Chapter 2

Existing System : MySQL - Relational DataBase

A relational database is a database that has a collection of tables of data items, all of which is formally described and organized according to the relational model. The term is in contrast to only one table as the database, and in contrast to other models which also have many tables in one database. In the relational model, each table schema must identify a column or group of columns, called the primary key, to uniquely identify each row. Rows in one table can relate to a rows in another table by establishing a foreign key, a column or group of columns in one table that points to the primary key of another table. The relational model offers various levels of refinement of table organization and reorganization called database normalization. The database management system (DBMS) of a relational database is called an RDBMS, and is the software of a relational database. The relational database was first defined in June 1970 by Edgar Codd, of IBM's San Jose Research Laboratory. Codd's view of what qualifies as an RDBMS is summarized in Codd's 12 rules. A relational database has become the predominant choice in storing data. Other models besides the relational model include the hierarchical database model and the network model. A relation is defined as a set of tuples that have the same attributes. A tuple usually represents an object and information about that object. Objects are typically physical objects or concepts. A relation is usually described as a table, which is organized into rows and columns. All the data referenced by an attribute are in the same domain and conform to the same constraints. The relational model specifies that the tuples of a relation have no specific order and that the tuples, in turn, impose no order on the attributes. Applications access data by specifying queries, which use operations such as select to identify tuples, project to identify attributes, and join to combine relations. Relations can be modified using the insert, delete, and update operators. New tuples can supply explicit values or be derived from a query. Similarly, queries identify tuples for updating or deleting. Tuples by definition are unique. If the tuple contains a candidate or primary key then obviously it is unique, however, a primary key need not be defined for a row or record to be a tuple. The definition of a tuple requires that it be unique. The definition does not require a Primary Key to be defined. The attributes of a tuple may be referred to as a super key.

2.1 The Problem with Relational Database Systems

RDBMSes have typically played (and, for the foreseeable future at least, will play) an integral role when designing and implementing business applications. As soon as you have to retain information about your users, products, sessions, orders, and so on, you are typically going to use some storage backend providing a persistence layer for the frontend application server. This works well for a limited number of records, but with the dramatic increase of data being retained, some of the architectural implementation details of common database systems show signs of weakness. The RDBMS gives you the so called ACID properties, which means your data is strongly consistent. Referential integrity takes care of enforcing relationships between various table schemas, and you get a domain- specific language, namely SQL, that lets you form complex queries over everything. Finally, you do not have to deal with how data is actually stored, but only with higher-level concepts such as table schemas.

Chapter 3

Proposed System : HBase

Hadoop excels at storing data of arbitrary, semi-, or even unstructured formats, since it lets you decide how to interpret the data at analysis time, allowing you to change the way you classify the data at any time: once you have updated the algorithms, you simply run the analysis again. Hadoop also complements existing database systems of almost any kind. It offers a limitless pool into which one can sink data and still pull out what is needed when the time is right. It is optimized for large file storage and batch-oriented, streaming access. This makes analysis easy and fast, but users also need access to the final data, not in batch mode but using random accessthis is akin to a full table scan versus using indexes in a database system.

We are used to querying databases when it comes to random access for structured data. RDBMSes are the most prominent, but there are also quite a few specialized variations and implementations, like object-oriented databases. Most RDBMSes strive to implement Codds 12 rules, which forces them to comply to very rigid requirements. The architecture used underneath is well researched and has not changed significantly in quite some time. The recent advent of different approaches, like column-oriented or massively parallel processing (MPP) databases, has shown that we can rethink the technology to fit specific workloads, but most solutions still implement all or the majority of Codds 12 rules in an attempt to not break with tradition.

Note, though, that HBase is not a column-oriented database in the typical RDBMS sense, but utilizes an on-disk column storage format. This is also where the majority of similarities end, because although HBase stores data on disk in a column-oriented format, it is distinctly different from traditional columnar databases: whereas columnar databases excel at providing real-time analytical access to data, HBase excels at providing key-based access to a specific cell of data, or a sequential range of cells.

