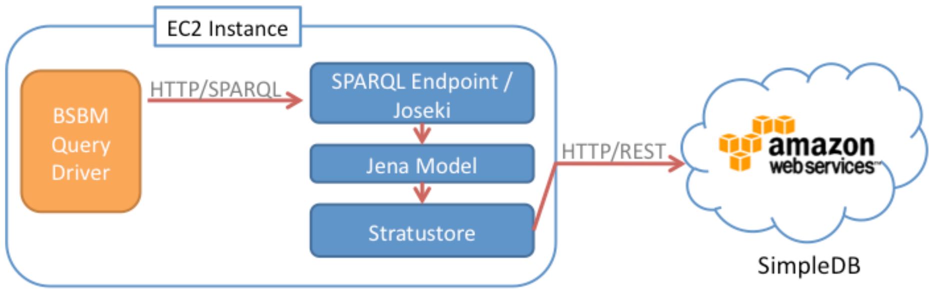

Fig. 6. The testing environment

Query 2 retrieves the labels of various subjects. The combination of query patterns of the same subject leads to a reordering of the patterns of this SPARQL query. This reordering however leads to a certain pattern not being as specific as before. In particular, not only some but all the labels of the whole database have to be retrieved. The combination of patterns is needed to take advantage of the entity oriented data model which the Stratustore is based on. In the particular case of query 2, the execution time is worsened by this behavior. A weighted sorting mechanism which prefers more specific query patterns and puts them first is thought to bring better results here.

Query 5 models a search for products with similar properties in the e-commerce use case. Naturally, this query touches a lot of data. It uses rather complex FILTER statements which, in the current implementation, cannot be mapped into SELECT queries. This forces the Stratustore to locally evaluate them which shows a bad performance. The solution to decrease the runtime of query 5 would be to push the interpretation of the FILTERs into the Stratustore.

The third query which had to be excluded from benchmarking is query 11. It contains a query statement with the the predicate as a variable. A query of the predicate is currently not supported in the Stratustore. A second index with predicates as the index key would be needed to cover this kind of query.

5 Results

Both EC2 and SimpleDB are hosted multi-tenancy services under constant development - hence it comes as no surprise, that runtimes fluctuated considerable in an unpredictable way. The fluctuations observed where within one order of magnitude. This has to be taken into account for the results reported below: albeit the BSBM benchmark ran over a considerable amount of time, actual queries might run half as fast or with double the speed in a way unpredictable to us.

Garfinkel also experienced the unpredictability of Amazon's web services [15]. He states that Amazon has a very broad service level agreement with no throughput guarantees. Thus variations in performance can and do occur on all services.

Uploading a set of 1 million triples with a total size of 86MB, took as long as 144 minutes using only one thread. The use of multi threading speeded this up by a factor of 4.6. For 10 threads, the upload took 31 minutes to complete. This means, the transfer rate to the SimpleDB is about 47 kB (or 500 triples) per second. Adding more threads or more computers does not increase the transfer rate. This restriction was identified as a limitation of SimpleDB's acceptance rate. As mentioned before, bulk uploading is thought to bring an improvement.

A performance analysis of the Stratustore was done by running the query driver from the BSBM against the Stratustore. The data generator was used to create a data set of 1 million triples. These triples were then uploaded onto the Stratustore and queried against. However, we were not able to execute all of BSBM's queries against the Stratustore. The aforementioned three queries which had abnormal runtimes of several minutes were excluded from the benchmarking.

In the following we present the results of 9 different queries being run 50 times against a triple set of 1 million triples on the SimpleDB. The amount of executed queries per second (QpS) is evaluated. This metric is used in BSBM to compare the speed of execution of single queries.

In table 1, the runtimes of the different queries are presented. The row indicates which query was evaluated. The column indicates how many instances of the Stratustore were accessing the SimpleDB at the same time. Up to 20 simultaneous instances were tested. The figures in table 1 show the amount of queries which were successfully executed per second. The amounts of the single instances were added up for the total value in each field of the table.

