UPPSALA
UNIVERSITET

IT 12 011
Examensarbete 30 hp
Maj 2012

Performance of native SPARQL query processors

Shridevika Maharajan

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Performance of native SPARQL query processors

Shridevika Maharajan

Expressing data in RDF is one approach for making data available as Linked Data on the Web. Searching such data requires an RDF database engine providing some query language. The standard query language for RDF is called SPARQL. An RDF database engine can either be a middleware on top of an existing (relational) database or a native RDF store having its own internal data repository. Organizations often have difficulties to decide which solution they should adopt because there are few comprehensive comparisons of existing native RDF stores with respect to performance and scalability. The Berlin Benchmark provides a framework for comparing the performance different implementations of SPARQL engines in general. We have made a performance comparison between some existing RDF store solutions based on the Berlin benchmark and summarize their performance outcomes with respect to load and query time. The RDF stores OpenLink Virtuoso, and AllegroGraph are compared. Furthermore we also evaluate the performance of the general graph database Neo4j with the general SPARQL processor Squirrel on top. As a base-line for middleware solutions we also compare with Jena SDB, running of top of MySQL.

Handledare: Silvia Stefanova
Ämnesgranskare: Tore Risch
Examinator: Anders Berglund
IT 12 011
Tryckt av: Reprocentralen ITC

TABLE OF CONTENTS

Performance of native SPARQL query processors

1. Introduction

A benchmark can be considered as a useful tool to test the performance of a system and help to decide which system would be useful for a particular use case depending on the demands and the availability. Since Semantic Web technology is adopted for a wide range of applications, there is a growing need for benchmarking of Semantic Web technologies and its query language SPARQL [1]. One of the well-known SPARQL benchmark is the Berlin benchmark [2]. This project investigates the performance of SPARQL features such as OPTIONAL, FILTERS, DESCRIBE, UNION operators etc. This project's main aim is to compare the performance of native RDF stores offering persistent storage with their own internal data stores with non-native RDF data stores using relational database back-ends. The comparison is done in terms of the data load time, indexing time, and SPARQL querying time.

The evaluated queries are the Berlin SPARQL benchmark queries. The performance of the following RDF stores has been measured: Jena SDB[3], AllegroGraph version 3.3 [4], Virtuoso version 6.2 [5] and Neo4j [6]. AllegroGraph and Neo4j are graph databases and store RDF triple data as graphs in their internal RDF data stores. The used Virtuoso configuration stores RDF triple data in Virtuoso's owns relational database. Jena SDB uses a commercial RDBMS for its storage, i.e. MySQL, SQL server, etc.

2. Background

The Semantic Web [7] is defined by Berners-Lee [8] as **"a web of data that can be processed directly and indirectly by machines."** It can be considered as a collection of information which is connected in a way that is easily understood by the machines. It can be thought of as the next level of World Wide Web. The semantic web is built on triple based structures called the Resource Description Framework. The triples constitutes of URI‟s (Uniform resource Identifier) and literals [9]. A URI [10] is the unique identifier for resources in web. Basically Semantic Web is built triple by triple, where a particular triple is linked to another triple which in turn is linked to some other triple and this continues as a chain. For instance the link between the triples continues to connect pages of different blogs then different sites and different countries thus forming a tight connection between the resources.

2.1 Resource Description Framework

The Semantic Web where information is exchanged between websites and applications needs a structured and unified manner to represent the information which is understood by the machines. This is supported by the Resource Description Framework [11] recommended by the World Wide Web Consortium, which defines the metadata representation enabling interoperability between the web applications.

Though the web is filled with abundant information available for access, the quality of the relevant information retrieved when a user enters a particular query to the search engine is still a doubt. Search results might contain both useful and irrelevant pages for the user, which demands further filtering by the users themselves (manually). If authors associate metadata with web resources it becomes easier for the information access and integration tools-search engines to give more precise and relevant results. Thus in order to have a generalized format for describing metadata avoiding problems of compatibility, efforts are taken to come up with a good and efficient meta-data framework. The most promising framework among these is RDF, which provides strong fundamentals for the description of the resources" metadata thus enabling interoperability between the websites and applications. These results in a directed labeled graph where the graph nodes represent the resources and the edges represent the relationship between them. RDF can also be used to represent resources which can be identified on the web but cannot be retrieved.

2.1.1 RDF statements/triples

The basic component of RDF is the RDF triple [12]. An RDF triple is used for defining objects and the relationship between them. A triple consists of three parts: *subject*, *predicate*, and *object*. All three together create a *sentence* called an *RDF triple*; the triple is an encoding of a sentence that is comprehended by both computers and humans.

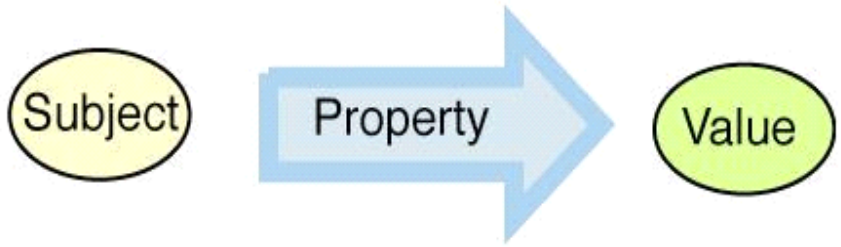


Figure 1 : Triple representation

For example, consider a simple sentence:"David likes sandwiches". This sentence represented as an RDF triple will have "David" as the subject, "likes" as the triple predicate and "sandwiches" as the triple object .

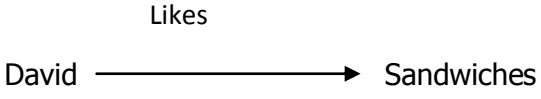


Figure 2: Triple representation

The idea of triples relies on how resources are related together and with literal data. For instance, a research paper can use triples to express the bibliographic information associated, while the experimental results and other information can also be linked together using triples. Linking data in this way can help the computers to retrieve relevant data from all over the internet. The triples are built using URIs which are unique to a particular concept. URI is a type of URL [13]. Thus the Semantic Web is built triple by triple, where one triple is linked to another one, which is linked to some other and so on. In this manner the links go on between web-pages, blogs, web-sites, linking countries and so making data to be connected tightly.

2.1.2 RDF Serialization

The RDF statements can be expressed in a variety of formats. RDF/XML [14] is an XML based format. Notation3 [15], Turtles [16] and N-Triples [17] are examples of the non-XML based serialization formats. It is usually not so important which format is used since most of the RDF parser tools [18] can parse all these formats. The non-XML based formats are much easier to write and understand as these are represented in tabular form. Notation 3 is closely related to NT and Turtles format. The triples are often stored in a special database called a triple store.

2.2 RDF Stores

An RDF store or triple store is a database for storing and querying RDF data. Like in relational databases, RDF data is stored in a triple store and then is retrieved by querying using a query language. These stores are used for storage and retrieval of RDF data and metadata in the form of RDF triples. Some triple stores like AllegroGraph, Virtuoso etc [24] have a storage capacity of even billions of triples. Recently there have been major developments in storage and query processing techniques of RDF-stores. Some RDF stores are built from scratch, while others are using an existing RDBMS database in the backend. The RDF stores fall under three different types based on their architecture. There are in-memory stores [24], native stores [24] and non- memory and non native stores [24].