Hbase achieve BASE characteristics. The BASE acronym is used to describe the properties of certain databases, usually NoSQL databases. It's often referred to as the opposite of ACID.

- Basically available indicates that the system does guarantee availability, in terms of the CAP theorem.
- Soft state indicates that the state of the system may change over time, even without input. This is because of the eventual consistency model.
- Eventual consistency indicates that the system will become consistent over time, given that the system doesn't receive input during that time.

First, a quick summary: the most basic unit is a column. One or more columns form a row that is addressed uniquely by a row key. A number of rows, in turn, form a table, and there can be many of them. Each column may have multiple versions, with each distinct value contained in a separate cell. This sounds like a reasonable description for a typical database, but with the extra dimension of allowing multiple versions of each cells. But obviously there is a bit more to it. All rows are always sorted lexicographically by their row key he row keys can be any arbitrary array of bytes and are not necessarily human-readable.

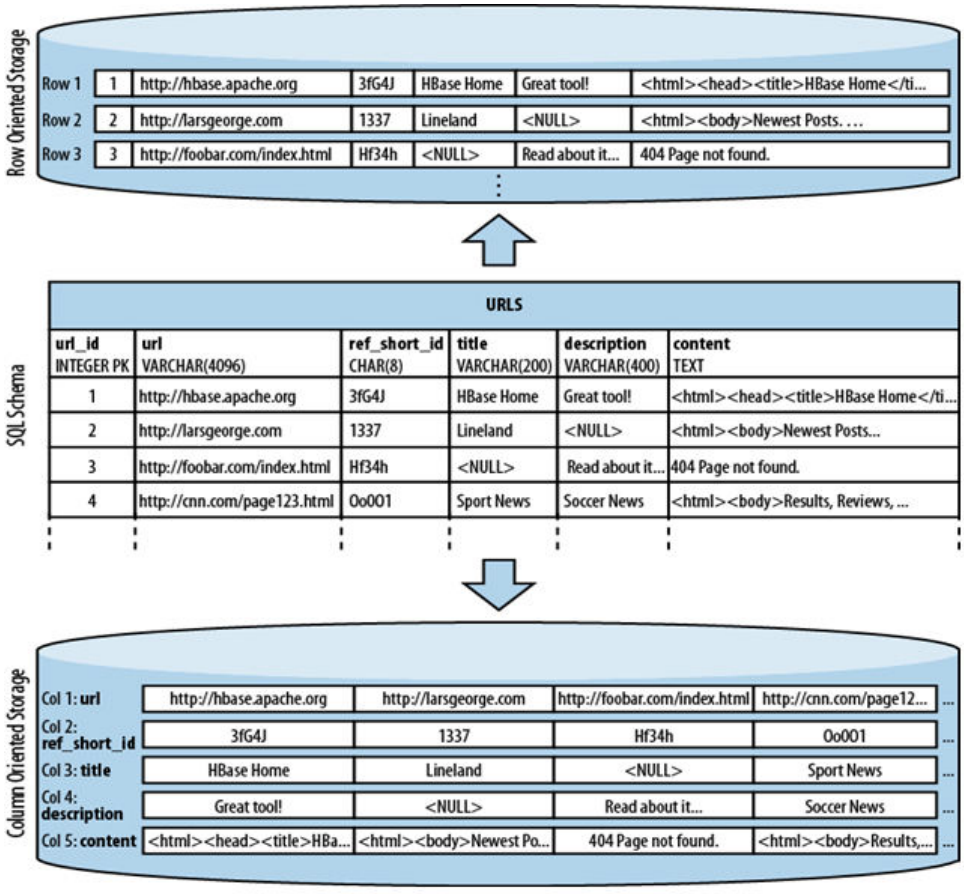

Figure 3

Rows are composed of columns, and those, in turn, are grouped into column families. This helps in building semantical or topical boundaries between the data, and also in applying certain features to themfor example, compressionor denoting them to stay in-memory. All columns in a column family are stored together in the same low- level storage file, called an Hfile. Column families need to be defined when the table is created and should not be changed too often, nor should there be too many of them. There are a few known short comings in the current implementation that force the count to be limited to the low tens, but in practice it is often a much smaller number. The name of the column family must be composed of printable characters, a notable difference from all other names or values. Columns are often referenced as family:qualifier with the qualifier being any arbitrary array of bytes. As opposed to the limit on column families, there is no such thing for