Concurrent instances	1	5	10	20
Query 1	10.84	71.66	157.36	334.75
Query 3	1.99	10.77	22.89	48.27
Query 4	10.66	59.38	127.78	266.44
Query 6	0.28	1.31	3.00	6.24
Query 7	1.32	9.89	30.27	65.47
Query 8	0.81	6.42	22.56	47.01
Query 9	11.45	62.88	160.08	333.3
Query 10	1.94	11.33	27.07	57.2
Query 12	0.02	0.05	0.10	0.20

Table 1. Performance of queries of the Berlin SPARQL Benchmark in queries per second

When comparing the query per second rate of single queries to existing RDF stores like Jena[4], Virtuoso[5], Sesame[6] or Mulgara[7], the Stratustore is outperformed by multiple orders of magnitude. The authors of the BSBM evaluated these stores on [14]. The figures in the text used as comparison against the Stratustore are taken from these tests.

Examining for example query 6 with a QpS rate of 0.28 when executed on a single instance. The Virtuoso Store using a local database reaches a QpS rate of 55.0 in the same setting. The Virtuoso Store is about 2 magnitudes faster than the Stratustore. Comparing the fastest query on the Stratustore shows a similar outcome. On the Stratustore, query 1 reaches a speed of 10.84 queries per second, whereas the Virtuoso store processes 202.3 of this query per second. One of the main reasons for the lower performance of single query execution on the Stratustore can be found in the overhead of transforming SPARQL queries into queries of SimpleDB's SELECT language. The lower expressiveness of SimpleDB's query language requires a time intense restructuring of queries.

The strength of the Stratustore lies in the ability to easily scale in terms of users simultaneously accessing it. This can be seen in table 1 as well. Taking query 1 as an example again, when doubling the number of concurrent users from 5 to 10, the rate of processed queries per second increases by a factor of 2.2. Again doubling the concurrent accesses to 20, the QpS rate increases by a factor of 2.1. The Stratustore is able to answer the additional queries with no performance decrease. All queries are still answered at the same speed, despite the fact that the number of concurrent queries has doubled. The unexpected speed up of more than a factor of 2 is due to different loads of the SimpleDB on different times. Running a whole query set could take up to several hours thus we could not estimate the impact of different usage loads on the SimpleDB.

The BSBM contains results for the Virtuoso store running 8 and 64 clients simultaneously. For 8 clients, Virtuoso comes to 286.84 queries per second, while for 64 clients, the rate reaches 232.94 QpS. As can be seen from table 1, the Stratustore with 20 concurrent clients is able to reach a higher throughput than the Virtuoso store with as many as 64 clients. This shows that the Stratustore is able to provide the throughput of a state-of-the-art RDF store with the potential of scaling very well in terms of concurrent users.

6 Discussion and Future Work

Is the Stratustore on SimpleDB the "worry free" triple Store that frees the user from having to worry about the complexities of handling a distributed database; from having to worry about redundancy, backups and scaling to large data sets and many concurrent database clients?

[4] `http://jena.sourceforge.net/`
[5] `http://www.openlinksw.com/virtuoso`
[6] `http://www.openrdf.org`
[7] `http://www.mulgara.org/`

On the plus side it really is an instantly available distributed and replicated database that can serve many concurrent clients and that is payed for with a utility pricing model. For queries that make the best use of the SimpleDB data model it is - for a large number of concurrent accesses - even competitive to state of the art triples stores running on dedicated servers. For these simple queries throughput and response time is sufficient for interactive applications.

On the negative side, however, the Stratustore cannot store and query arbitrary RDF data, is very slow on more complex queries and it is inherently limited for transactional data. The following paragraphs detail these points.

As stated above the Stratustore is currently bound by some of SimpleDBs limitations. For this reason it currently cannot process queries with variables in predicate position, cannot store more than 255 triples involving the same subject and cannot store values larger than 1024 bytes. There are, however, simple workarounds for each of these limitations that could be implemented in Stratustore.

A more serious problem is the slow runtime for more complex queries; a slow runtime caused by joins having to be executed on the client side. Some optimizations are possible in Stratustore that could speed this up, for example some filter statements could be "pushed down" into the SimpleDB select queries - for some cases this could radically reduce the amount of data that needs to be transferred over the network and that needs to be joined. Another optimization would be the use of summary graphs or dataset statistics to arrive at better query plans that avoid transferring too large amounts of data. Finally, if the query types that will need to be answered are known in advance, then query dependent indexes (also stored in SimpleDB) could be used to tackle this problem.