An in-memory RDF store stores the RDF data in the main memory. This helps in performing operations like caching and inference example, Jena TDB[25]. The native RDF stores have their own storage implementation that is they are built as database engines from the scratch. Examples of native triple stores are AllegroGraph[4], Virtuoso[5], Mulgara[26], Garlik JXT[27] etc. The non-memory and non-native RDF stores take help of a third party database for their storage. It has been seen recently that native RDF stores are gaining momentum and popularity [24]. The reason is that these stores exhibit effective data loading and efficient query optimization. Example Jena SDB that can be coupled with third-party databases like MySQL, Oracle, PostsgreSQL.

2.3 RDF Query Language - SPARQL

SPARQL [1] is the standard RDF query language. SPARQL is an acronym for Simple Protocol and RDF Query Language. SPARQL was standardized by the RDF data access working group of the World Wide Web consortium, and is considered as a key Semantic Web Technology. The SPARQL query language for RDF is designed to meet the use cases and requirements identified by the RDF Data Access Working Group. SPARQL can be used to express queries across diverse data sources, whether the data is stored natively as RDF or viewed as RDF via middleware.[1] SPARQL has the capability for querying the required and optional graph patterns from the RDF graph along with their conjunctions and disjunctions. The RDF query language, SPARQL can also be seen as a protocol for accessing RDF. SPARQL does not support inference in itself. SPARQL does not do anything more than taking the descriptions of what the application wants, in the form of a query and returns the result in a form of RDF graph.

Some of the features of the SPARQL language are as follows:

FILTERS – A SPARQL filter constraints the query results to only those, where the filter expression evaluates to TRUE

OPTIONAL - RDF data is a semi-structured data, so when a query is executed, it never fails even when the data does not exist. This is achieved by the OPTIONAL clause.

LIMIT – The LIMIT modifier puts a bound to the number of query results returned.

ORDER BY – The ORDER BY clause is used to order the query result in a particular sequence. The ordering can be done in ascending or descending order.

DISTINCT – This clause is used to eliminate the duplicates that are present in the query result.

REGEX – This operator invokes the match function to match text against a regular expression pattern.

UNION operator – It combines graph patterns.
DESCRIBE – It returns a RDF graph describing a particular RDF resource.

CONSTRUCT – It returns a RDF graph constructed by substituting variables in a set of triple templates.

3. BSBM- Berlin SPARQL Benchmark

The Berlin SPARQL Benchmark (BSBM)[29] is developed to compare the SPARQL query performance of SPARQL endpoints [30]. The SPARQL endpoint include graph storage systems like AllegroGraph [4], Neo4j [6] etc native RDF stores like Virtuoso [5], Mulgara [26] etc and also systems mapping relational data to RDF. Many storage systems for RDF implement the SPARQL language and the SPARQL protocol within the enterprise and open web settings. As SPARQL is adopted by the community, there is a need for the comparing the performance of the RDF stores that expose the SPARQL endpoints via SPARQL protocol. The aforementioned idea led to the development of the Berlin SPARQL Benchmark.

The Berlin benchmark is built around e-commerce concept where a set vendors offer a range of products to the customers and the customers post their reviews about the products. The Berlin benchmark consists of the following parts:

1.The benchmark data generator generates the datasets used in the evaluation. The size of the generated dataset is scaled using a scale factor. It supports three different versions of datasets. The first version represents data using the RDF triple model; the second version represents data in a data model called the Named Graphs [44]. Finally the third data representation is in the form the relational data model. It has to be noted that the semantics of the generated data is independent of its representation

2. A standard set of 12 SPARQL benchmark queries for the evaluation.

3. There are three query mixes among the 12 basic SPARQL queries. These mixes define different common use cases that measure the performance of the RDF stores.

4. The performance metrics are presented along with the rules on how the benchmark is to be run and how the measurements are taken

5. The Data generator and the Test driver are available for use with a GNU license.

3.1 The Berlin data generator

The Berlin data generator [29] is a Java implementation requiring at least JVM 1.5. The source code of the data generator is downloadable and is licensed under the terms of GNU general public. The data generator is used to generate datasets of different sizes and the generation is deterministic. The output of the data generator supports formats like N-triples, turtles, XML, TriG and MySQL dump. There are several options available to make use in the configuration:

Option	Description
-s <output format>	For the dataset there are several output formats supported. Default: nt
-pc <number of products>	Scale factor: The dataset is scaled via the number of products. For Example: 91 products make about 50K triples. Default: 100
-fc	The data generator by default adds one rdf:type statement for the most specific type of a product to the dataset. However, this only works for system under test that support RDFs reasoning and can inference the remaining relations. If the SUT doesn't support RDFS reasoning, the option -fc can be used to include the statements for the more general classes also. Default: disabled
-dir	The output directory for all the data the test driver uses for its runs. Default: "td_data"
-fn	The file name for the generated dataset (suffix is added according to the output format). Default: "dataset"
-nof <number of files>	The number of output files. This option splits the generated dataset into several files.
-ud	Enables generation of update dataset for update transactions. The dataset file name is 'dataset_update.nt' and is in N-TRIPLES format with special comments to separate update transactions.
-tc <number of update transactions>	Specifies for how many update transactions, update data has to be written. Default: 1000
-ppt <nr of products per transaction>	This option specifies how many products with their corresponding data (offers, reviews) will be generated per update transaction. Default: 1

Note: the product count has to be at least as high as the product of the numbers defined with the -tc and -ppt options.

