Archiving RDF Data in Relational Databases

By

Daming Chu, BComp

A dissertation submitted to the School of Computing in partial Fulfillment of the requirements for the degree of

Master of Computing

University of Tasmania

(NOV, 2011).

Abstract

RDF has become the de-facto standard for the representation and exchange of information. It is not only used for representing the context of Linked Open Data and semantic-Web, but also used as to publish structured data in science and business. It is also the driving force behind the increasing research interest in RDF data management. RDF is the data format for publishing liked data, and links between databases are based on database state. Failure to maintain the history of a database may lead to loss of evidence for links. In addition, regarding storage and querying, it is not reasonable to simply keep all the database versions. Therefore, this paper will develop specialized archiving technologies.

This paper have compared existing archiving techniques and the approaches of store RDF in relational databases, and based on one of existing approach, the paper developed two solution of archiving RDF data in relational databases. Also, in order to track the history of changes to the data, the paper proposes the strategies to extend existing methods for storing RDF in relational databases. For each of strategy, the paper present the efficient way for updating and archiving RDF data. Based on the strategies, the paper develops approaches about how to query these data. At last, there are evaluations of the proposed methods via different experiments

1. INTRODUCTION

With the rapid growth of the semantic web, a great deal of resource description framework (RDF) data has been created and published for knowledge sharing and information searching. According to the definition from the World Wide Web Consortium (W3C), RDF is a language which has been designed as a flexible representation of information about resources in the World Wide (RDF Primer 2004). Examples include representing information about published CDs (e.g., artist, country, company, price, year) in a web-accessible collection, or descriptive information about a user's preference for the CD's style. Recently, RDF has gain momentum in the context of Linked Open Data (LOD) and the Semantic-Web, where it is used to publish structured data in domains like life sciences and environmental monitoring, as well as in supporting Web 2.0 platforms. The RDF (Resource Description Framework) is increasingly becoming the de-facto standard for the representation and exchange of information (Duan et al 2011). It is obviously that in the recent Linked Open Data (LOD) initiative where the data from various domains such as geographic locations, people, companies, books, films, scientific data (genes, proteins, drugs), statistical data are interlinked to provide one large data cloud (Duan et al 2011). The cloud has around 200 data sources which have contributed a total of 25 billion RDF triples until of October 2010. RDF are used for many large companies and organizations as the business data representation format, either for search engine optimization, better product search semantic data integration, or for representation of data from information extraction. RDF is also used for Google and Yahoo to optimize search engine, there is a clear incentive for its growth on the web. For example, in E-science, there is an increasing support for RDF as an import/export format. In the area of life sciences, RDF also has been selected for data extractions. Finally, Web 2.0 platforms for online communities are considering RDF as a non-proprietary exchange format and as an instrument for the construction of information mash-ups (Neumann & Weikum 2009).

In RDF, all data items are represented in the form of (subject, predicate, object) triples (Klyne & Carroll 2004). Here are some examples from the CIA

World Factbook. The World Factbook (ISSN 1553-8133; also known as the CIA World Factbook) is a reference resource produced by the Central Intelligence Agency of the United States with almanac-style information about the countries of the world (CIA 2010). The Factbook is available in HTML format, which is partially updated every week. It can be downloaded for use off-line. It provides a two- to three-page summary of the demographics, geography, communications, government, economy, and military of 267 entities including U.S.-recognized countries, dependencies, and other areas in the world. It is frequently used as a resource for academic research papers. This paper also uses the CIA Factbook as the dataset for experimental evaluation. RDF can be used to describe a fact like population information for a given country from the Factbook database. For instance, current population information about Australia can be described by using the following triples (subject, predicate, object):

- (Australia, Type, Country)
- (Australia, hasCategory, People)
- (People, hasProperty, Population)
- (Population hasValue, 21,262,641)

If translating the examples as a graph, it would be shown as Figure 1:

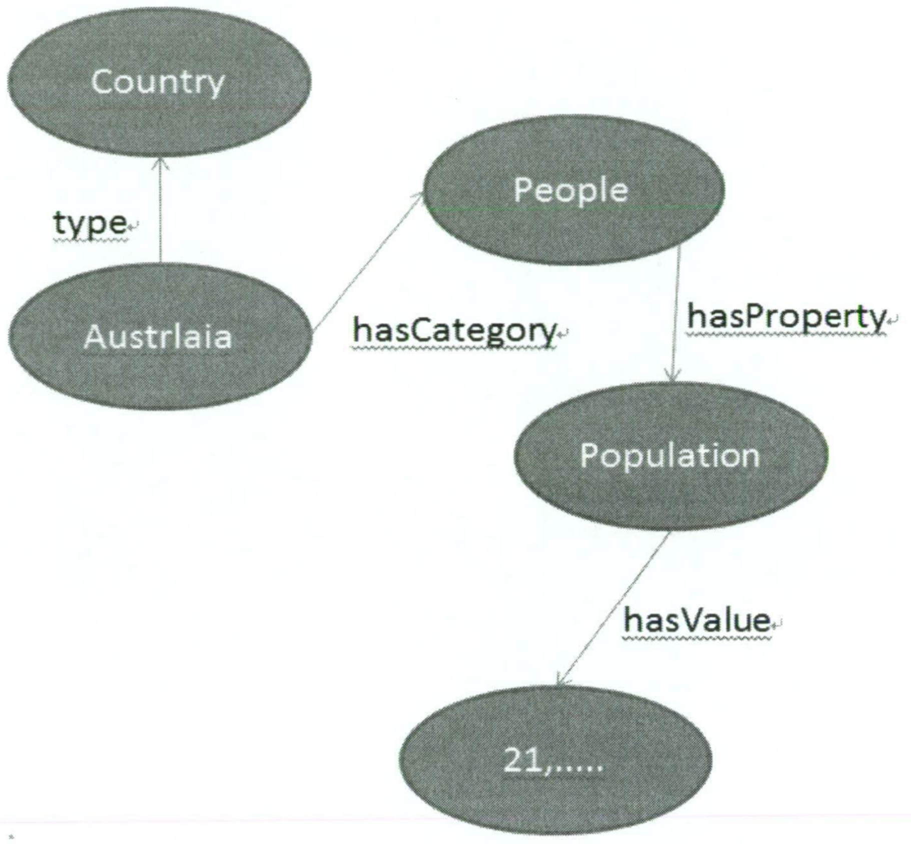


Figure 1 the fact of Australian population translate RDF triples into graph

The Australian Bureau of Statistics may want to query information about Australian population; however, with the current practice of overwriting the current database state whenever a new version is published, it is difficult to find out about the population of a specified year (say 2009) or how the population has changed since 2001. In order to be able to keep track of Australian population we will have to maintain all the past versions of the database. The same is true for scientific data, and the ability to store all the previous versions of a database is especially important for scientific data (Buneman et al 2004). Many science experiments are based on particular versions of public available databases. Failure to maintain the history of a database may lead to loss of scientific evidence as the versions may be lost and scientific findings cannot be verified later on. Therefore, maintaining all the history of data is important. Given the increased popularity of RDF, the goal of this project is to explore efficient and scalable ways for storing RDF data.

Since RDF data has become mass data, a significant problem that needs to be addressed is how to efficiently store and query large archives of RDF databases. Obviously, it is not reasonable to simply keep all the database versions. Since the RDF data will be increasingly large, completely storing all the versions is not an efficient approach regarding storage space. Additionally, when querying the database, it may cause some performance problems. Assume that all the previous versions of the database are stored in separate files. If someone wants to retrieve Australian population history, firstly, he has to scan through all these files. Secondly, he has to find population record, identify whether it has changed, and then present the result to the user. The advantage of the system that we are proposing is that this task can be done using a SPARQL query. It will be much easier for the user to get the history of Australian population.

2. RELATED WORK AND LITERATURE REVIEW

2.1 EXISTING ARCHIVING TECHNIQUES

There has been considerable research on RDF data querying and storing: Since scientific data is held in a hierarchical format and has a key structure, so [7] have utilized these features to develop an archiving technique. Archiving plays a significant role on scientific data with the function of recording all past versions of database to assist in verifying findings grounded on a specific version. Much scientific data is stored in a hierarchical format as well as in possession of key structure which is used to offer a canonical identification for each element of the hierarchy. In this research, based on these properties, an archiving technique that is not only efficient in its making use of space, but also maintains the continuity of elements by versions of the databases has been developed, all of which is not offered by the traditional minimum-edit-distance diff approaches. However, timestamps is applied into the approach. By merging all versions of data into one hierarchy, an element that appear in multiple versions is recorded only once with the assistant of a timestamp. Compared with the approaches that store a sequence of deltas where it required undoing a large number of changes or significant reasoning with the deltas, the archiving technique has capability that related with offering meaningful change description by the way of identifying the semantic continuity of elements as well as merging them into one data structure. Besides that, the archive could ensure us easily answer certain temporal queries such as restoring of any specific version from the store and seeking for the history of an element. The archive that does not result in any significant space has been proved by a suite of experiments. Besides that, considering with utilization of XML format to represent hierarchical data as well as resulting archive, it is obvious that XML could be regarded as significant tool that directly applied on their archive, which is another helpful property of their approach. One of most particular aspect is that an XML compressor is applied into their archive and compresses archive outperforms compresses diff-based repositories in space efficiency based on the results of their experiments. Lastly, they also present how they can extend their archiving tool to an external memory archive for higher scalability and introduce various index structures that can further improve the efficiency of some temporal queries on their archive.

[3, 9] have argued that there could be some problems, which was resulted by storing a sparse data set (like RDF) in multiple tables. Consequently, storing a sparse data set in a single table has been suggested while the complexities of sparse data management can be handled inside an RDBMS with the addition of an interpreted storage format (Chu, Beckmann & Naughton 2007). The proposed format starts with a header which contains fields such as relation-id, tuple-id, and a tuple length. When a tuple has a value for an attribute, the attribute identifier, a length field (if the type is of variable length), and the value appear in the tuple. The attribute identifier is the id of the attribute in the system catalog while the attributes that appear in the system catalog but not in the tuple are null for that tuple. The sparse data sets in a horizontal schema can in general be stored much more compactly in the format by the reason that the interpreted format stores nothing for null attributes. The resorting retrieving the values from attributes in tuples is more complex, whereas the storage of the interpreted format has benefits for sparse data. Actually, the format is called interpreted by the reason that the storage system must

discover the attributes and values of a tuple at tuple-access time, rather than using precompiled position information from a catalog, as the positional format allows (Beckmann et al 2006). In order to deal with this problem, there is a new operator (called EXTRACT operator) introduced to the query plans to Prior to any reference to attributes recorded in the interpreted format and returns the offsets to the referenced interpreted attribute values which is then used to retrieve the values. Due to reliance on the number attributes stored in a row or the length of the tuple, it is obvious that the Value extraction from an interpreted record is a potentially expensive operation (Chu, Beckmann & Naughton 2007). Besides that, once a query evaluation plan fetches each attribute individually and uses an EXTRACT call per attribute, the record will be detected for each attribute and will thus be very slow. Therefore, in order to in order to save time, a batch EXTRACT technique is considered as an effective method to allow for a single scan of the present values.