the number of columns: you could have millions of columns in a particular column family. There is also no type nor length boundary on the column values.
The Entry structure in HBase is as follows :
{rowid { columnfamily1 {column1,column2},
columnfamily2{column1,column3}
}
}

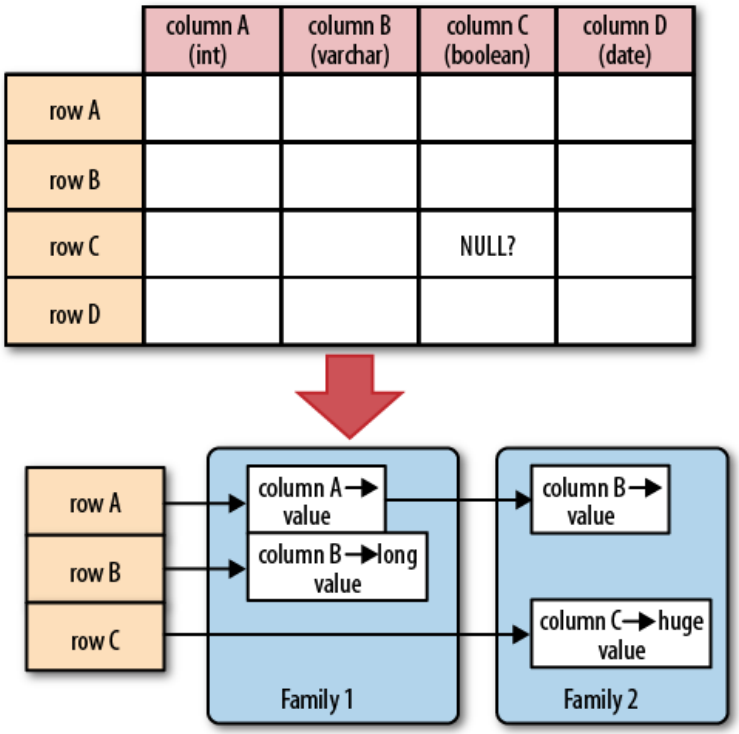

Figure 4

The basic unit of scalability and load balancing in HBase is called a region. Regions are essentially contiguous ranges of rows stored together. They are dynamically split by the system when they become too large. Alternatively, they may also be merged to reduce their number and required storage files.

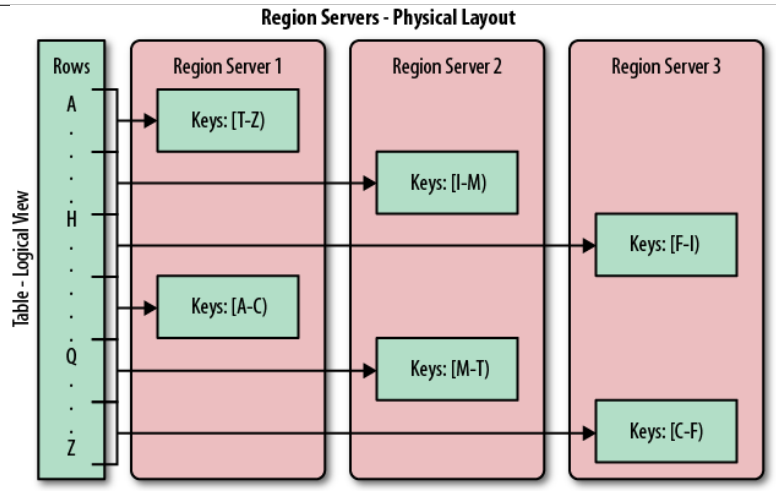
Region Servers - Physical Layout
Table - Logical View
Rows
A
H
Q
Z
Region Server 1
Region Server 2
Region Server 3
Keys: [T-Z)
Keys: [I-M)
Keys: [F-I)
Keys: [A-C)
Keys: [M-T)
Keys: [C-F)

Figure 5

Chapter 4

LUBM Benchmarking

4.1 Data Generation and OWL Datasets

1. We would like the benchmark data to be of a range of sizes including considerably large ones. It is hard to find such data sources that are based on the same ontology.