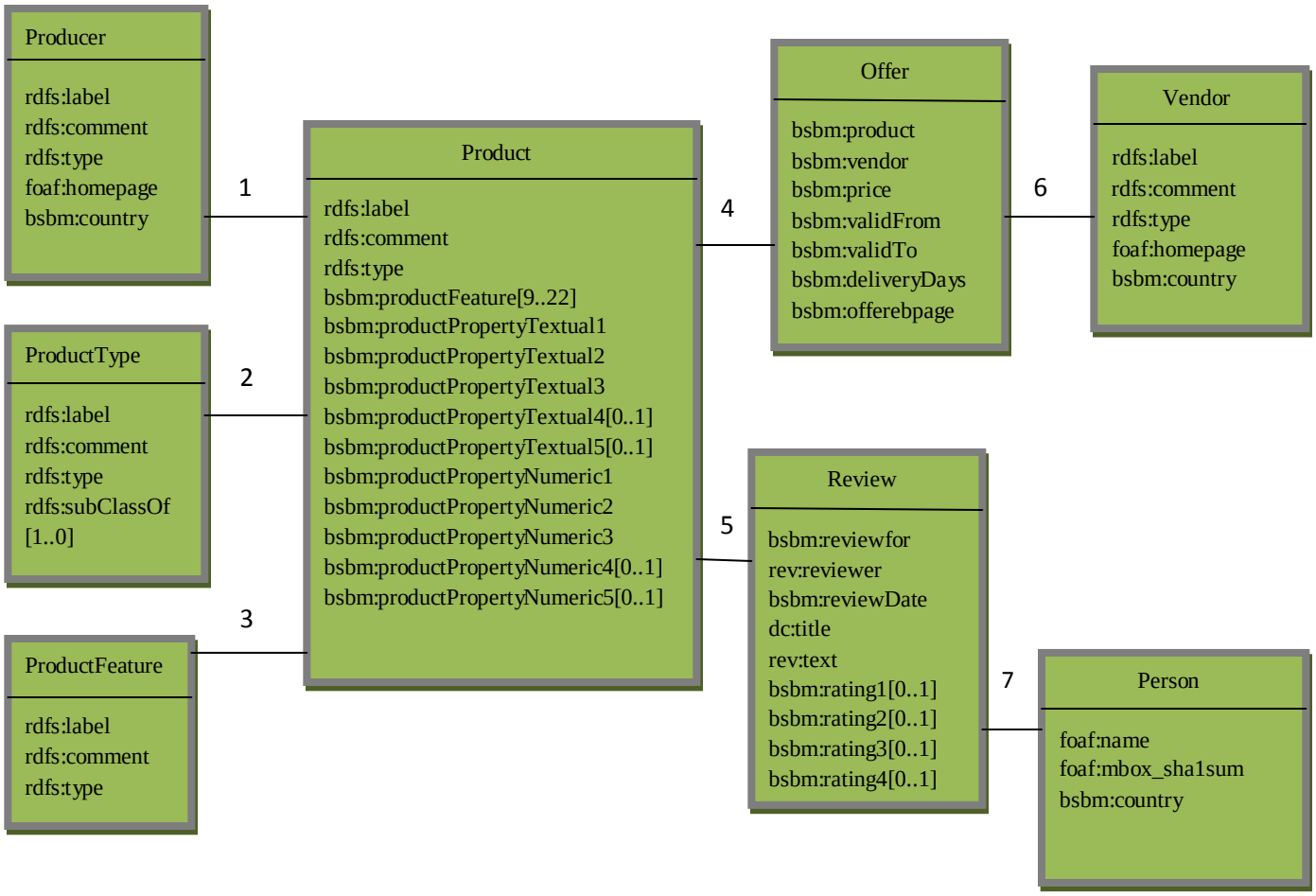
The following is an example of how to generate a dataset in turtle format. The number 1000 represents the scale factor

```
$ java -cp lib/* benchmark.generator.Generator -fc -pc 1000 -s ttl
The Benchmark Dataset
```

The scale factor shows the number of the generated products.

The BSBM data model has the following classes, namely *Product, Product Type, Vendor, Offer, Product Feature, Producer, Reviews and Person.*

Figure 3: Overview of the abstract data model.



The classes are related as follows:

2- rdf : type

3- bsbm : productFeature

4- bsbm : product

5- bsbm : reviewFor

The BSBM generator generates *n* product instances, where each product is defined by the properties *rdfs:lable* and *rdfs:comment*. A *product* (an instance of the class "Product") has textual properties between 3 and 5. The values of the properties are chosen from a dictionary in a random fashion. Each *product* instances consists of 5 to 15 words. A *product* can also have 3 to 5 numeric properties whose value ranges between 1 to 2000. Each product has a type that is part of the type hierarchy. The depth and width of the type depends on the scale factor.

Every product is considered to be produced by *producers*. The number of *products* produced by a *producer* assumes a normal distribution with mean = 50 and with a standard deviation of 16.6. *Producers* are continuously created until all the products are assigned to some producer. The *products* are actually being sold by *vendors*. *vendors* are characterized by the properties *label, comment, homepage URL and country URI*. Another class called *offer* is available. The number of *offers* available for a *vendor* follows a normal distribution with mean=2000 and standard deviation=667. New *vendors* are generated until all offers are assigned to some *vendor*.

4. BSBM Queries

There are two possible ways to design a query. The first method is designing a query based on certain features and thereby evaluating the how those features work. The second method of designing the query is based more on the real word use cases. The second method of designing obviously has more complicated combinations of the language features. For our experiments we use the BSBM queries[38] which are actually based on the second method where the queries are motivated on use cases, which eventually simulate a realistic work load on the systems under test.

The queries actually reflect the on-line search that is done by the consumer who is looking for products to purchase. such queries can be executed from a shopping portal on a real time where consumers are interested in the products, their sales, offers, delivery time needed etc. The following table gives the overview of the BSBM queries and the highlights of the SPARQL features that are used by the queries. Furthermore not all the features of the SPARQL language are exploited by the benchmark queries, for instance blank nodes, containers, property
Hierarchies, reified triples and the ASK query forms are not explored.

Characteristic	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Simple filters	√		√	√			√	√	√	√		
Complex filters					√	√						
More than 9 patterns		√		√			√	√				
Unbound predicates											√	
Negation			√									
OPTIONAL operator		√	√				√	√				
LIMIT modifier	√		√	√	√			√		√		
ORDER BY modifier	√		√	√	√			√		√		
DISTINCT modifier	√				√					√		
REGEX operator						√						
UNION operator				√							√	
DESCRIBE operator									√			
CONSTRUCT operator												√

Table 1: The above table is the SPARQL representation of the BSBM queries

Query 1: Find products for a given set of generic features

```
SELECT DISTINCT ?product ?label
WHERE {
?product rdfs:label ?label .
                      ?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature2% .
?product bsbm:productPropertyNumeric1 ?value1 . FILTER (?value1 > %x%) }
ORDER BY ?label
LIMIT 10
```

Query 2: Retrieve basic information about a specific product for display purposes

```
SELECT ?label ?comment ?producer ?productFeature
?propertyTextual1
?propertyTextual2 ?propertyTextual3 ?propertyNumeric1
?propertyNumeric2 ?propertyTextual4 ?propertyTextual5
?propertyNumeric4
WHERE {
%ProductXYZ% rdfs:label ?label .
%ProductXYZ% rdfs:comment ?comment .
%ProductXYZ% bsbm:producer ?p .
?p rdfs:label ?producer .
%ProductXYZ% dc:publisher ?p .
%ProductXYZ% bsbm:productFeature ?f .
?f rdfs:label ?productFeature .
%ProductXYZ% bsbm:productPropertyTextual1 ?propertyTextual1 .
%ProductXYZ% bsbm:productPropertyTextual2 ?propertyTextual2 .
%ProductXYZ% bsbm:productPropertyTextual3 ?propertyTextual3 .
%ProductXYZ% bsbm:productPropertyNumeric1 ?propertyNumeric1 .
%ProductXYZ% bsbm:productPropertyNumeric2 ?propertyNumeric2 .
OPTIONAL { %ProductXYZ% bsbm:productPropertyTextual4
?propertyTextual4 }
OPTIONAL { %ProductXYZ% bsbm:productPropertyTextual5
?propertyTextual5 }
OPTIONAL { %ProductXYZ% bsbm:productPropertyNumeric4
?propertyNumeric4 }}
```