[26] have proposed a path-based relational RDF database. This approach primarily emphasize on improving the performance for path queries by the way of extracting all reachable path expressions for each resource and then storing them. As a result of that, in opposed to the flat tripe stores or the property tables approach, it is unnecessary to perform join operations. In this approach, each subgraph is stored by applicable techniques into distinct relational tables, which is following to separate the RDF graph into subgraphs. To more exactly, all classes and properties are extracted from RDF schema data, and all resources are also extracted from RDF data. There is a corresponding relational table, which is used to store each extracted item that assigned an identifier and a path expression (Matono et al 2005.

[37] have introduced the Hexastore RDF storage scheme with primary emphasis on scalability and generality in its data storage, processing and representation, which is based on the idea of indexing the RDF data in a multiple indexing scheme. It could treat any RDF element and treats subjects, properties and objects equally rather than discriminate against them. Without doubt, there are special index structures that built around for each RDF element type. Moreover, every possible ordering of the importance or

precedence of the three elements in an indexing scheme is materialized. Each index structure in a Hexastore centers around one RDF element and defines a prioritization between the other two elements (Weiss et al 2008). Two vectors are associated with each RDF element (e.g. subject), one for each of the other two RDF elements (e.g. property and object). In addition, lists of the third RDF element are appended to the elements in these vectors. In total, six distinct indices are used for indexing the RDF data. These indices materialize all possible orders of precedence of the three RDF elements. A clear disadvantage of this approach is that Hexastore features a worst-case five-fold storage increase in comparison to a conventional triples table (Weiss et al 2008).

[39] have proposed to decompose RDF graph into a forest of semantically correlated XML trees with two decomposition algorithms. They store them in an XML repository and rewrite SPARQL queries into XPath/XQuery queries to be evaluated in the XML repository. In order to achieve the aim of harvest, such search power requires robust and scalable data repositories which are used to store RDF data as well as support efficient evaluation of SPARQL queries. The relation model and relational database technologies for these tasks have become the primary basis for most of the existing RDF storage techniques (Auer & Herre 2005). They either keep the RDF data as triples, or decompose it into multiple relations. Once there is mis-match between the graph model of the RDF data and the rigid 2D tables of relational model, it will damage the scalability of such repositories and frequently renders a repository inefficient for some types of data and queries. [39] propose to separate RDF graph into a forest of semantically correlated XML trees, store them in an XML repository and rewrite SPARQL queries into XPath/XQuery queries to be evaluated in the XML repository. As analyzed above, this research is with the purpose of discussing the basic idea of RDF to-XML decomposition and the criteria of such decomposition in terms of correctness, redundancy and query efficiency. After that, relied on these criteria, it will propose two RDF-to-XML decomposition algorithms. Depended on the results of experimental evaluation, it illustrates that compared to the existing RDF techniques; their approach has the capabilities of improving the

efficiency of storage as well as query processing.

[11] have presented the function RDFMATCH of Oracle-based SQL table to query RDF data. By effective utilization of rich querying capabilities of SOL as well as seamless combination with queries on traditional relational data, the results of RDFMATCH table functions could be further processed. The core implementation of RDFMATCH query translates to a self-join query on triple-based RDF table store. By the way of making use of B-tree indexes and creating materialized join views for specialized subject property, the resulting query could operate efficiently. Subject-Property Matrix materialized join views is used with the aim of minimizing the query processing overheads that are inherent in the canonical triple-based representation of RDF (Eugene et al 2005). Depending on the user demand and query workloads could hold out the increment of the materialized join views. There is a special module which is offered to analyze the table of RDF triples and estimate the size of various materialized views, based on which a user can define a subset of materialized views. For a group of subjects, the system defines a set of single-valued properties that occur together. These can be direct properties of these subjects or nested properties. A property p1 is a direct property of subject x1 if there is a triple (x1, p1, x2). A property pm is a nested property of subject x1 if there is a set of triples such as, (x1, p1, x2), ..., (xm, pm, xm+1), where m > 1. For example, if there is a set of triples, (John, address, addr1), (addr1, zip, 03062), then the zip property is considered as a nested property of John (Eugene et al 2005).

2.2 STORING RDF AND APPROACHES FOR STORING RDF USING RELATIONAL DATABASES

There are several approaches to storing RDF data:

- **Keeping RDF data in RAM.** It is efficient but just can tackle small data due to the limitation to the storage capacity. In addition, keeping RDF data in RAM cannot store data persistently, that is, the data will get lost once the system is down. Thus, this approach is only suitable for small–scale and non-persistent data.

- **Keeping data in files.** This approach can read the data from files to RAM, and then operate on the data in RAM, and write the updated data from RAM back to files. The advantage of this approach is simplicity, but it is inefficient because this approach needs to read and write files frequently (Brickley & Guha, 2004). Moreover, it is also not appropriate for mass data manipulating.

- **Keeping data in native XML/RDF databases.** XML databases support storing and querying data with a special XML/RDF format (Abadi et al 2007). Since RDF's representation is based on XML syntax, RDF data can be saved in XML database. The advantage is that XML databases start to become powerful enough to maintain and large amounts of data. However, querying these databases in general is inefficient and querying XML/RDF in particular is not very user friendly.

- **Keeping RDF in relational databases.** Many researches have verified that relational database management systems are very efficient, scalable and successful in storing and querying RDF data. Since the RDF data can be represented in the form of (subject, predicate, object) triples, these triples can be stored in a relational database with an intuitive schema in a single table. This table has three columns, subject, predicate and object. For instance, to represent the fact that "Australian birth rate is 15births/1,000", we can use triples:

- (Australia, Type, Country)
- (Australia, hasCategory, People)
- (People, hasProperty, Birth rate)
- (Birth, hasValue, 15 births/1,000 population)

The first advantage of this approach is that it allows storing and querying large amounts of RDF data in an efficient way. The linked data initiative and the semantic web will generate billions of triples in the near future and relational database management systems by now are advanced enough to;

provide secure and reliable technologies to persistently store and manipulate large amounts of RDF data.

The advantages regarding storage and querying large-scale data render relational databases an ideal candidate for archiving RDF data. Relational databases provide a scalable off-the-shelf solution to data storage and I therefore will use relational databases to store and query RDF data.

There are some different ways of storing RDF data in a relational database. [31] have presented a classification of the relational RDF stores:

• **Triple (Vertical) Approach**: Each RDF triple (subject, predicate, object) is stored in a three-column schema directly.

Triple-stores have been developed using relational databases for a long time. They are implemented in databases, with only three columns in a schema corresponding to the three components of a RDF triple, respectively (Sintek & Decker 2001). Triple-stores are easy to realize with relational technologies and the schema has a simple and intuitive structure. However, since all data are in one table, this causes many self-join operations when translating a SPAQL query expression into a SQL query. Thus, querying data in databases using the vertical triple-store approach can be inefficient. As a result, many approaches try to overcome this limitation by creating the exhaustive set of indexes and relying on fast processing of merge joins.

Subject	Predicate	Object
Australia	hasName	Australia
Australia	type	Country
Australia	hasCategory	People
People	hasName	People
People	type	Category
People	hasProperty	Population
Population	hasName	Population
Population	hasValue	16,923,478 (July 1990)

Population	type	Property

Table 1: RDF triples-store in relational database

Table 1 shows that some RDF triples are stored in relational database by triples-stroe approach, if someone wants to query the population of Australia.

In SPARQL, it would be:

```
PREFIX factbook: <http://www.csiro.au/au/CIAWFB/ns#>
SELECT ?pvalue
  WHERE {
   ?country <factbook:type> <factbook:Country> .
   ?country <factbook:hasName> "Australia" .
   ?country <factbook:hasCategory> ?people .
   ?people <factbook:hasName> "People" .
   ?people <factbook:hasProperty> ?population .
   ?population <factbook:hasName> "Population" .
   ?population <factbook:hasValue> ?pvalue
}
```

In SQL, it would be:

```
SELECT F.object
    FROM  rdfdataset as A, rdfdataset as B, rdfdataset as C, rdfdataset as D
          rdfdataset as E, rdfdataset as F
        WHERE
                A.predicate = "hasName" AND
                A.object = "Australia" AND
                B.predicate = "hasCategory" AND
                A.subject = B.subject AND
                B.object = C.subject AND
                C.predicate = "hasName" AND
                C.object = "People" AND
                D.subject = C.subject AND
```

D.predicate = "hasProperty" AND

D.object = E. subject AND

E.predicate = "hasName" AND

E.object = "Population" AND

E.subject = F.subject AND

F.predicate = "hasValue";

Form the example, we can find out that the more triple patterns there are in the SPARQL query the more self-joins are necessary in the corresponding SQL query

[28] have described the RDF-3X (RDF Triple express) engine for querying large-scale RDF data. With creating the exhaustive set of indexes and relying on fast processing of merge joins, it tries to overcome the criticism that triples storing may cause a large amount of self-joins. The physical design of RDF-3x is based on regardless of workloads, and enable to eliminate the need for physical design tuning. It does these by creating indexes over all 6 permutations {spo,sop,pos,pso,osp,ops}of the three dimensions that constitute an RDF triple. In addition, indexes over count-aggregated variants for all three two-dimensional and all three one-dimensional projections are building. Using these 6 indexes is able to enhance the efficiency of querying data. Following the RISC-style design philosophy (Chaudhuri & Weikum 2000), the query processor relies mostly on merge joins over sorted index lists by using the full set of indexes on the triple tables. The query optimizer depends on its cost model that mostly focuses on join order and the generation of execution plans and finding the lowest-cost execution plan. In theory, selectivity estimation has a huge effect on plan generation. By virtual of this is a standard problem in database systems, the schema-free nature of RDF data causes the problem to more challenging.

RDF-3X makes use of dynamic programming for plan enumeration, with a cost model based on RDF-specific statistical synopses.

It relies on two kinds of statistics (Neumann & Weikum 2009):

1) Specialized histograms which are generic and can handle any kind of triple patterns and joins. The disadvantage of histograms is that it assumes independence between predicates.

2) Frequent join paths in the data which give more accurate estimation. During query optimization, the query optimizer uses the join path selectivity information when available and otherwise assumes independence and use the histograms information.

As a result, in the situation of uncompressed indexes, the overhead for data storage is six times its original overhead, but it can change to double overhead for data storage from sextuple.

• **Property table stores:** Multiple RDF properties are modeled as n-ary table columns for the same subject.

Due to a large amount of shelf-joins involved in the Triples-store approach, the researchers proposed two types of property tables to speed up queries over the triple-stores.