2. We may need the presence of certain kinds of instances in the benchmark data. This allows us to design repeatable tests for as many representative query types as possible. These tests not only evaluate the storage mechanisms for Semantic Web data but also the techniques that exploit formal semantics. We may rely on instances of certain classes and/or properties to test against those techniques.

4.2 Test Queries

In choosing the queries, first of all, we wanted them to be realistic. Meanwhile, we have mainly taken into account the following factors:

1. Input size. This is measured as the proportion of the class instances involved in the query to the total class instances in the benchmark data. Here we refer to not just class instances explicitly expressed but also those that are entailed by the knowledge base. We define the input size as large if the proportion is greater than 5

2. Selectivity. This is measured as the estimated proportion of the class instances involved in the query that satisfy the query criteria. We regard the selectivity as high if the proportion is lower than 10

3. Complexity. We use the number of classes and properties that are involved in the query as an indication of complexity. Since we do not assume any specific implementation of the repository, the real degree of complexity may vary by systems and schemata. For example, in a relational database, depending on the schema design, the number of classes and properties may or may not directly indicate the number of table joins, which are significant operations.

4. Assumed hierarchy information. This considers whether information from the class hierarchy or property hierarchy is required to achieve the complete answer.

5. Assumed logical inference. This considers whether logical inference is required to achieve the completeness of the answer. Features used in the test queries include subsumption, i.e., inference of implicit subclass relationship, owl:TransitiveProperty, owl:inverseOf, and realization, i.e., inference of the most specific concepts that an individual is an instance of. One thing to note is that we are not benchmarking complex description logic reasoning. We are concerned with extensional queries. Some queries use simple description logic reasoning mainly to verify that this capability is present.

Chapter 5

RDF Data Storage and Querying Architecture

To store and query RDF data in HBase and MySQL Cluster, we designed a database system with a client, middleware, and HBase and MySQL Cluster back ends (see Figure 1). The middleware hides all back-end details from the client, exposing only a conventional Semantic Web API to upload and store RDF datasets and execute SPARQL queries. Both HBase and MySQL Cluster middleware have similar high-level components, such as a storage engine and query engine. The storage engine generates database schemas and loads data according to generated schemas. The data-loading component includes algorithms for statement- by-statement, batch, and bulk data insertion. Next, the query engine for the HBase back end uses our algorithms to translate SPARQL queries into Java programs that employ the HBase API to retrieve data. The generated Java programs are compiled and executed to return query results. Finally, for the MySQL Cluster back end, the query engine relies on our algorithms to translate SPARQL queries into equivalent SQL queries and execute the latter to obtain query results.

5.1 RDF Data Storage Schemas

For both HBase and MySQL Cluster, we present our rationale for choosing a particular database schema, describing the HBase approach in more detail owing to its novelty.

5.2 HBase Storage

HBase stores data in tables that can be described as sparse multidimensional sorted maps and are structurally different from relations found in conventional relational databases. An Hbase table stores data rows that are sorted according to the row keys. Each row has a unique key and an arbitrary number of columns. A full column name comprises a column family and a column qualifier (family:qualifier). Column families are usually specified when the table is created, so their number doesnt change, but column qualifiers are dynamically added or deleted as needed. A column of a given row, denoted as a table cell, can store a list of time-stamp value pairs, such that each value has a unique time stamp. Rows in a table can be distributed over different machines in an HBase cluster and searched using two basic operations. The first operation is a table scan, and the second