Query 3: Find products having some specific features and not having one feature

```
 SELECT ?product ?label
 WHERE {
 ?product rdfs:label ?label .
?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productPropertyNumeric1 ?p1 .
FILTER ( ?p1 > %x% )
?product bsbm:productPropertyNumeric3 ?p3 .
FILTER (?p3 < %y% ) OPTIONAL {
?product bsbm:productFeature %ProductFeature2% .
?product rdfs:label ?testVar } FILTER (!bound(?testVar)) } ORDER BY ?label
LIMIT 10
```

Query 4: Find products matching two different sets of features

```
SELECT ?product ?label
WHERE {
{ ?product rdfs:label ?label .
?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature2% .
?product bsbm:productPropertyNumeric1 ?p1 . FILTER ( ?p1 > %x% )
} UNION {
?product rdfs:label ?label .
?product rdf:type %ProductType% .
?product bsbm:productFeature %ProductFeature1% .
?product bsbm:productFeature %ProductFeature3% .
?product bsbm:productPropertyNumeric2 ?p2 .
FILTER ( ?p2> %y% ) }} ORDER BY ?label
LIMIT 10 OFFSET 10
```

Query 5: Find products that are similar to a given product

```
SELECT DISTINCT ?product ?productLabel
WHERE {
?product rdfs:label ?productLabel . FILTER (%ProductXYZ% != ?product)
%ProductXYZ% bsbm:productFeature ?prodFeature .

?product bsbm:productFeature ?prodFeature .
%ProductXYZ% bsbm:productPropertyNumeric1 ?origProperty1 .
?product bsbm:productPropertyNumeric1 ?simProperty1 .
FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1
```

```
>
(?origProperty1 - 120))
%ProductXYZ% bsbm:productPropertyNumeric2 ?origProperty2 .
?product bsbm:productPropertyNumeric2 ?simProperty2 .
FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2
>
(?origProperty2 - 170)) }
ORDER BY ?productLabel
LIMIT 5
```

Query 6: Find products having a label that contains a specific string

```
SELECT ?product ?label
WHERE {
?product rdfs:label ?label .
?product rdf:type bsbm:Product . FILTER regex(?label, "%word1%")}
```

Query 7: Retrieve in-depth information about a product including offers and reviews

```
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review
?revTitle ?reviewer ?revName ?rating1 ?rating2
WHERE {
%ProductXYZ% rdfs:label ?productLabel .
OPTIONAL {
?offer bsbm:product %ProductXYZ% .
?offer bsbm:price ?price .
?offer bsbm:vendor ?vendor .
?vendor rdfs:label ?vendorTitle .
?vendor bsbm:country <http://downlode.org/rdf/iso-
3166/countries#DE>.
?offer dc:publisher ?vendor .
?offer bsbm:validTo ?date .
FILTER (?date > %currentDate% ) } OPTIONAL {
?review bsbm:reviewFor %ProductXYZ% .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?revName .

?review dc:title ?revTitle .
OPTIONAL { ?review bsbm:rating1 ?rating1 . } OPTIONAL { ?review bsbm:rating2
?rating2 . } } }
```

Query 8: Give me recent English language reviews for a specific product

```
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1
?rating2 ?rating3 ?rating4
WHERE {
?review bsbm:reviewFor %ProductXYZ% .
?review dc:title ?title .
?review rev:text ?text .
FILTER langMatches( lang(?text), "EN" )
?review bsbm:reviewDate ?reviewDate .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?reviewerName .
OPTIONAL { ?review bsbm:rating1 ?rating1 . }
OPTIONAL { ?review bsbm:rating2 ?rating2 . } OPTIONAL { ?review bsbm:rating3
?rating3 . } OPTIONAL { ?review bsbm:rating4 ?rating4 . } } ORDER BY
DESC(?reviewDate) LIMIT 20
```

Query 9: Get information about a reviewer.

```
DESCRIBE ?x
WHERE {
%ReviewXYZ% rev:reviewer ?x }
```

Query 10: Get cheap offers which fulfill the consumer's delivery requirements.

```
SELECT DISTINCT ?offer ?price
WHERE {
?offer bsbm:product %ProductXYZ% .
?offer bsbm:vendor ?vendor .
?offer dc:publisher ?vendor .
?vendor bsbm:country %CountryXYZ% .
?offer bsbm:deliveryDays ?deliveryDays .
FILTER (?deliveryDays <= 3)
?offer bsbm:price ?price .

?offer bsbm:validTo ?date .
FILTER (?date > %currentDate% ) } ORDER BY xsd:double(str(?price)) LIMIT 10
```

Query 11: Get all information about an offer.

```
SELECT ?property ?hasValue ?isValueOf
WHERE {
{ %OfferXYZ% ?property ?hasValue }
UNION
{ ?isValueOf ?property %OfferXYZ% } }
```

Query 12: Export information about an offer into another schema.

```
CONSTRUCT {
%OfferXYZ% bsbm-export:product ?productURI .
%OfferXYZ% bsbm-export:productlabel ?productlabel .
%OfferXYZ% bsbm-export:vendor ?vendorname .
%OfferXYZ% bsbm-export:vendorhomepage ?vendorhomepage .
%OfferXYZ% bsbm-export:offerURL ?offerURL .
%OfferXYZ% bsbm-export:price ?price .
%OfferXYZ% bsbm-export:deliveryDays ?deliveryDays .
%OfferXYZ% bsbm-export:validuntil ?validTo }
WHERE {
%OfferXYZ% bsbm:product ?productURI .
?productURI rdfs:label ?productlabel .
%OfferXYZ% bsbm:vendor ?vendorURI .
?vendorURI rdfs:label ?vendorname .
?vendorURI foaf:homepage ?vendorhomepage .
%OfferXYZ% bsbm:offerWebpage ?offerURL .
%OfferXYZ% bsbm:price ?price .
%OfferXYZ% bsbm:deliveryDays ?deliveryDays .
%OfferXYZ% bsbm:validTo ?validTo }
```

The parameters in the queries are enclosed within the % symbol. When running the queries, the parameters are replaced with random values selected from a generated dataset. The formulated queries can be found in the appendix.

5. RDF Stores under test

5.1 Jena SDB

Jena [31] is a Java framework which is useful for developing Semantic Web applications. It is an open source work grown with HP Labs Semantic Web Program. It provides API for RDF and OWL. The serializations supported by Jena are RDF/XML, N3, N-triples and Turtles. The Jena framework includes a SPARQL query engine, which interprets SPARQL queries against RDF data present in a back-end RDF store. Furthermore, it is an API that facilitates different RDF stores to make use of the SPARQL querying capabilities. This liberates the stores from worrying about the query implementation and specification.

For our experiments, we have used Jena-SDB [3] which is a component of Jena. It supports RDF storage and querying through SPARQL. Jena uses a back-end SQL database for the persistent storage. A number of SQL databases are supported example, Oracle 10g, PostgresSQL, MySQL, Apache Derby etc. We have chosen MySQL for our experiments since it is open source and robust. Using Jena SDB with MySQL requires a JDBC[32] connection.