Jena is an open-source toolkit for Semantic Web programmers (McBride 2002). It implements archiving RDF graphs using an SQL database through a JDBC connection. The schema of the first version of Jena is combined by a resources table (JENA API 2009), a statement table, and a literals table. The statement table (Subject, Predicate, ObjectURI, ObjectLiteral) referenced the resources and literals tables for subjects, predicates and objects and included all statements. Two columns were used for differentiating literal objects from resource URIs. The literals table contained all literal values and the resources table contained all resource URIs in the graph (McBride 2002). However, every query operation needed to multiple joins between the statement table and the literals table or the resources table. To tackle with this problem, space has been sacrificed to achieve for saving time in Jena 2. It uses a schema in which resource URIs and simple literal values are stored directly in the statement table. Column values are encoded with a prefix that indicates the type of the value in order to distinguish database references from literals and

URIs. Literal values whose length exceeds a threshold such as blobs are stored in a separate literals table. Likewise, long URIs is stored in a separate resources table. By virtual of storing values directly in the statement table, it is possible to run many queries without a join. However, since the same value (literal or URI) is stored repeatedly, the schema uses up a large amount of database space. The solution of increasing database space consumption is using string compression schemes. Jena allows multiple graphs to be stored in a single database instance. All graphs were stored in a single statement in Jena1. However, because Jena2 is able to support the use of multiple statement tables in a single database, applications can flexibly map graphs to different tables. In this way, graphs may be stored by two ways, those are often accessed together may be stored together, the others are hardly accessed together may be stored separately (Sakr & AI-Naymat 2009).

Basically, applications usually have access patterns in which certain subjects and/or properties are accessed together. For instance, a graph of data about country might have many occurrences of objects with properties name, location, population, climate that are referenced together. Jena2 uses property table as a general facility for clustering properties that are commonly accessed together (Guha 2001). A property table is a separate table that stores the subject-value pairs related by a particular property. Another property table stores all instances of the property in the graph where that property does not appear in any other table used for the graph. In Jena1, each query is evaluated with a single SQL select query over the statement table. In Jena2, due to there can be multiple statement tables for a graph, queries have to be generalized (Sakr & AI-Naymat 2009). Using the knowledge of the frequent access patterns to construct the property-tables and influence the underlying database storage structures can provide a performance benefit and reduce the number of join operations during the query evaluation process. Table 2 illustrates the example of property table store

Property table:

subject	hasName	type	hasCategory

Australia	Australia	Country	People
People	People	Category	NULL
Population	Population	Property	NULL

Othertriples table:

subject	predicate	object
People	hasProperty	Population
Population	hasValue	16,923,478 (July 1990)

Table 2 : Property table store

Also, the SQL query for retrieving the population of Australia would be:

```
SELECT B.object
    FROM Property as X, Property as Y, Property as Z,
        Othertriples as A, Othertriples as B
        WHERE
            X.hasName = "Australia"      AND
            X.hasCategory = Y.subject    AND
            Y.hasName = "People"         AND
            Y.subject = A.subject        AND
            A.object = Z.subject         AND
            Z.hasName = "Population"     AND
            Z.subject = B.subject
```

The most important advantage of the property tables is that they can reduce subject-subject self-joins of the triples table.

However, the disadvantage of property tables is that RDF data usually not to be very structured and not all the properties have been defined for every subject listed in the table (Berners-Lee, Handler & Lassila 2001). The more NULL values will exist in the table that is caused by the less structured the data. In fact, these representations can be extremely sparse – containing hundreds of NULLs for each non-NULL value. These NULLs impose a substantial performance overhead.

The second problem with property tables is the abundance of multivalued attributes found in RDF data (Abadi et al 2007). Multi-valued attributes are surprisingly prevalent in the Semantic Web. In general, there always seem to be exceptions, and the RDF data model provides no disincentives for making properties multi-valued.

Multi-valued properties are problematic for property tables for the same reason they are problematic for relational tables. They cannot be included with the other attributes in the same table unless they are represented using list, set, or bag attributes. However, this requires an object-relational DBMS, results in variable width attributes. (Abadi et al 2007)

[25] have introduced another property table approach in relation with storing RDF data without any assumption about the query workload statistics. The primary objectives of this approach are (Levandoski & Mohamed 2009): (1) reducing the number of joins operations which are required during the RDF query evaluation process by storing related RDF properties together (2) reducing the need to process extra data by tuning null storage to fall below a given threshold. A tailored schema is offered to each RDF data set through this approach, which represents a balance between property tables and binary tables and is based on two main parameters: 1) Support threshold which represents a value to measure the strength of correlation between properties in the RDF data. 2) The null threshold which represents the percentage of null storage tolerated for each table in the schema. Besides that, the approach involves two phases: Clustering and partitioning. During the phase of clustering, the RDF data are scanned to automatically discover r groups of related properties. Based on the support threshold, each group of n properties which are grouped together in the same cluster are good candidates to constitute a single n-ary table and the properties which are not grouped in any cluster are good candidates for storage in binary tables. The partitioning phase is with the purpose of checking the formed clusters and balancing the tradeoff between storing as many RDF properties in clusters as possible while keeping

null storage to a minimum based on the null threshold. One of the main concerns of the partitioning phase is twofold (Levandoski & Mohamed 2009). Firstly, it is necessary to ensure that no overlap exists between the clusters as well as that the existence of each property should be in a single cluster. Secondly, during the process of query, it is obvious to reduce the number of table accesses and unions necessary.

In summary, while property tables can significantly improve performance by reducing the number of self-joins and typing attributes (Theoharis, Christophides & Karvounarakis 2005), they introduce complexity by requiring property clustering to be carefully done to create property tables that are not too wide, while still being wide enough to answer most queries directly. Ubiquitous multi-valued attributes cause further complexity. In addition, though property tables are very good at speeding up queries that can be answered from a single property table; they require joins or unions to combine data from several tables.

• **Horizontal** (vertically partitioned) **table stores**: RDF triples are modeled as one horizontal table or a set of vertically partitioned binary tables (one table for each RDF property)

[1] first presented the idea of using a fully decomposed storage-model to store RDF data. In this approach, the triple table is rewritten into *n* two-column tables where *n* is the number of unique properties in the data (Abadi et al 2007). All triples that have the same predicate are grouped in the same tables. Thus the triples table will be partitioned into n two-column tables where n is the number of unique properties in the data. In each of these tables, the first column contains the subjects that define that property and the second column contains the object values for those subjects while the subjects that do not define a particular property are simply omitted from the table for that property. In every two-column table, predicate is the name of the table, subject is the first column and object is the second column. Each table is sorted by subject, thus specific subjects can be queried quickly. In addition, since tables are sorted by subject, fast merge joins are available to reconstruct information

about multiple properties for subsets of subjects. For a multi-valued attribute, each distinct value is listed in a successive row in the table for that property. Moreover, optionally indexing the value column for each table is also feasible (or a second copy of the table can be created clustered on the value column). Table 3 shows the relational representation of vertically partitioned approach.

hasName:

subject	object
Australia	Australia
People	People
Population	Population

Type:

subject	object
Australia	Country
People	Category
Population	Property

hasCategory:

subject	object
Australia	People

hasProperty:

subject	object
People	Population

hasValue:

Subject	object
Population	16,923,478 (July 1990)

Table 3: vertically partitioned store

The SQL query for retrieving the population of Australia would be:

SELECT hasValue.object

FROM hasName, type, hasCategory, hasProperty

```
WHERE
        hasName.object = "Australia"
        AND
        hasName.subject = hasCategory.subject
        AND
        hasCategory.object = hasName.subject
        AND
        hasName.object = "People"
        AND
        hasName.subject= hasProperty.subject
        AND
        hasProperty.object = hasValue.subject
        AND
        hasName.object = "Population";
```

The advantage of this approach is that each triple is (Sidirourgos et al 2008):

1). in the decomposed storage model, a multi-valued attribute has been no longer a problem. Each distinct value is listed in a consecutive row in the table for that property if a subject has more than one object value for a particular property. For example: if Australia has 2 categories, the table would be:

Australia	Economy
Australia	People

2).Support for heterogeneous records. Subjects that do not define a particular property are simply omitted from the table for that property. In the example above, category Type is defined for one subject (Australia), the table therefore can be kept small (NULL data need not be explicitly stored). The advantage becomes increasingly significantly if the data is not well-structured.

Only those properties accessed by a query need to be read. I/O costs can be substantially reduced.

Fewer unions and fast joins. Since all data for a particular property is stored in the same table, union clauses in queries are less for using. Although the vertically partitioned approach will require more joins relative to the property table approach, properties are joined using simple, fast (linear) merge joins.

A disadvantage of this approach is that when querying several properties, these two-column tables will be merged, so there are some costs for merge join. Also, inserts can be slower into vertically partitioned tables, since multiple tables need to be accessed for statements about the same subject. Additionally, [33] have identified drawbacks for the vertically-partitioned approach regarding complexity of generated SQL queries and query execution efficiency. If a query is not isolated to access a predefined number of properties, the SQL code becomes large and complex. It challenges the capabilities offered by most optimizers. Moreover, if the property in a query is bound to a variable, then the rows returned from each property table must be union-ed. In the case where the property is not part of the result, then the union operator must also perform a duplicate elimination. Finally, since the data is not clustered on objects, a query which joins on objects, will not allow the use of a fast (linear) merge join.

From [33]'s evaluations, they have compared the triple-store RDF storage solution with the others approach, when they implemented using a state-of-the-art commercial row-store engine, [33] conclude that once the proper clustered indices are used, the triple-store performs better than the vertically-partitioned approach and others. [33] shows that the vertically-partitioned approach exhibits better query execution times with column-store implementation. However, they present that if the number of properties in an RDF data-set is high, there are potential scalability problems for the vertically-partitioned approach. With a larger number of properties, the triple-store solution manages to outperform other approaches on their column-store implementation as well. Combined with the fact that other approach is data-dependent, they show that the 3 column approach shows good performance in most cases. In addition, 3-column approach is one of the common methods to store RDF. Therefore, this paper's solutions are based on 3-column approach.

3 PROPOSAL

From the Section 2, we can conclude that the vertically partitioned table stores causes complexity of generated SQL queries and query execution efficiency, property table approach have NULL values problem – it will exist in the table that causes substantial performance overhead, and 3-column approach have the best performance in most cases. Therefore, the goal of this project is to explore efficient and scalable ways for archiving RDF data using relational database technologies.

Specifically, I will make the following contributions:

1) **I will propose the strategies to extend existing methods for storing RDF in relational databases in such a way that the history of changes to the data can be tracked**. The strategies will be easy to generalize----they should not depend on a particular schema of the RDF data; the strategies will have persistent structure----changing the structure of the RDF data will not cause a change of the relational database schema, and efficient to query.

2) **For each of proposed strategy I will describe how to update and archive RDF data in an efficient way.** In addition, I also intend to develop approaches about how to query these data (in contribution 3)).

3) **For each proposed extensions there has to be a method describing how to translate queries into SQL queries of the respective schema.** I will be especially interested in answering three types of queries:

a) How did the data look like at a certain point in time (e.g. what was Australian population salary in version 1st March 2003?