Another issue is that the lack of complex database transactions together with the model of eventual consistency mean that the Stratustore cannot be used for use cases where transactions are important. In conclusion, the Stratustore is a solution for use cases that need a simple, robust, cheap, always on, mirrored database that is accessed concurrently with a read heavy work-load where simple queries are enough. For other applications the "worry free triple store" will need either indexes adapted to the applications need or a next generation, more powerful SimpleDB.

Lastly there is the question whether a "normal" database benchmark is appropriate for a cloud service or whether different benchmarks need to be designed. For example Binnig et al. outline how benchmarking Cloud applications differs from well established benchmarking systems [16]. Cloud services cannot be tested in a managed environment where all configuration parameters are under control. Instead, they are always exposed to unpredictable variations. This loss of control has to be taken into account when designing a benchmark. They argue that Cloud services ideally scale in a linear way with the amount of queries because more resources are added as needed to fulfill the requests. Their suggestion is to increase the amount of issued queries over time and count the successful interactions over a given amount of time. We partly followed this scheme by running different test cycles and increasing the number of clients.

References

1. Lenk, A., Klems, M., Nimis, J., Tai, S., Sandholm, T.: What's inside the Cloud? An architectural map of the Cloud landscape. In: CLOUD '09: Proceedings of the 2009 ICSE Workshop on Software Engineering Challenges of Cloud Computing, Washington, DC, USA, IEEE Computer Society (2009) 23–31
2. Amazon Web Services: SimpleDB. [Online] `http://aws.amazon.com/simpledb/`.
3. Abadi, D.J., Marcus, A., Madden, S.R., Hollenbach, K.: Scalable semantic web data management using vertical partitioning. In: VLDB '07: Proceedings of the 33rd international conference on Very large data bases, VLDB Endowment (2007) 411–422
4. Abadi, D.J., Madden, S.R., Hachem, N.: Column-stores vs. row-stores: how different are they really? In Wang, J.T.L., ed.: SIGMOD Conference, ACM (2008) 967–980
5. Sidirourgos, L., Goncalves, R., Kersten, M.L., Nes, N., Manegold, S.: Column-Store Support for RDF Data Management: not all swans are white. PVLDB **1**(2) (2008) 1553–1563
6. Talis: Talis connected commons. [Online] `http://www.talis.com/platform/cc/`.
7. Vogels, W.: Eventually consistent. Communications of the ACM **52**(1) (January 2009) 40–44
8. Carroll, J.J., Dickinson, I., Dollin, C., Reynolds, D., Seaborne, A., Wilkinson, K.: Jena: implementing the semantic web recommendations. In: WWW Alt. '04: Proceedings of the 13th international World Wide Web conference on Alternate track papers and posters, New York, NY, USA, ACM (2004) 74–83
9. W3C: SPARQL Query Language for RDF. [Online] (2008) `http://www.w3.org/TR/rdf-sparql-query`.
10. Kavanagh, D.: Typica : A Java client library for a variety of Amazon Web Services. [Online] `http://code.google.com/p/typica/`.
11. Bizer, C., Schultz, A.: Benchmarking the Performance of Storage Systems that expose SPARQL Endpoints. In: Proceedings of the 4th International Workshop on Scalable Semantic Web knowledge Base Systems (SSWS2008). (2008)
12. Bizer, C., Schultz, A.: SPARQL Queries for the Berlin SPARQL benchmark. [Online] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/spec/index.html#queriesTriple`.
13. Amazon Elastic Compute Cloud Developer Guide: Instance types. [Online] `http://docs.amazonwebservices.com/AWSEC2/latest/DeveloperGuide/instance-types.html`.
14. Bizer, C., Schultz, A.: Berlin SPARQL Benchmark Results. [Online] `http://www4.wiwiss.fu-berlin.de/bizer/BerlinSPARQLBenchmark/results/index.html`.
15. Garfinkel, S.: An Evaluation of Amazon's Grid Computing Services: EC2, S3 and SQS. Technical report (2007)
16. Binnig, C., Kossmann, D., Kraska, T., Loesing, S.: How is the Weather tomorrow?: Towards a Benchmark for the Cloud. In: DBTest. (2009)