A *buffer pool* is a memory that is used for caching relational tables and indexing the pages that are read frequently from the disk. The size of the buffer pool is configurable in MySQL. The configuration variable improves the performance of MySQL by caching disk tables in main memory to improve performance. When an application accesses a row from the table for the first time, the database manager will place this in the buffer pool. The next time when the same request is made the manager looks into the buffer pool and retrieves the result quickly, thus improving the performance profoundly.

Another feature to be discussed about Jena SDB is the table layouts. Jena SDB supports more than one table layout[33] for its relational database. The layouts supported by SDB are layout2, layout2/hash and layout2/index. Layout2 uses a triple table for storing a ‘default graph’ and a *quads table* for storing a ‘named graph’. The columns in the triple and quad tables have integers which refer to a ‘nodes table’. The next form of layout is hash, where the integers represent 8 bytes hashes of the node. In the third type of layout called the index, the integers are 4 byte sequence ids into the node table. For our experiments Indexed based layouts were used.

Triples

S	P	O

Primary key: SPO Indexes: PO, OS

Quads

G	S	P	O

Where S- subject, P- predicate, O-object and G-graph

Quads table for storing the named graph

Nodes

Index-based layout, the table is:

Id	Hash	Lex	Lang	DataType	ValueType

Primary key: Id
Index: Hash

Hash-based layout, the table is:

Hash	Lex	Lang	Data type	Value type

Primary key: Hash

All character fields are unicode, supporting any character set, including mixed language use.

Layout2 stores triples as strings in an S-P-O table. It is not for general use at any scale and really exists to test the variable layout framework.

Layout2/Index uses integers for node id, allocated by auto increment on the node table.

Layout2/Hash uses hash for node ids.

5.1.1 Loading and Indexing into Jena SDB Loading in Jena SDB

RDF data can be loaded into Jena SDB RDF store using the SDB bulk loader [34]. By the SDB loader, data is streamed into the database instead of loading it in a single transaction. The file's extension determines the syntax of SDB bulk loading command. Loading RDF data stored in a file, *FILE* into a named graph has the following syntax:

```
sdbload SPEC --graph=URI FILE [FILE ...]
```

Example `sdbload --sdb=sdb.ttl dataset.nt`

Loads RDF data (RDF triples) form the N-triple file *dataset.nt* into an SQL database, configured by the file *sdb.ttl*. The file *sdb.ttl* is the store description file, which is used to describe what store is to be used. The bulk load through command line is apparently faster compared to the loading done through `model.read` or `model.add` operations. The shell

scripts are used for running the commands from the command line. There is a configuration file which consists of environmental variables, whose values are set by the user.

```
$ export SDBROOT="/path/to/sdb
$ export SDB_USER="YourDbUserName"
$ export SDB_PASSWORD="YourDbPassword"
$ export SDB_JDBC="/path/to/driver.jar"
```

The `model.read` operation will automatically bulk load the data for each call of the operation. Another operation called the `model.add` performs in any form or combination of forms like loading single statement, list of statements or another

Indexing in Jena SDB

The Jena SDB command --index is used for indexing the data before querying. For example, the command
`sdbconfig --sdb=sdb.ttl --index`

where `--index` creates indexes for queries, enabling faster retrieval of the query results.

5.2 AllegroGraph

AllegroGraph [35] is a modern and persistent graph database for storing and querying RDF data. It uses disk based storage and can store billions of triples. The types of serialization supported by AllegroGraph are N-triple[17], RDF/XML[14], N-Quads[47], Trix [46]. The bulk of AllegroGraph database consists of triples and each of the triples has five sections subject, predicate, object, graph and triple-id. All of these sections are of arbitrary size. To speed up the queries, AllegroGraph creates indices. The default set of indices are called **SPOGI**, **POSGI**, **OSPGI, GPOSI, GOSPI** and **I**. **S, P, O, G** are subject, predicate, object and graph respectively I stands for triple identifier. The order of the index indicates how the indexing is sorted or done. The graph indices, namely **GSPOI, GPOSI, GOSPI** are used when the triple store is divided into sub graphs. The **I** index is a special type of index which lists all the triples by an id number. This index is helpful for faster deletion where a triple store is deleted by Ids. AllegroGraph builds all the mentioned indexes in the background. It is also possible to customize the indices that are built. AllegroGraph provides a very powerful Java API for connecting and interacting with the triple store. Some of the most important Java methods used for this purpose are :

renew() – Creates a new triple

loadNtriples() – Bulk load the triples from a RDF/N triples file

indexAllTriples() - Index all triples in the triple store

addStatement() – Add triple by triple in the triple store

getStatement() – Retrieve all the triples from the store

removeStatement() – Delete the triples from the store

AllegroGraph also provides a GUI based browser called "GRUFF"[36]

AllegroGraph has two variables that can be used for tuning the memory. They are `ags.setDefaultExpectedResources()` and `ags.setChunkSize([37])`.The number of unique resources and literals present and their total size in the RDF input file(s) are the main factors of memory utilization. To optimize the number of triples in your system, set the number of expected resources when opening a triple store. This will immediately allocate the right amount of memory and your image size won't grow much beyond that. If you set the initial value too small, the string table will have to be rebuilt, possibly many times, and more memory will be used, this can be set using the first variable i.e `ags.setDefaultExpectedResources()`. The second variable determines the number of triples that will be indexed at a particular time, the default setting should be appropriate for a 1GB Windows box but might be too large for a 512 MB Windows box.

5.2.1 Loading and Indexing into AllegroGraph

We loaded RDF data into AllegroGraph"s native store using a Java program. No command line tools were used.The Java code itself contained the method for indexing, i.e `indexAllTriples()`.

5.3 Virtuoso

Virtuoso is a native triple store that comes with both open source and a commercial license. It allows loading and querying by SPARQL of RDF data through a command line tool called the interactive *sql* or the *isql*. Virtuoso provides support for web servers which help in querying and loading data over HTTP.

For our experiments we used Virtuoso version 6.2 together with the Jena bridge called *Virtuoso Jena provider* [41]. the *Virtuoso Jena RDF Data provider* is a Native Graph Model Storage Provider which is fully operational and allows Semantic Web application written using the Jena RDF Framework to query the Virtuoso RDF store directly. Jena API (Jena SDB) is more mature and supportive with well established libraries as it is one of the earliest RDF storages that were developed. Virtuoso can be tuned for better performance. While running large datasets 2/3 or 3/5 of RAM size should be used. The other parameters in the Virtusos INI file that can improve the performance are `NumberOfBuffers` and `MaxDirtyBuffers` (3/4 of `NumberOfBuffers`) Furthermore while running with a large database, setting MaxCheckpointRemap to 1/4th of the database size is recommended [38].

5.3.1 Loading and Indexing into Virtuoso

For the newer versions of Virtuoso (higher than 6.00.3126) the indexing of RDF data includes 2 full indices over RDF quads and 3 partial indices. The indexing scheme in Virtuoso consists of the following indices:[39]

- **PSOG** - primary key

- **POGS** - bitmap index for lookups on object value.
- **SP** - partial index for cases where only S is specified.
- **OP** - partial index for cases where only O is specified.
- **GS** - partial index for cases where only G is specified.