For this question, the query can find Australian population in a given version that was valid on 1st March 2003.

b) How did the value of some object change (e.g. how did Australian population change over the past 5 years?)

The answer should list all the population of Australia in past 5 years by joining all the related versions.

c) What is the difference between two versions (e.g. the different between version 65(2001) and version 67 (2003)).

Using these queries, one can query RDF data in relational database to track how data changed and where or when a key fact happened.

4) The last part of my project will be **an evaluation of the proposed methods.** The proposed methods will be evaluated regarding:

a) Storage space:

Large-scale data like library data will be used to evaluate the capacity of storage, and compare with other approaches.

b) Efficiency of merging a new version into the archive:

Operations include bulk updates and single updates. The evaluation criteria will be execution time and complexity of implementing the approach.

c) Query answering

This part evaluate the complexity of SQL statements to answer the queries and how efficient (time) for the queries to be answered.

4 IMPLEMENTED APPROACH

Two solutions (4- and 5-column) are proposed which are based on 3- column approach. Additionally, two RDBMS are used for implementation, namely MySQL and PostgreSQL.

4.1 4-COLUMN SOLUTION

The idea of the 4-column solution is to add a fourth column for storing the version info, and to introduce a "version_info" table for maintaining details of individual database versions. Table 4 illustrates the solution.

RDFSET:

Subject	Predicate	Object	Version
Australia/People/Birth rate	hasValue	14.43 births/1,000 population	1
Australia/People/Birth rate	hasValue	14.29 births/1,000 population	2
Australia/People/Birth rate	hasValue	14.13 births/1,000 population	3
Australia/People/Birth rate	hasValue	13.99 births/1,000 population	4
Australia/People/Birth rate	hasValue	13.73 births/1,000 population	5

VERSION_INFO:

id	available_time
1	1993 est.
2	1994 est.
3	1995 est.
4	1996 est.
5	1997 est.

Table 4:4-column solution for archiving RDF data

In 4-column solution, when merging a new version database into database, we just need to create a new version id in VERSION_INFO table, and then insert the new triples into RDFSET table with the new version id.

With this solution, users can query the history of data. For example, if someone wants to query the Australian Birth rate in 1995, users query the VERSION_INFO to find out the corresponding id for 1995, and then they can retrieve the birth rate in 1995.

The solution is a simple way to archiving RDF data; however, it has some drawbacks. First, it has to store redundant triples, even the updating triples have same subject, predicate and object with the triples in the archive, but in 4-column they are two different tuples with the different versions; second, answering queries may become slower due to the size of the relation and the necessary self-joins in the queries. Nevertheless, the solution offers a basic idea for how to archive RDF data and track the data. Based on it, the 5-column solution is next proposed.

4.2 5-COLUMN SOLUTION

The main idea of the 5-column solution is to use two columns to indicate the period when the triples are valid. Table 5 shows 5-column solution's table structure.

RDFSETA:

Sub	Pre	Obj	Startversion	Endversion
Australia/People/Birth rate	hasValue	12.26 births/1,000 population	16	16
Australia/People/Birth rate	hasValue	12.14 births/1,000 population	17	17
Australia/People/Birth rate	hasValue	12.02 births/1,000 population	18	18
Australia/People/Birth rate	hasValue	11.9 births/1,000 population	19	20
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	22
Australia/People/Birth rate	hasValue	12.47 births/1,000 population	23	-1

RDFSETV:

Sub	Pre	Obj
Australia/People/Birth rate	hasValue	12.26 births/1,000 population
Australia/People/Birth rate	hasValue	12.14 births/1,000 population
Australia/People/Birth rate	hasValue	12.02 births/1,000 population
Australia/People/Birth rate	hasValue	11.9 births/1,000 population
Australia/People/Birth rate	hasValue	12.55 births/1,000 population
Australia/People/Birth rate	hasValue	12.47 births/1,000 population

VERSION_INFO:

Id	Available_time
16	2004
17	2005
18	2006
19	2007
20	2008
21	2009
22	2010
23	2011

Table 5 5-column solution for archiving RDF data

This solution has two columns to identify the available time of the triples. For example, the record from above (Australia/People/Birth rate, hasValue, 12.26 births/1,000 population, 16, 16) means this record was inserted into Database from "16" (in accordingly VERSION_INFO table, it is 2005) and no longer available from "17" (which means there was a newer record has updated Australia/People/Birth rate with a new value); the record (Australia/People/Birth rate, hasValue, 12.47 births/1,000 population, 23, -1) means the record is available from version "23" and "-1" means it is still available now.

Since 5-column solution has 2 additional columns to indicate the period when triples are valid, the first advantage of it is that it can reduce the redundant triples. Compared with 4-column solution, 5-column does not have to store all the triples which have same subject, predicate and object but different version ids in table, 2 additional columns are able to identify the available period of the triples. The second advantage of this solution is efficiency of querying. In the 4-column solution, the query for a certain fact will be much slowly, because there are numerous tuples have same subject, predicate and object. But in 5-column solution, the querying has better performance than 4-column's due to reducing the superfluous triples.

To implement this solution, we need to create three tables in the relational database schema. RDFDATASETA is the archival table, VERSION_INFO is the version information table, and RDFSETV is the table that contains the next version of the data. The records in RDFSETV are the data set which will be inserted into RDFSETA. After transforming RDF data from XML/RDF, N3 and TURTLE into (subject, predicate, object) format, the data are stored in RDFSETV table. Before the records in the next database version (RDFSETV) can be inserted into RDFSETA, information about the new version has to be stored in VERSION_INFO. Therefore, a new id with current date is inserted into VERSION_INFO at first. The column "Id" in VERSION_INFO is designed as an AUTO_INCREMENT (called in MySQL) or SERIAL (called in PostgreSQL) column to store the new version information automatically. As a result the next version id is created in the VERSION_INFO with the current time before the next database version is inserted into RDFSETA.

4- and 5-column solutions include single operations and bulk insert. First, single operations are used to update the data without RDFSETV. In this scenario, the archive is updated directly using single triple modification operation. These operations are insertSingelTriple, deleteSingleTriple and updateSingleTriple. Second, in bulk insert, we consider the archive as a data warehouse for versions. The data is maintained elsewhere and we want to "dump" full versions of the data into the archive at certain points in time in order to keep the history of the data. However, since 4-column as introduced before, it is easy to implement, so this paper focus on the 5-column solution implementation.

4.2.1 single archiving RDF data in RDBMS

In single operations, a triple does "not exist" means there is no tuple in the current archival table that matches the triple on subject, predicate, and object and that has an end-value of -1.

1) insertSingleTriple: insert a single triple into RDFSETA means a new triple will be stored in the database with its corresponding version id. The first step is to check whether the inserting record exists in the RDFSETA. So the related method exist(Triple)'s SQL statement is as follows:

```
"SELECT COUNT (*)
    FROM RDFSETA
    WHERE
        sub = Triple.sub AND pre = Triple.pre AND
        obj = Triple.obj AND endversion = -1;"
```

If count (*) returns 0, a new version id will be created in VERSION_INFO and stored in a Java variable. The Java code for the related method createNextVersion () code is:

```
String sql =
"insert into versioninfo (availabletime) values (current_timestamp);";
            st = conn.createStatement();
            st.execute(sql, Statement.RETURN_GENERATED_KEYS);
            rs = st.getGeneratedKeys();
            rs.next();
            nextversion = rs.getInt(1);
```

The third step to insert the single triple into Database with created version and endversion-value of -1.

The related code of insertSingleTriple (Triple, start_version) is:

```
public void insertSingleTriple(Triple triple) {
    if (!this.exists(triple, " RDFSETA")) {
      int version = this.createNextVersion();
      this.insertTriple(triple, version);
    }
}
```

Table 6 shows that table RDFSETA and VERSION_INFO before and after insert the triple (Australia/People/Urbanization/rate of urbanization, hasValue, 1.2% annual rate of change)

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	-1

Id	Available_time
1	1990
2	1991
…	…
21	2009

Table 6.1 before insert the triple, RDFSETA and VERSION_INFO

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	-1
Australia/Urbanization/rate of urbanization	hasValue	1.2% annual rate of change	22	-1

Id	Available_time
1	1990
2	1991
…	…
21	2009
22	2010

Table 6.2 after insert the triple, RDFSETA and VERSION_INFO

2) deleteSingleTriple: the meaning of "delete" is a little bit different from

normal meaning of "delete" in database operation. The deleted records in RDF archival table mean their information is no longer available, but in order to track the history of data, we also need to keep them in Database. Similar to insertSingleTriple, deleteSingleTriple also needs to check whether the triple exists in the archival table; if yes, a new version id is created, and the end-value of the deleting record is set to version_id – 1 by deleteTriple (Triple, id). The corresponding SQL statement is as follows:

```
UPDATE RDFSETA SET endversion = id
    WHERE
        sub = Triple.sub AND pre = Triple.pre AND obj = Triple.obj;
```

To set the end interval to new version -1 is in the implementation of deleteSingleTriple(), the Java code of deleteSingleTriple () is as follows:

```
public void deleteSingleTriple(Triple triple) {
    if (this.exists(triple, " RDFSETA ")) {
        int version = this.createNextVersion();
        this.deleteTriple(triple, version - 1);
        }
}
```

Table 7 shows that table RDFSETA and VERSION_INFO before and after delete the triple in RDFSETA.

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	-1

Id	Available_time
1	1990
2	1991
…	…
21	2009

Table 7.1 before delete the triple, RDFSETA and VERSION_INFO

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	21

Id	Available_time
1	1990
2	1991
…	…
21	2009
22	2010

Table 7.2 after delete the triple, RDFSETA and VERSION_INFO

3) updateSingleTriple: update the triple is that there is an existing triple in archival table, it needs to insert a new triple with same subject, predicate and a different object. Thus, there are two steps to implement the function. First the triple is deleted by deleteSingleTriple, after that, the triple with new object's value is inserted into RDFSETA by insertSingleTriple. The method code is:

```
public void updateSingleTriple(Triple oldTriple, Triple newTriple) {
    if ((this.exists(oldTriple," RDFSETA "))
    &&(!this.exists(newTriple," RDFSETA"))) {
        int version = this.createNextVersion();
        this.deleteTriple(oldTriple, version - 1);
        this.insertTriple(newTriple, version);
        }
}
```

Table 8 shows that table RDFSETA and VERSION_INFO before and after update the triple (Australia/People/Birth rate, hasValue, 12.55 births/1,000 population) with (Australia/People/Birth rate, hasValue, 12.47 births/1,000 population) in RDFSETA.