is the retrieval of row data based on a given row key and (if available) columns and time stamps. Given that the table-scan access path is inefficient for large datasets, key-based retrieval works best. The sparse nature of tables makes them an attractive storage alternative for RDF data. RDF graphs are also usually sparse, because different resources are annotated with different properties and some annotations might not be stated explicitly due to inference. To support efficient retrieval of RDF data from tables in HBase, you should consider the basic querying constructs of SPARQL, such as triple patterns. At the minimum, the database should be able to retrieve RDF triples by the values of their subjects, predicates, and objects, and the arbitrary combination of these values. We propose using a database schema with two tables to store RDF triples Table Tsp stores triple subjects as row keys, triple predicates as column names, and triple objects as cell values. Table Top stores triple objects as row keys, triple predicates as column names, and triple subjects as cell values. In Figure 6, s and o denote row keys rather than columns; type, name, and memberOf are column qualifiers that belong to the same column family p; and denotes sets of cell values with time stamps omitted. Although the graphical representation shows blank values for some table cells, the row contains no information about such values or the respective columns in other words, such cells dont exist so no space is wasted. The proposed schema requires storing RDF data twicea trade-off between storage space SPA RQL queries can be executed over this HBase storage schema using the algorithms. Tables Tsp and Top can be used to efficiently retrieve triples with known subjects and objects, respectively. Retrieval of triples based on a predicate value requires a scan of one of the tables, which might not be efficient. To try to remedy this problem, we could have created a table with predicates as row keys and subjects or objects as columns. However, such a solution can only provide marginal improvements, because the number of predicates in an ontology is usually fixed and relatively small, which implies that this new table can contains only a small number of large rows (one per distinct predicate) and retrieval of any individual row is still expensive.

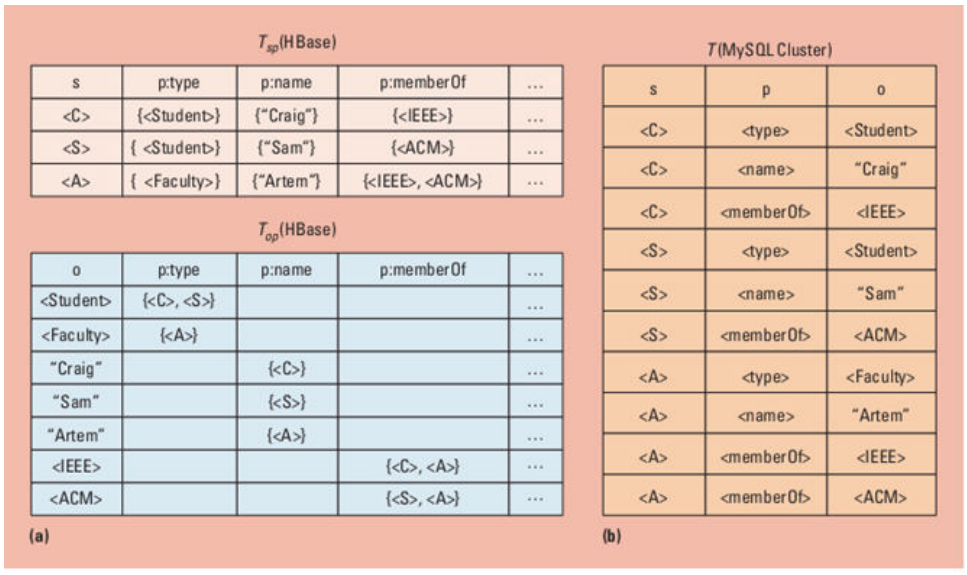

Figure 6

5.3 MySQL Cluster Storage

Relational RDF databases use several approaches to database schema generation, including schema-oblivious, schema-aware, data-driven, and hybrid strategies. These approaches feature various database relations, such as property, class, class-subject, class-object, and clustered property tables. In this work, we use a schema-oblivious approach that employs a generic schema with a single table T(s, p, o), where columns s, p, and o store triple subjects, predicates, and objects, respectively. Figure 6 shows table T, storing our sample RDF triples. Our rationale for choosing this schema is threefold. First, it can support ontology evolution with no schema modifications. The schema proposed for HBase is also flexible, because only column qualifiers can dynamically change, and such changes are performed on the row level. Second, most mentioned tables employed by relational RDF databases can be viewed as a result of horizontal partitioning of table T. However, partitioning is already performed by MySQL Cluster automatically. Finally, this schema allows lossless storage and is easy to implement. In particular, it greatly simplifies SPARQL-to-SQL translation, required to query stored RDF data.