In our experiments with Virtuoso, we chose to load RDF data into the system from TTL files. The loading was done through the *isql* tool. The following command function was used for carrying out the bulk loading of large TTL files [40].

```
DB.DBA.TTLP_MT(in strg any,in base varchar,[in graph varchar],[i n flags
integer]);
```

strg – text of the resource
base – base IRI to resolve relative IRIs to absolute
graph – target graph IRI, parsed triples will appear in that graph.
flags – bitmask of parsing flags. permits some sorts of syntax errors in resource. Default is 0, meaning no permitted deviations from the spec.

file_to_string_output function is used for loading lengthy files. It is important that the data file is accessible to the Virtuoso server. For example,

```
DB.DBA.TTLP (file_to_string_output ('.\tmp\data.ttl'), '',
'http://my_graph', 0);
```

loads into Virtuoso the RDF data from the file *data.ttl* and puts it under the identifier *http://mygraph* .The file *data.ttl* is located in the *tmp* folder or in a subfolder. Note that this example folder is a subfolder of the Virtuoso Server working directory.

5.4 Neo4J

Neo4j [43] is an open source graph database, implemented in Java. It is embedded and disk based and stores all the data as graphs rather than tables. It can also be called the NoSQL graph database. A graph database is a database where the data is stored and manipulated as a graph, the most generic of all the data structures. It is capable of representing the data in an easily

Accessible way. A graph contains nodes and relationships among them. A node can be connected to a second node(s) by so called *properties*. The second node(s) on its turn is connected by properties to a third node(s) and so on. Thus, the graph grows to millions of nodes It can be considered as a richly interconnected structure. In Neo4J the Java packages Org.neo4j.rdf [43] and Org.neo4j [43] are used to create a graph database, import RDF data into it and query it using SPARQL.

5.4.1 Loading and Indexing into Neo4j

Tinker pop[42] is an open source project that provides an entire stack of technologies within the Graph Database space. At the core of this stack is the Blueprints framework. Blueprints can be considered as the JDBC of Graph Databases. By providing a collection of generic interfaces, it allows to develop graph-based applications, without introducing explicit dependencies on concrete Graph Database implementations. By exposing a Neo4J Graph Database[43] (containing RDF triples) through the *Sail* interface, which is part of the openrdf.org project, we can reuse an entire range of RDF utilities (e.g. parsers and query evaluators) that are part of the openrdf.org project. The Blueprints framework provides us with a similar ability: each Graph Database binding that implements the Tinkerpop *TransactionalGraph* and *IndexableGraph* interfaces can be exposed as a*GraphSail*, which is Tinkerpop‟s implementation of the *Sail* interface. Once you have your *Sail* available, storing and querying RDF is standardized [41].

6. Experiments Configurations

The evaluation here focuses on centralized triple stores with a single client making queries at a given time. For this reason we have both the BSDB dataset and the triple store all present in the same system. The queries are executed one after the other.

The test environment hardware is an HP Compaq 8100 Elite with Intel(R) Core™ i5 CPU running at 2.66 GHz. The installed capacity of RAM is 8.00 GB. All the tests were carried out using the Sun Java virtual machine 1.7.0-b147. The operating system running was Windows 7 Professional

6.1. Evaluation Metrics

6.1.1 Loading time

The loading time is the time it takes to load a triple file in to the triple store. The triple files are either in Turtle or .NT formats. The time is measured as seconds, minutes, or hours depending on the size of the dataset. Loading data into the system was done one at a time that is at a particular instance only one system was loaded with one dataset as we did not want to divide the processor resource, which in turn can affect the performance evaluation.

6.1.2 Indexing time

Indexing improves the speed of the triple retrieval operations. There are more than one way to index the triples, which affects the overall efficiency of the triple store. The disk space required to store the index is small compared to the space required to store the actual table containing the data. This is again measured in terms of seconds, minutes or hours depending on the size of the data.

6.1.3 Query response time

The query response time is the time it takes to execute a SPARQL query. The query response time is calculated by taking the mean response time of executing each query a number of times. The query response time is measured in two ways, one is called *cold* runs and the other is called *warm* runs. The cold runs are measured immediately after the server has been started, so there are no data in the cache memory of the DBMS. The warm runs are measured without a running server, so the response time here is faster as data is already stored in the cache and therefore retrieved faster. Averages of cold values (3 cold measurements were taken)are taken are plotted in one graph and similarly averages of warm values (measurements were taken until a constant value was reached)are plotted in another graph. Thus for every query there are two runs, warm and cold and there are two graphs one for warm values and another for cold values.

6.2 Systems Configuration

The following illustrates the total configuration of the system under test. For some systems the default settings were changed to improve the performance, which was suffering otherwise.

To make things easier we use the WAMP [45] package in which we use the MySQL and phpMyAdmin. WampServer 2.1 is used for our experiments. The changes made to the default configuration of MySQL are as follows:

key buffer = 5600M

To minimize the disk I/O, MySQL uses cache mechanism which keeps the most frequently used table blocks in the memory. For index blocks a buffer pool in main memory contains a number of frequently used disk blocks. The *key- buffer* size parameter increases the size of the cache. After making changes to the MySQL configuration the DBMS server is restarted to activate the change. It is recommended to set the *key buffer* to 25-50% of the RAM size and therefore we set.

innodb_buffer_pool_size = 5120M (This is 70% of the total RAM size)

In AllegroGraph 3.3 Free Server Edition the default setting was used. Since we are using only the free version of AllegroGraph, we were limited to 50 million triples. In a similar manner the default settings were used for both Virtuoso 6.2 and Neo4j. Even though for Virtuoso some changes could have been made to default configuration parameters like `MaxDirtybuffers`, `numberOfBuffers` and `MaxcheckPoint` the default settings by itself were found to be performing better than the other systems so no changes were made. The new AllegroGraph 4.3 version does not support Windows and was not evaluated. The same set of formulated queries is used across all the RDF stores and all the datasets, irrespective of their sizes.
.

6.3 SPARQL Queries

The benchmark queries contain parameters that are enclosed within the "%" symbol. During the test runs, the parameters are replaced by random values from the generated dataset. The formulated SPARQL queries used for this experiment can be seen in Appendix I.

7. Evaluation

The following table illustrates the loading and indexing time of different RDF stores. The first column identifies the RDF stores and the first row represents the number of triples that was loaded into the system.