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	-1

Id	Available_time
1	1990
2	1991
…	…
21	2009

Table 8.1 before update the triple, RDFSETA and VERSION_INFO

Subject	Predicate	Object	Start	end
Australia/People/Birth rate	hasValue	12.55 births/1,000 population	21	21
Australia/People/Birth rate	hasValue	12.47 births/1,000 population	22	-1

Id	Available_time
1	1990
2	1991
…	…
21	2009
22	2010

Table 8.2 after update the triple, RDFSETA and VERSION_INFO

4.2.2 bulk archiving RDF data in RDBMS (MySQL and PostgreSQL)

Bulk insert is to store the next database version (RDFSETV) into archive (RDFSETA), and update existing triples in archive in order to track the history of the data. In 5-column solution, author defined 3 operations to archive RDF data.

1) Load data: RDF data is represented by XML/RDF, N3 and TURTLE format [15], and inserted into table RDFSETV in MySQL or PostgreSQL. Since the data set is huge, these inserts have started to become a bottleneck in the bulk archiving RDF data. In [21]'s research, he indicate that people were doing the classic batch inserts using a PreparedStatment and executeBatch on MySQL. The inserts that was too slow, even after adding the parameter rewriteBatchedStatements to users' JDBC URL. Fortunately, MySQL Connector/J 5.1.3 and later include two additional methods:

MySQL. The inserts that was too slow, even after adding the parameter rewriteBatchedStatements to users' JDBC URL. Fortunately, MySQL Connector/J 5.1.3 and later include two additional methods:

setLocalInfileInputStream() sets an InputStream instance that will be used to send data to the MySQL server for a LOAD DATA LOCAL INFILE statement rather than a FileInputStream or URLInputStream that represents the path given as an argument to the statement (MySQL API 2010).

This stream will be read to completion upon execution of a LOAD DATA LOCAL INFILE statement, and will automatically be closed by the driver, so it needs to be reset before each call to execute*() that would cause the MySQL server to request data to fulfill the request for LOAD DATA LOCAL INFILE (MySQL API 2010).

getLocalInfileInputStream() returns the InputStream instance that will be used to send data in response to a LOAD DATA LOCAL INFILE statement.

Figure 2 (cited from [21]) shows the results of compared LOAD INFILE with BATCH INSERT.

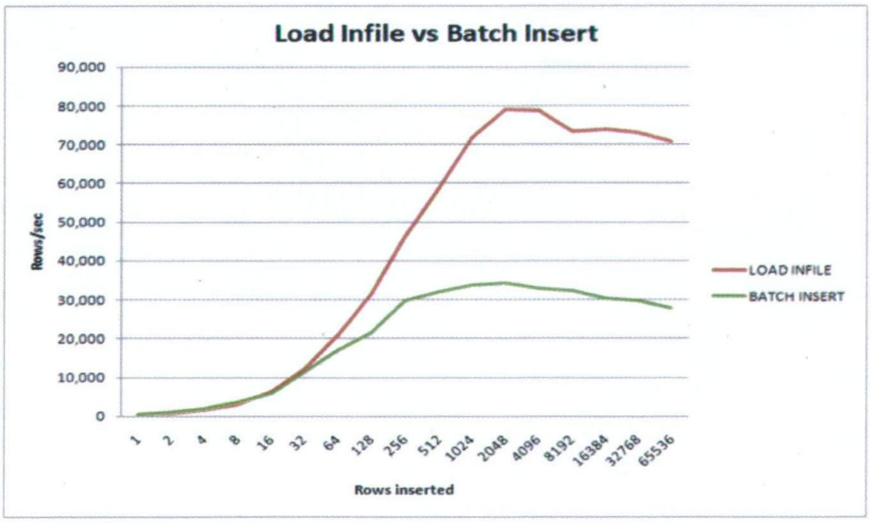


Figure 2 LOAD INFILE with BATCH INSERT

Therefore, on MySQL, the load data code is as follows:

```
    else {
        is = new FileInputStream(file);
        }
    st.setLocalInfileInputStream(is);
    st.execute
        ("LOAD DATA LOCAL INFILE 'file.txt' INTO TABLE RDFSETV
        (sub, pre, obj);");
```

Also, PostgreSQL has its own bulk insert method: copy data. The Java code on PostgreSQL is:

```
File temp = new File("d:\\temp.txt ");
    os = new FileOutputStream(temp);
    byte[] b = new byte[IO_BUFFER_SIZE];
    int read;
    while ((read = is.read(b)) != -1) {
    os.write(b, 0, read);
    }
    st.execute("COPY RDFSETV FROM 'd:\\temp.txt'");
```

2) Bulk insert: after loading the data into RDFSETV, the next step is to merge the tuples into RDFSETA. For the tuples which exist in RDFSETV but not in RDFSETA, they will be inserted into RDFSETA. The meaning of exist here is the same as defined before, a tuple in the current database version that matches the triple on subject, predicate, object and that has an end-value of -1. Similar with single operations, before performing the insert operation, a new version needs to be created and retrieved. The bulk insert method can be implemented using different SQL statements (as shown below). The differences and efficiencies among them are introduced in Section 5. The tuples which will be bulk inserted can be illustrated by red part in Figure 3. The bulk insert operations are "NOT IN", "NOT EXISTS" and "LEFT JOIN" on MySQL, and it has an addition option "EXCEPT" on PostgreSQL compared to MySQL.

"NOT IN" SQL statement is:

```
INSERT INTO RDFSETA
    SELECT sub, pre, obj, version_id, -1 FROM RDFSETV
    WHERE
        (RDFSETV.sub, RDFSETV.pre, RDFSETV.obj)
        NOT IN
            (SELECT RDFSETA.sub, RDFSETA.pre, RDFSETA.obj
                FROM RDFSETA
                    WHERE RDFSETA .endversion = -1);
```

"NOT EXISTS" SQL statement is:

```
INSERT INTO RDFSETA (sub,pre,obj, start, endversion)
    SELECT sub, pre, obj, version_id,-1 from RDFSETV
        WHERE NOT EXISTS
            (SELECT * FROM RDFSETA
                        WHERE
                            RDFSETA.sub = RDFSETV.sub AND
                            RDFSETA.pre = RDFSETV.pre AND
                            RDFSETA.obj = RDFSETV.obj AND
                            RDFSETA.endversion = -1);
```

"LEFT JOIN" SQL statement is:

```
INSERT INTO RDFSETA
    SELECT RDFSETV.sub, RDFSETV.pre, RDFSETV.obj, version_id,-1
        FROM RDFSETV
            LEFT JOIN RDFSETA on
                ( RDFSETA.sub = RDFSETV.sub AND
                RDFSETA.pre = RDFSETV.pre AND
                RDFSETA.obj = RDFSETV.obj AND
                RDFSETA.end = -1)
                    WHERE RDFSETA.sub IS NULL;
```

"EXCEPT" (EXCEPT only be supported by PostgreSQL) SQL statement is:

```
                RDFSETA.obj = RDFSETV.obj AND
                RDFSETA.end = -1)
                    WHERE RDFSETA.sub IS NULL;
```

"EXCEPT" (EXCEPT only be supported by PostgreSQL) SQL statement is:

```
INSERT INTO RDFSETA (select sub, pre, obj, version_id,-1
    FROM
        (SELECT sub, pre, obj FROM RDFSETV
            EXCEPT
                (SELECT RDFSETA.sub, RDFSETA.pre,
            RDFSETA.obj
                    FROM RDFSETA
                    WHERE RDFSETA.endversion = -1)) as C);
```

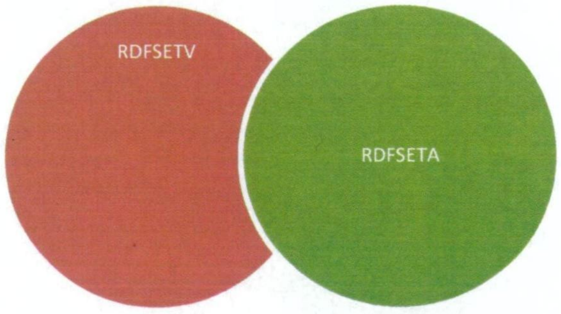
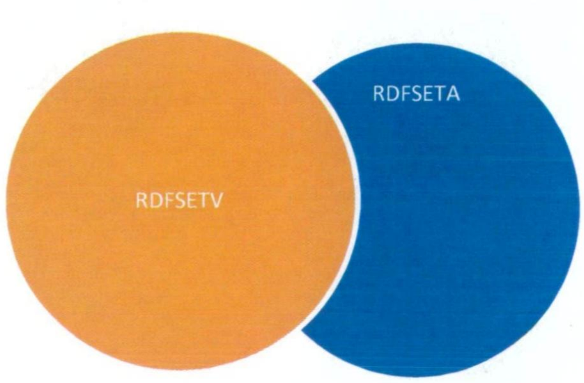


Figure 3. The tuples will be bulk inserted (red)

3) Bulk delete: the records are deleted from RDFSETA exist in RDFSETA but not in RDFSETV, that is to say. In Figure 4, the blue part represents the records that need to be deleted.



```
NOT EXISTS
    (SELECT * FROM RDFSETV
        WHERE
                RDFSETA.sub = RDFSETV.sub AND
                RDFSETA.pre = RDFSETV.pre AND
                RDFSETA.obj = RDFSETV.obj);
```

5 EXPERIMENT

5.1 EXPERIMENT GOAL

The Goal of the experiments is to test the solutions' storage space and efficiency of adding a new version of the database to the archive

5.2 EXPERIMENTAL SETUP

The experiments have been implemented with different SQL statements, different database configurations (on MySQL and PostgreSQL), and the different datasets (i.e. FACTBOOK and GO).

5.2.1 DATASETS

This paper uses FACTBOOK as the experiment data set. The FACTBOOK provides information on the history, people, government, economy, geography, communications, transportation, military, and transnational issues for 267 world entities. Their Reference tab includes: maps of the major world regions, as well as Flags of the World, a Physical Map of the World, a Political Map of the World, and a Standard Time Zones of the World map [19]. In the experiments, we use 100 version of the CIA Factbook in RDF format (starting from 1990).

The Gene Ontology project is a major bioinformatics initiative with the aim of standardizing the representation of gene and gene product attributes across species and databases. The project provides a controlled vocabulary of terms for describing gene product characteristics and gene product annotation data from GO Consortium members, as well as tools to access and process this data [14].In the experiments, we use 50 version of the GO in RDF format (starting from 2009)

Figure 6 and 7 show the number of delete tuples, insert tuples and all the tuples in archive after every RDFSETV version has been inserted into RDFSETA (5-column solution), from the figure, we can find out that there are increasingly bulk inserting and deleting operations in the first several versions and gradually less in the other versions for Factbook dataset. Since the two different datasets has similar performance, so this paper focuses on one of result from the dataset, e.g. World Factbook.

5.2.2 MYSQL

Figure 5 shows the storage space statistics of 4- and 5-column solution. It clearly shows that the one of 4-column solution's drawbacks is using huge storage space to keep data's versions. Therefore, this paper's experiments are related to 5-column solution.

On MySQL, Every type of SQL statement runs 3 times to get their average number with creating different index on archive (RDFSETA).