Chapter 6

Performance Study

Here, we report our empirical comparison of the proposed approaches to distributed Semantic Web data storage and querying in HBase and MySQL Cluster.

6.1 Experimental Setup

Our experiments used nine commodity machines with identical hardware. Each machine had a late-model 3.0-GHz, 64-bit Pentium 4 processor; 2-Gbyte DDR 2-533 R A M; and an 80-Gby te, 7200-rpm Serial ATA hard drive. The machines were networked together v i a t hei r add- on g igabit Et he r ne t adapters connected to a Dell PowerConnect 2724 Gigabit Ethernet switch. They were all running 64-bit Debian Linux 5.0.7 and Oracle JDK 6. Each machine had Hadoop 0.20.2 (with a modified core library) and HBase 0.90 installed. Minor changes to the default configuration for stability included setting each block of data to replicate two times and increasing the Hbase maximum heap size to 1.2 Gbytes. MySQL Cluster 7.1.9a was used with a modified configuration based on the MySQL Cluster Quick Start Guide with increased memory available for use by the data nodes. We implemented our algorithms in Java and conducted the experiments using Bash shell scripts in an automated and repeatable manner.

6.2 Datasets and Queries

The experiments were based on the LUBM,7 a popular benchmark for RDF databases that includes the OWL university ontology, RDF data generator, and 14 test queries. We generated 11 LUBM datasets with a varying number of universities (shown in parenthesis) and triples: L1(1), L2(5), L3(10), L4(30), L5(50), L6(70), L7(90), L8(110), L9(200), L10(400), and L11(600), with L1 and L11 being the smallest and the largest datasets with 38,600 and 80,043,000 triples, respectively. The LUBM queries are expressed in a language similar to Knowledge Interchange Format (KIF), which can be found on the LUBM website. For the purpose of our experiments, we rewrote the queries in SPARQL. Because our experiments tested query performance and not reason- ing ability, we augmented each generated LUBM dataset with additional triples to produce the sample query results supplied by LUBM. Data Loading Performance We evaluated multiple data-loading methods under both HBase and MySQL Cluster, including statement-by-statement, batch, and bulk-load methods. The first graph in Figure 7 only reports the best performers for each system.

In particular, we used batch data loading1,000 triples at a timein HBase, and we used bulk loading in MySQL Cluster. We didnt use bulk loading in HBase, because its a new feature to HBase 0.90, and it involved a somewhat complicated data format, placement of the data in the Hadoop Distributed File System (HDFS), and use of the map-reduce framework to convert the plain text data into the HBase binary data format. Our simpler batch loading alternative implemented in Java showed better performance. Bulk loading in MySQL Cluster relied on the Load Data Infile statement, which loaded large data files converted from the N-Triple format to the format preferred by MySQL. The load times of LUBM datasets are reported in Figure 7. For the given data and index memory configuration, MySQL Cluster was able to load datasets up to L8. HBase successfully loaded all the datasets. MySQL Cluster initially demonstrated a significant advantage over Hbase, but this performance advantage decreased with increases in dataset size. For example, MySQL Cluster was 3 times faster than HBase on L1 and only 1.5 times faster on L8. Its possible that, with larger datasets, the advantage would be further reduced, if not eliminated. It also should be noted that HBase stored twice as many triples with tables Tsp and Top as MySQL Cluster with only one table T for each dataset, and thus HBase data loading required more work. Overall, the data loading performance proved efficient and revealed linear scalability for each back end.