Triples / System	40377	374911	1075626	1809874	24711725	49279230
Jena (loading and Indexing) Secs	15.76	223.2	532.2	1416.6	35243.99	100166.4
Allegro (loading and Indexing) Secs	1.608	15.505	45.135	77.627	1156.8	3198
Virtuoso (loading and Indexing) Secs	1.38	13.22	55.74	90.48	6408	4860
Neo4j (loading and Indexing) Secs	24.98	6.35	1347	2484	53496	175032

Table 2:Loading and Indexing

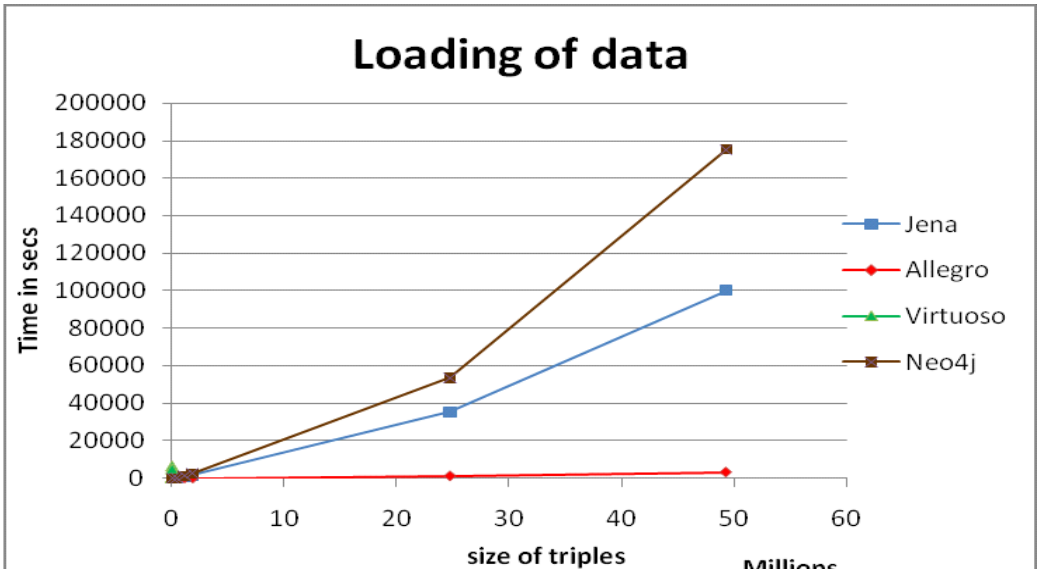


Figure 4

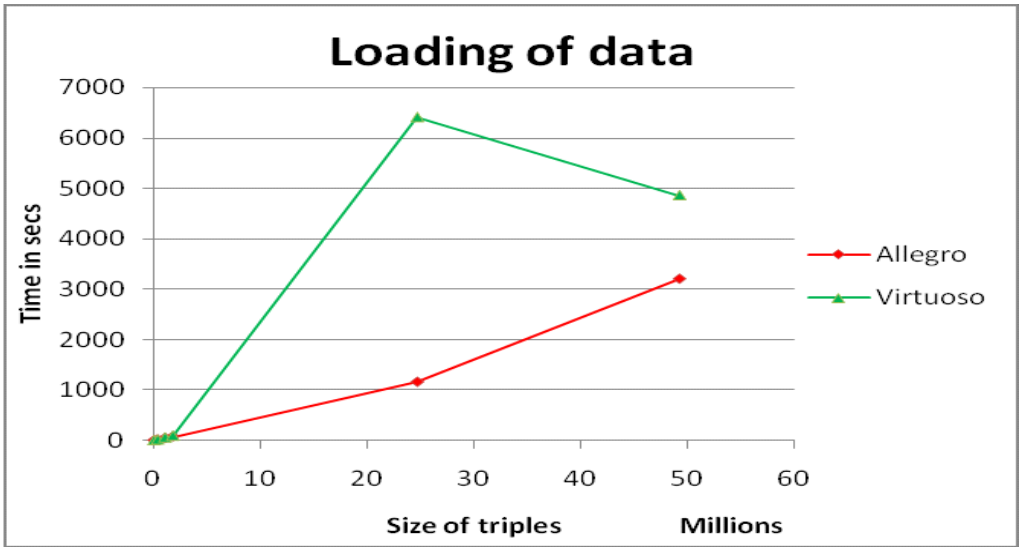


Figure 5

7.1 Loading and indexing performance

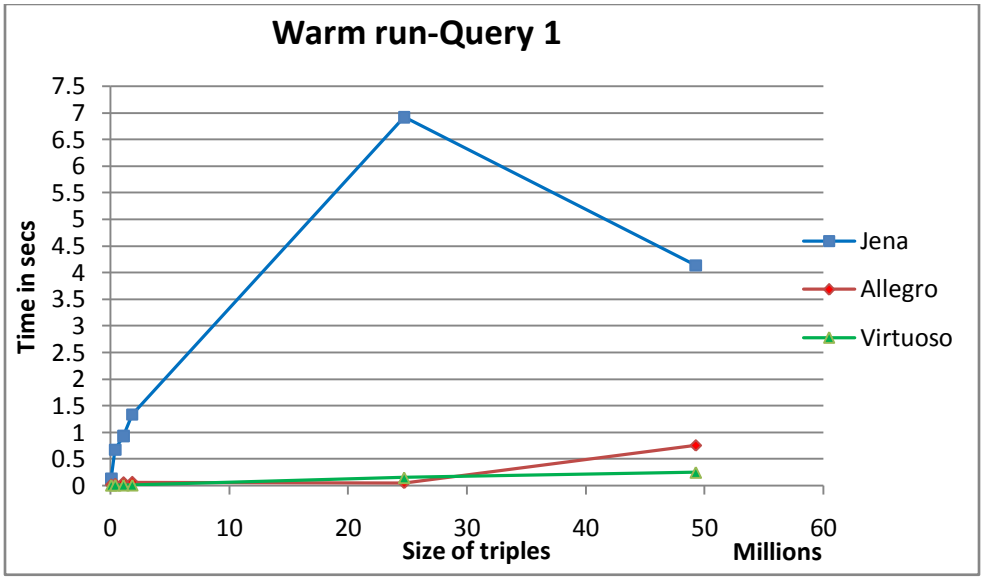
The above Table 2 , Figure 4 and Figure 5 illustrates the loading times for the different RDF stores under test.

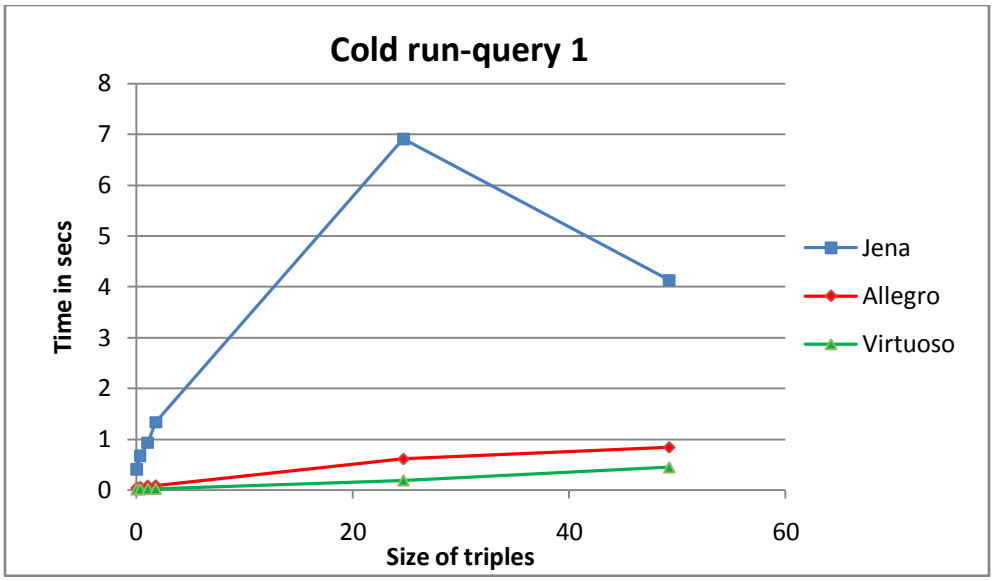
The time includes both loading and indexing time. Comparing the results from loading and indexing we can see that AllegroGraph and Virtuoso have similar best performance for smaller datasets. For larger datasets, AllegroGraph seems to be performing somewhat better than Virtuoso in our case where the maximum number of triples that were tested was 50 million triples. To confirm the above statement also larger number of triples needs to be loaded and a scalability advantage of Virtuoso ought to be observed. Jena SDB and Neo4j show poor