Author has developed NOT IN, NOT EXISTS, LEFT JOIN approaches on MySQL. With these different SQL statements, author tested the different indexes like SPOE (subject, predicate, object, endversion), SPO, OPS, PSO as well. In addition, it takes an unacceptable time if perform these SQL statements without index on MySQL.

Table 9 is the time of import triples into Database, since the 5-column solution running on MySQL always needs to create index, so the result is calculated by the average of all the experiments. We will compare the loading time on PostgreSQL (with index and without index) and on MySQL (with index).

Load data	Time

86226 (records)	9138 (ms)

Table 9 Load data on MySQL

The performances of 5-column solution with different SQL statements on MySQL are unexpected, different indexes with different SQL statements have similar time overhead. The reason for that is that when these three types of SQL statements are running, MySQL always uses index to optimize the queries. The results of time overhead with different SQL statements and indexes on MySQL are shown in Table 10.

From the Table, we can find out that bulk insert with index SPO (subject, predicate, object) has better performance than SPOE (subject, predicate, object, end). The reason of that is with index SPOE, its longer index increases the updating time. Also, it may conclude that bulk insert with index PSO is better than SPO and OPS on MySQL because less index updating time spent. Figure 8 and 9 show the insert time and merge time (insert time + delete time) for NOT IN, NOT EXISTS and LEFT JOIN with index PSO on MySQL. From the Figure 8 and 9, it can find out that LEFT JOIN has a little bit better performance than the others. In addition, respect to Figure 5, it can find that bulk insert operations' overhead increase quickly at first because there are more deleting and inserting operations at the beginning.

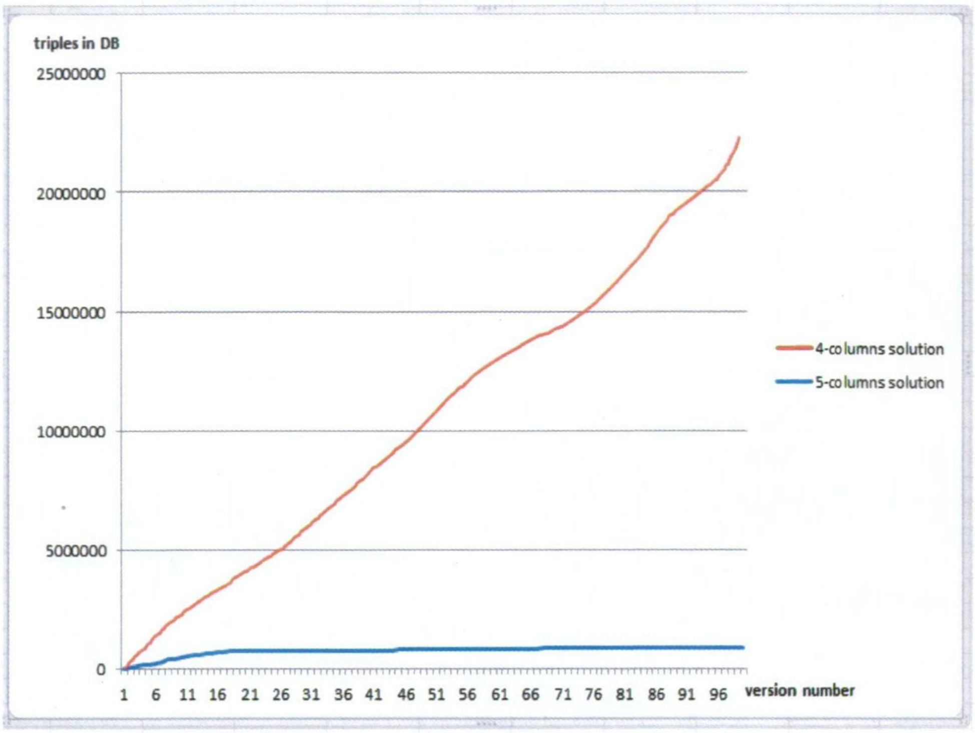


Figure 5. 4- and 5-column solution archive statistics on MySQL (FACTBOOK)

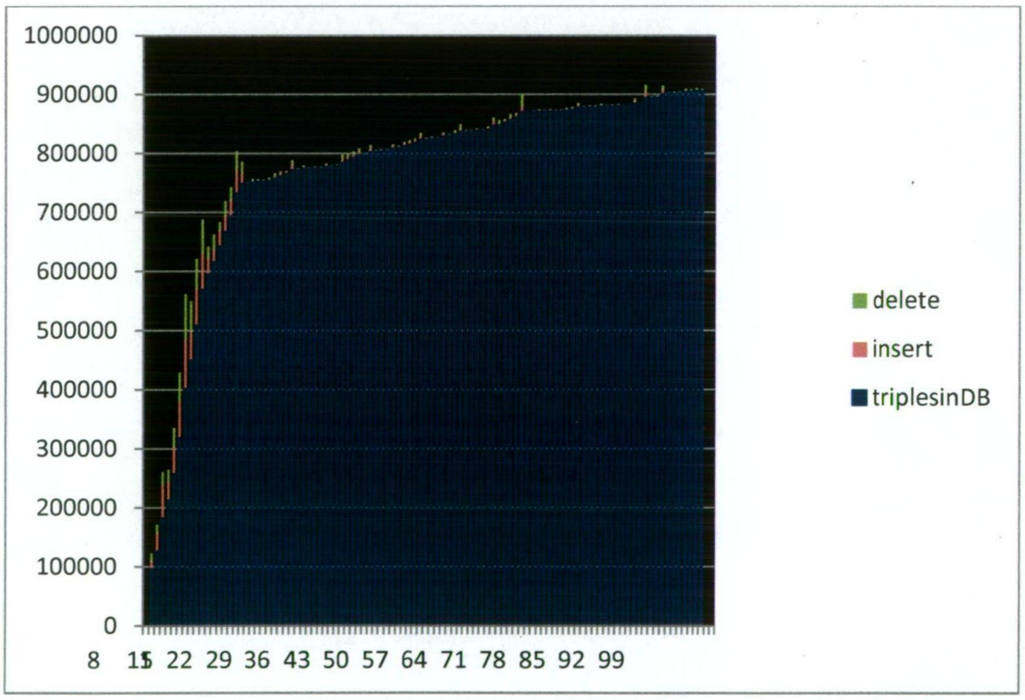


Figure 6 Dataset Statistics FACTBOOK

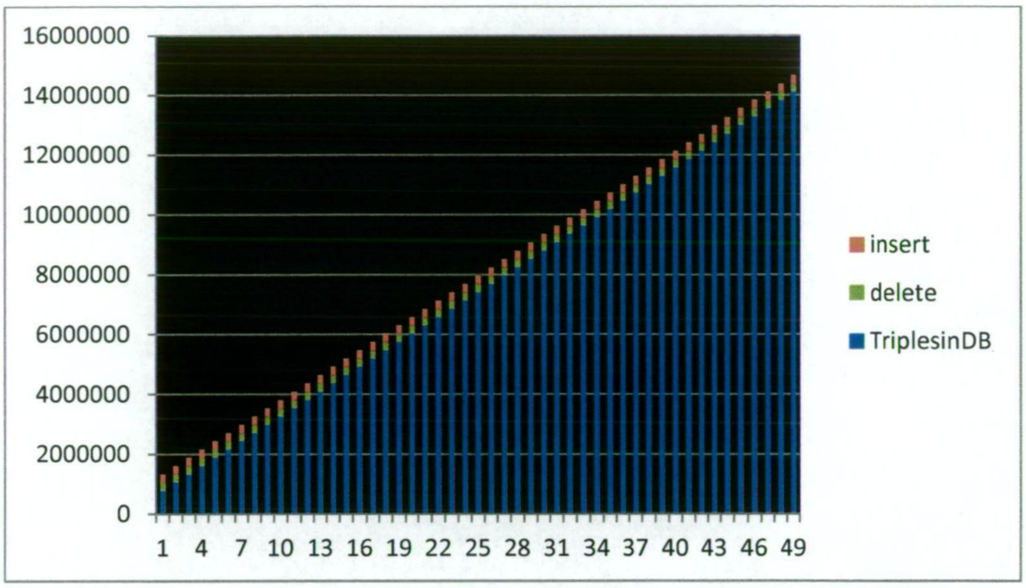


Figure 7 Dataset Statistics GO

(ms)	SPO	SPOE	OPS	PSO
NOT IN	4057558	4213184	4331715	4160777
NOT EXISTS	4020513	4239498	4380869	4097298
LEFT JOIN	4118943	4261755	4251236	3950536