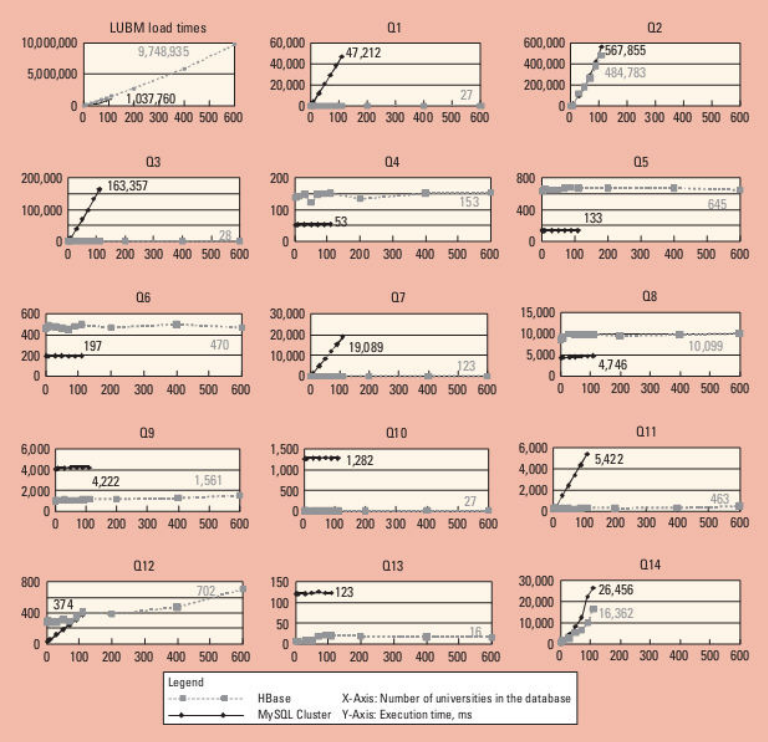

Figure 7

Chapter 7

Conclusion

Overall, the HBase approach showed better performance and scalability than the MySQL Cluster approach. Distributed Semantic Web data management is a crucial problem that must be solved to sustain the success of semantic technologies. We explored state-of-the-art cloud and relational database technologies represented by HBase and MySQL Cluster to tackle this challenge. The experimental comparison of the two approaches suggested that, while both approaches were up to the task of efficiently storing and querying large RDF datasets, the Hbase solution can deal with significantly larger RDF datasets and showed superior quer y performance and scalability. Cloud computing has a great potential for scalable Semantic Web data management. This study also identified open problems that need to be addressed, such as the architectural aspects of an RDF database management system in the cloud, system multitenancy, full-pledged SPARQL support, and inference support in distributed environments

References

[1] A. Chebotko and S. Lu, Querying the Semantic Web: An Efficient Approach Using Relational Databases: Lambert Academic Publishing, 2009.

[2] M.F. Husain et al. : Data Intensive Query Processing for Large RDF Graphs Using Cloud Computing Tools: Proc. IEEE Intl Conf. Cloud Computing,IEEE, 2010, pp. 110; http://dx.doi.org/10.1109/CLOUD.2010.36.

[3] J. Abraham et al.: Distributed Storage and Querying Techniques for a Semantic Web of Scientific Workflow Provenance: Proc. IEEE Intl Conf. on Services Computing, IEEE, 2010, pp. 178185; http://dx.doi.org/10.1109/SCC.2010.14.

[4] F. Chang et al.: Bigtable: A Distributed Storage System for Structured Data: ACM Trans. Computer Systems, vol. 26, no. 2, 2008.

[5] Guo, Z. Pan, and J. Heflin: LUBM: A Benchmark for OWL Knowledge Base Systems: J. Web Semantics, vol. 3, nos. 23, 2005, pp. 158182.

[6] Franke et al.: Distributed Semantic Web Data Management in HBase and MySQL Cluster: Proc. IEEE Intl Conf. Cloud Computing, 2011, pp. 105112; http://dx.doi.org/10.1109/CLOUD.2011.19.

[7] Lars George : HBase - The Definitive Guide: Copyright 2011 Lars George. Published by OReilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.