Table 10. Total time on MySQL statistic

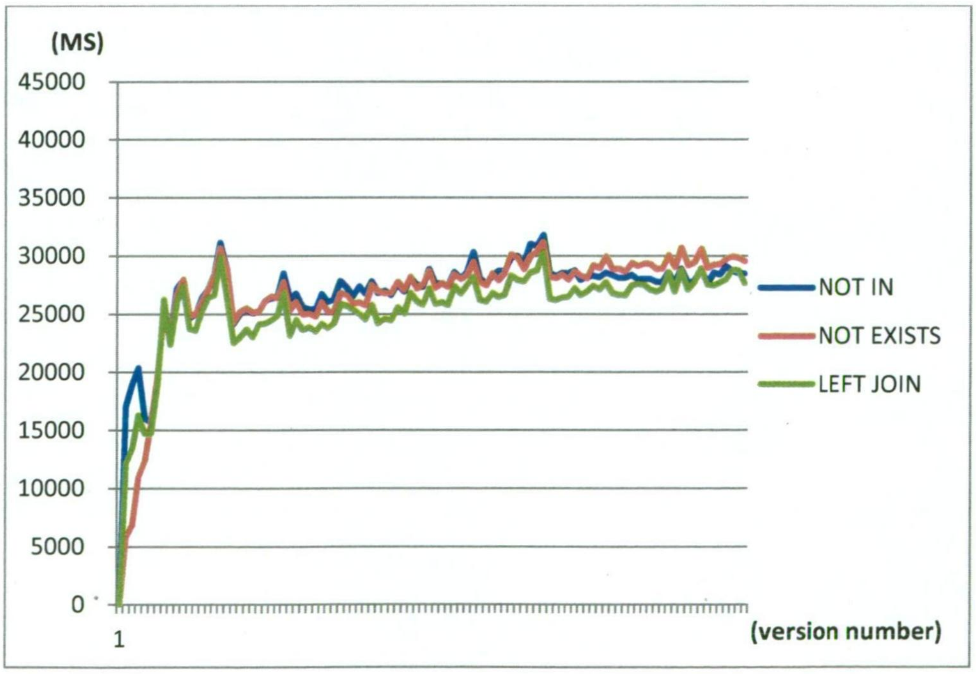


Figure 8. Merge time with index pso on MySQL

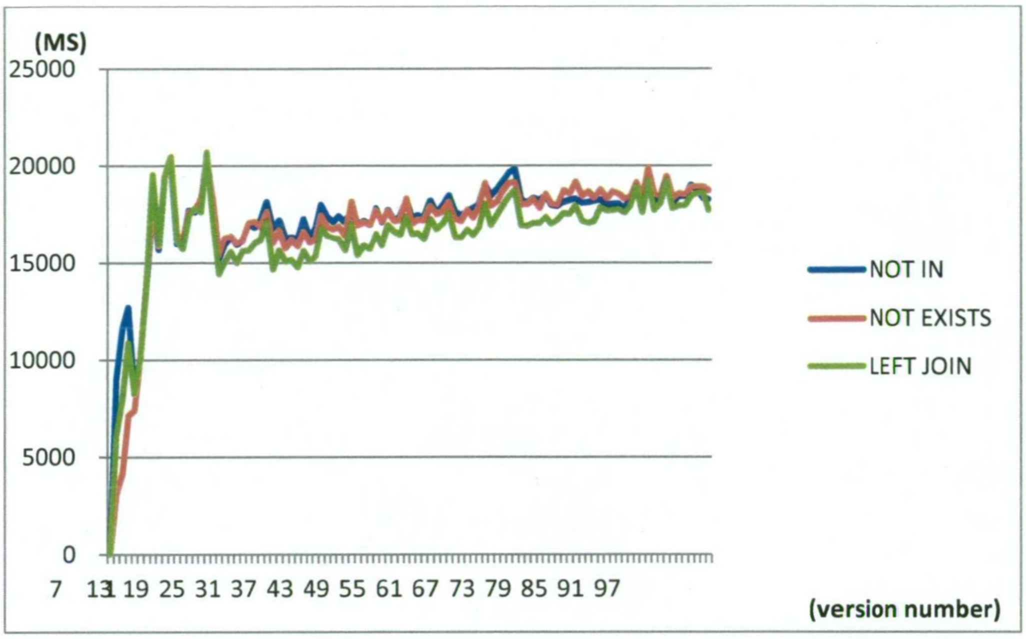


Figure 9. Insert time with index pso on MySQL

5.2.3 POSTGRESQL

On PostgreSQL, author implemented NOT EXISTS, EXCEPT, and LEFT JOIN for 5- columns solutions.

Since querying with index PSO has the best performance on MySQL, the experiment also uses this index on PostgreSQL to compare without index on PostgreSQL.

As the shown from Figure 10 and 11, EXCEPT is much better than LEFT JOIN and NOT EXISTS. In addition, no index on PostgreSQL manages to outperform with index for bulk inserting of 5-column solution. Respect to Figure 5, it also can find that at the beginning of bulk insert, the overhead increase rapidly since much inserting and deleting operations.

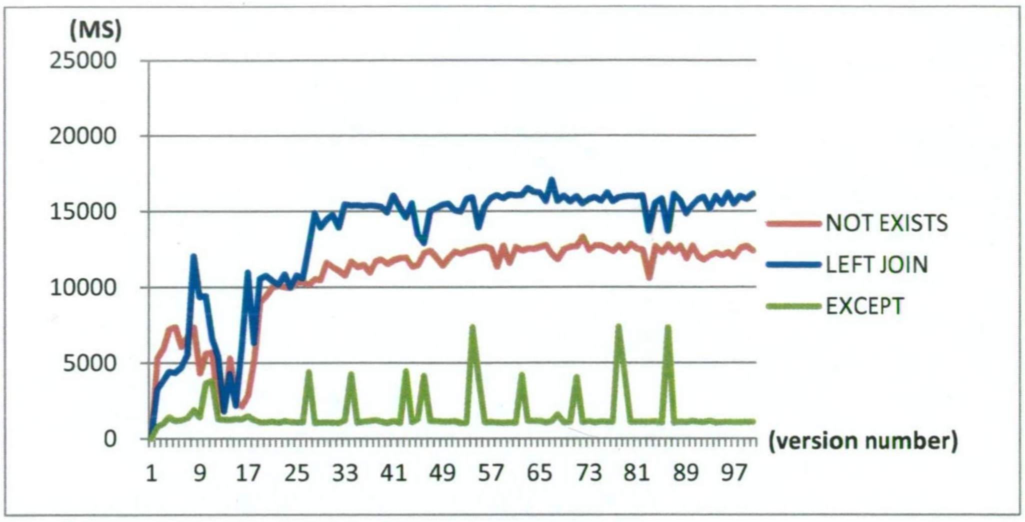


Figure 10.1 Insert time without index

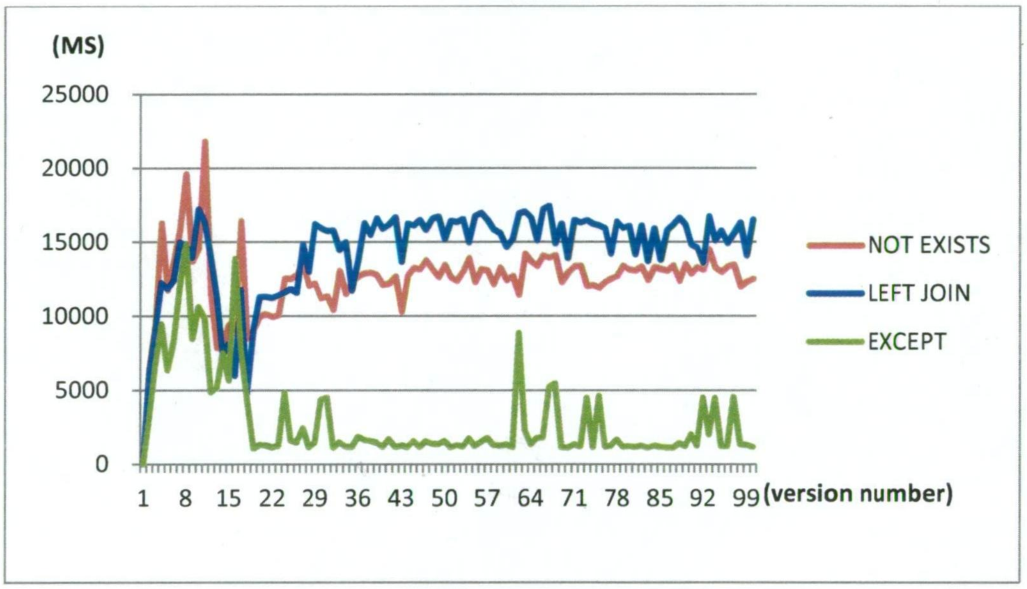


Figure 10.2. Insert time with index PSO

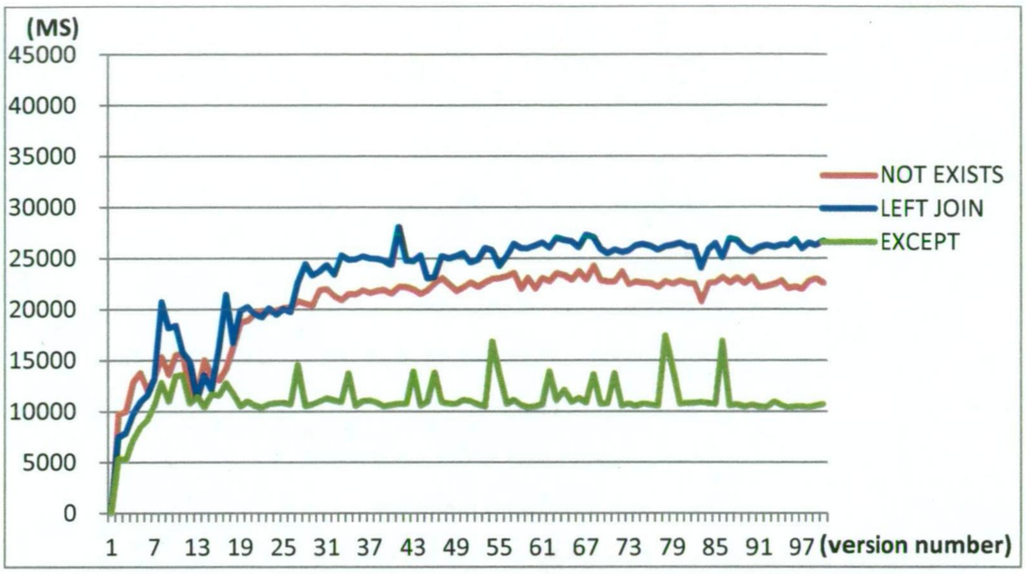


Figure 11.1 Merge time without index on PostgreSQL

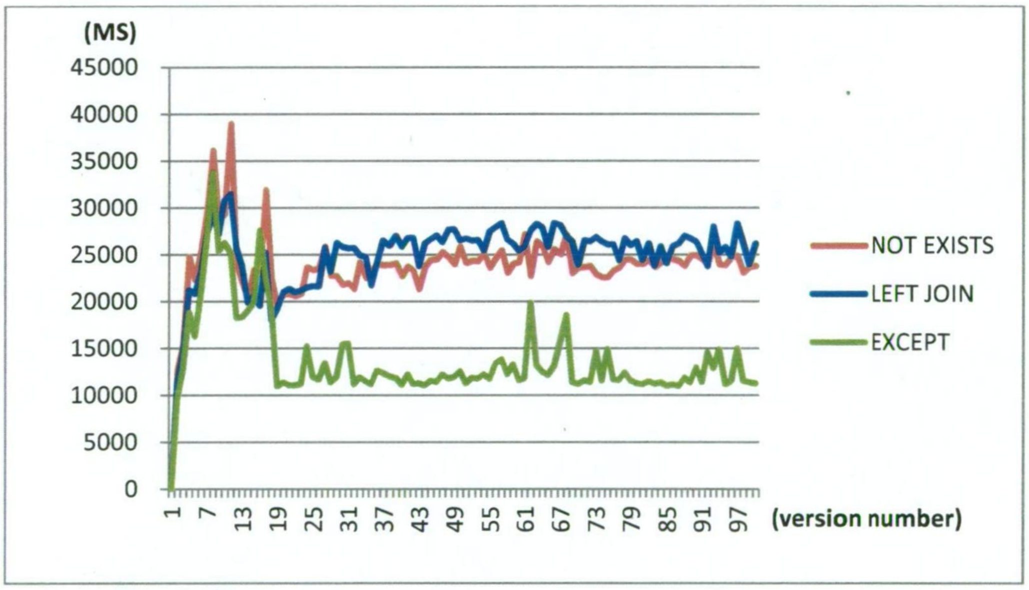


Figure 11.2 Merge time with index PSO on PostgreSQL

6 QUERY

In this section, the paper introduces three types of querying based on the 5-column solution. They are:

6.1 SNAPSHOT QUERY:

The query retrieves all the triples from one database version. In this query, we retrieve a database version from the archive, we need to retrieve all the triples that have an (start, end) interval which includes the version number we are looking for. There are two types of this querying, for example, the query which retrieves the history of a country and a query that retrieves the current population of the country. The SQL statements for retrieving the history of Australia in 2008 and the current population of Australia are:

The history of Australia in 2008:

```
SELCT    B.subject, B.predicate, B.object
          FROM RDFSETA as, A RDFSETA as B, VERSION_INFO
               WHERE
                    VERSION_INFO.availabletime = "2008"  AND
                    A.predicate = "hasName"              AND
                    A.object = "Australia"               AND
                    A.subject = B.subject                AND
                    B.start <= VERSION_INFO.id           AND
                    (B.end >= VERSION_INFO.id            OR
                    B.end = -1);
```

The current population of Australia:

```
SELECT F.object
     FROM  RDFDATASET as A, RDFDATASET as B, RDFDATASET as C,
           RDFDATASET as D, RDFDATASET as E, RDFDATASET as F
          WHERE
                    A.predicate = "hasName" AND
                    A.object = "Australia" AND
                    B.predicate = "hasCategory" AND
                    A.subject = B.subject AND
                    B.object = C.subject AND
                    C.predicate = "hasName" AND
                    C.object = "People" AND
                    D.subject = C.subject AND
                    D.predicate = "hasProperty" AND
                    D.object = E. subject AND
                    E.predicate = "hasName" AND
```

```
                E.object = "Population" AND
                E.subject = F.subject AND
                F.predicate = "hasValue" AND
                F.end = -1;
```

6.2 DIFF QUERY:

The query retrieves all the triples that are different between two versions of the archive. That is, all triples that are in version X but not in version Y will be retrieved by SQL query, so we need to retrieve all the triples that have one version but not the others.

The SQL statement is (say X<Y):

In X but not in Y:

```
SELCT  RDFSETA.sub, RDFSETA.pre, RDFSETA.obj
    FROM RDFSETA
        WHERE
            RDFSETA.start <= X AND
            RDFSETA.end <=Y-1
```

In Y but not in X:

```
SELECT RDFSETA.sub, RDFSETA.pre, RDFSETA.obj
    FROM RDFSETA
        WHERE
            RDFSETA.start > X AND
            (RDFSETA.end = -1 or RDFSETA.end >= Y)
```

6.3 TEMPORAL QUERIES

This type of query is able to query how a certain 'fact' or object changed over time, for instance, how did the population of Australia change? We need to find all the triples that have an (start, end) interval which locates the given range.

```
SELECT F.object
```

```
FROM  RDFDATASET as A, RDFDATASET as B, RDFDATASET as C,
      RDFDATASET as D, RDFDATASET as E, RDFDATASET as F
  WHERE
          A.predicate = "hasName" AND
          A.object = "Australia" AND
          B.predicate = "hasCategory" AND
          A.subject = B.subject AND
          B.object = C.subject AND
          C.predicate = "hasName" AND
          C.object = "People" AND
          D.subject = C.subject AND
          D.predicate = "hasProperty" AND
          D.object = E. subject AND
          E.predicate = "hasName" AND
          E.object = "Population" AND
          E.subject = F.subject AND
          F.predicate = "hasValue" AND
          (F.start <= A.start AND
          (F.end = -1 OR F.end >= A.start) OR
          A.start <= F.start AND
          (A.end = -1 OR A.end >= F.start)) AND
          (F.start <= B.start AND
          (F.end = -1 OR F.end >= B.start) OR
          B.start <= F.start AND
          (B.end = -1 OR B.end >= F.start)) AND
          (F.start <= C.start AND
          (F.end = -1 OR F.end >= C.start) OR
          C.start <= F.start AND
          (C.end = -1 OR C.end >= F.start)) AND
          (F.start <= D.start AND
          (F.end = -1 OR F.end >= D.start) OR
          D.start <= F.start AND
          (D.end = -1 OR D.end >= F.start)) AND
          (F.start <= E.start AND
(F.end = -1 OR F.end >= E.start) OR
E.start <= F.start AND
(E.end = -1 OR E.end >= F.start));
```

With start and end column, it is easy to query all the triples from one database version (snapshot) and retrieve all the triples that are different between 2 versions (diff). But it is complicated for querying how a certain fact has changed (temporal) because it needs to identify different situation of the fact. In addition, from the three types of queries, it can find out that the more triple patterns there are in the SPARQL query the more self-joins are necessary in the corresponding SQL query, it is also a drawback for 3-column approach.

7 COLUSIONS

In this paper, compared with existing archiving RDF data approaches, two archiving RDF data in relational database solutions which are able to track the history of changes to the data have been described. The experiments for the 5-column solution have demonstrated that the "EXCEPT" on PostgreSQL has the best performance. Also, with 5-column solution it can be conclude that PostgreSQL is more suitable for archiving RDF data than MySQL is. In addition, based on 5-column solution, three different types of SQL queries have been developed. The queries can be used for to retrieve the triples from a certain version, to find the triples that have one version but not the others, and to track how a fact has changed. An issue exists in queries is that much self-join have been caused when perform the temporal queries. It is also the problem in triples store approach. A natural question is whether it can be improved if developing the solution which based on other existing approaches like property table approach and vertically approach. Another issue is to develop archiving RDF data on RDFDB, and to compare them with approaches on relational database.

8 REFERENCES

[1]. Abadi, D.J, Marcus, A, Madden, S & Hollenbach, K.J 2007, 'Scalable semantic web data management using vertical partitioning', VLDB'07, pp. 411–422

[2]. Auer, S & Herre, H 2005 'A versioning and evolution framework for RDF knowledge bases', viewed 10th May 2011,< http://www.informatik.uni-leipzig.de/~auer/>.

[3]. Beckmann, J.L, Halverson, A, Krishnamurthy, R & Naughton. J.F 2006, 'Extending RDBMSs To Support Sparse Datasets Using An Interpreted Attribute Storage Format', In Proceedings of the 22nd International Conference on Data Engineering (ICDE), page 58.

[4]. Berners-Lee, T, Handler, J & Lassila, O 2001, 'The Semantic Web', Scientific American,vol. 184,2001,pp.34-43.

[5]. Brickley, D & Guha, R.V 2004, 'RDF Vocabulary Description Language 1.0: RDF Schema', W3C Recommendation, viewed 15th SEP, http://www.w3.org/TR/rdf-schema/.

[6]. Broekstra, J & Kampman. A 2004, 'SeRQL: An RDF Query and Transformation Language', Submitted to the International Semantic Web Conference, ISWC 2004.

[7]. Buneman, P, Khanna, S, Tajima, K & Tan, W.C 2004, 'Archiving scientific data', ACM Transactions on Database Systems, March, vol. 29, no. 1, pp. 2-42.

[8]. Chaudhuri, S & Weikum, G 2000, 'Rethinking Database System Architecture: Towards a Self-Tuning RISC-Style Database System', 26th International Conference on Very Large Data Bases (VLDB), pages 1–10, 2000

[9]. Chu, E, Beckmann, J.L & Naughton, J.F 2007, 'The case for a wide-table approach to manage sparse relational data sets'. In Proceedings of the ACM SIGMOD International Conference on Management of Data, pages 821–832.

[10]. Duan, S, Kementsietsidis, A, Srinivas, K & Udrea, O 2011. 'Apples and oranges: a comparison of RDF benchmarks and real RDF datasets', Proceedings of the 2011 international conference on Management of data ACM New York, NY, USA.

[11]. Eugene, I.C, Souripriya, D, George E & Jagannathan, S 2005. 'An Efficient SQL-based RDF Querying Scheme'. In Proceedings of the 31st International Conference on Very Large Data Bases (VLDB), pages 1216–1227.

[12]. Guha, R 2001, 'RDFDB-An RDF Database', http://www.guha.com/rdfdb/.

[13]. http://dev.mysql.com/doc/refman/5.0/en/connector-j-reference-implementation -notes.html.

[14]. http://www.geneontology.org/

[15]. http://www.w3.org/2000/10/swap/doc/cwm.html

[16]. http://www.w3.org/TR/rdf-mt/

[17]. http://www.W3C.org /standards/semanticweb/

[18]. http://jena.sourceforge.net/ontology/

[19]. https://www.cia.gov/library/publications/the-world-factbook/

[20]. http://www.hpl.hp.com/semweb/doc/RDB/rdb-performance.html,2003.

[21]. http://jeffrick.com/2010/03/23/bulk-insert-into-a-mysql-database/

[22]. http://www.w3.org/2001/sw/DataAccess/

[23]. http://www.w3.org/TR/rdf-sparql-query/

[24]. Klyne, G & Carroll, J 2004, 'Resource Description Framework (RDF): Concepts and Abstract Syntax', W3C Recommendation, viewed 15th SEP, http://www.w3.org/TR/rdf-concepts/.

[25]. Levandoski, J & Mohamed, F.M 2009 'RDF Data-Centric Storage', In Proceedings of the IEEE International Conference on Web Services (ICWS), 2009.

[26]. Matono, A, Amagasa, T, Yoshikawa, M & Uemura, S 2005, 'A Path-based Relational RDF Database'. In Proceedings of the 16th Australasian Database Conference (ADC), pages 95–103.

[27]. McBride, B 2002, 'Jena: A Semantic Web Toolkit', IEEE Internet Computing, 6(6):55–59.

[28]. Neumann, T & Weikum, G 2009 'the RDF-3X engine for scalable management of RDF data', VLDB, vol. 19, pp. 91-113.

[29]. RDF Primer. W3C Recommendation, viewed 15th SEP, <http://www.w3.org/TR/rdf-primer, 2004>.

[30]. Reggiori, A 2004, 'RDFStore Perl/C RDF storage and API', http://rdfstore.sourceforge.net/.

[31]. Sakr, S & Al-Naymat, G 2009 'Relational processing of RDF queries: a survey', SIGMOD, December, vol.38, no.4.

[32]. Seaborne, A 2004, 'Rdql-a query language for RDF', viewed 15th SEP, http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/.

[33]. Sidirourgos, L, Goncalves, R, Kersten, M, Nes, N & Manegold, S 2008, 'Column-store support for RDF data management: not all swans are white', PVLDB 23-28 Aug.

[34]. Sintek, M & Decker, S 2001, 'TRIPLE-an RDF query, inference and transformation language', Deductive Databases and Knowledge Management (DDLP).

[35]. Theoharis, Y, Christophides, V & Karvounarakis, G 2005, 'Benchmarking database representations of RDF/S stores', ISWC 2005, LNCS 3729, Springer-Verlag, Berlin, pp. 685-701.

[36]. Volkel, M 2004, 'D2.3.3.V1 SemVersion—versioning RDF and ontologies', viewed 10th May, <knowledgeweb.semanticweb.org/semanticportal/deliverables/D2.3.3v1.pdf>.

[37]. Weiss, C, Karras, P & Bernstein, A 2008, 'Hexastore: sextuple indexing for semantic web data management', Proceedings of the VLDB Endownment (PVLDB), 1(1):1008–1019.

[38]. World Wide Web Consortium (W3C), < http://www.w3.org/>.

[39]. Zhou, M & Wu, Y 2010 'XML-Based RDF data management for efficient query processing', WebDB 10.

APPENDIX:

All the code and all data used for experimental work have been submitted as electronic documents.