performance among the four triple stores, The comparison of Jena SDB and Neo4j shows though, that Neo4j has the longest loading times, furthermore Nei4j scales worst. It was found that Jena SDB showed a poorest performance if the MySQL was not tuned properly. For example before the MySQL was tuned properly, it took almost 8 days to load 25 million triples into Jena but we overcame this problem by tuning MySQL. The *key_buffer* (5600M) and *innodb_buffer_pool_size* (5120M) values were set appropriately for better performance. Though also Virtuoso and AllegroGraph could have been tuned for better results, this was not done as they did not show serious performance suffering while loading, in contrast to the other systems.

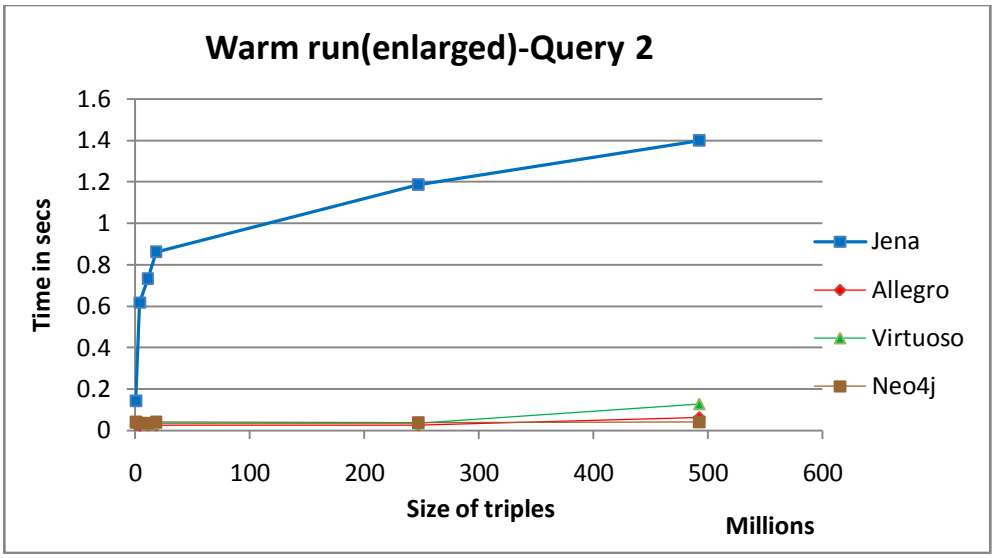
7.2 Query performance

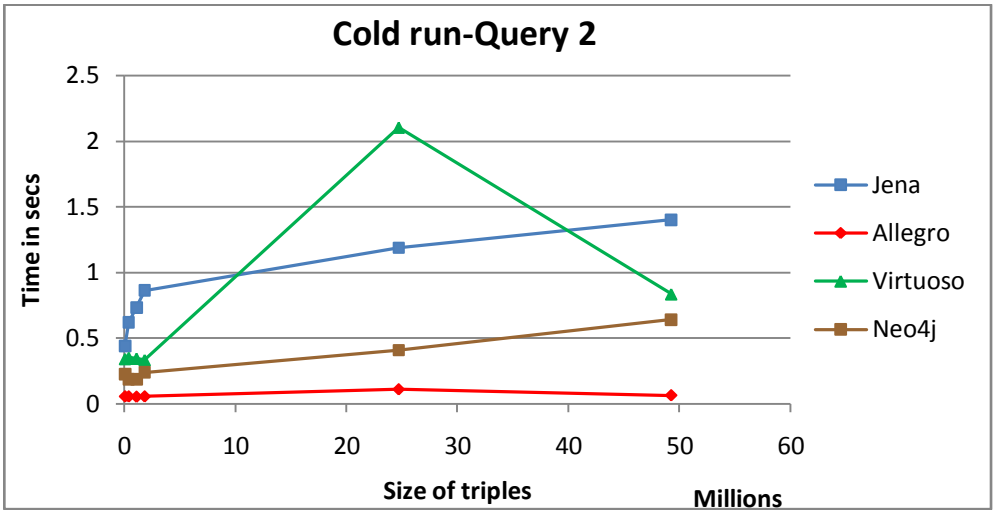
The following graphs illustrate the scalability of all the systems for the 12 queries in BSDB.



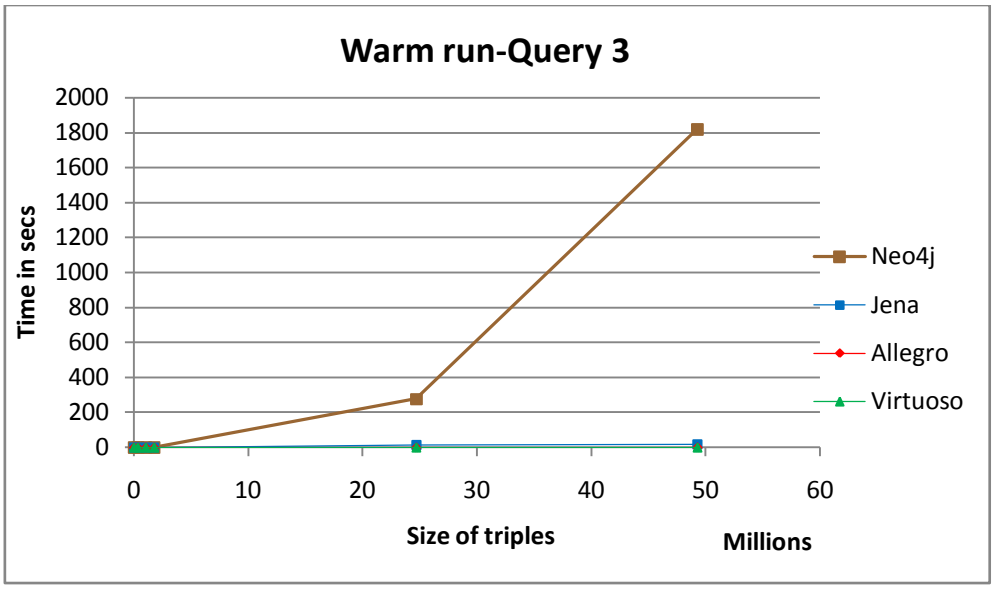


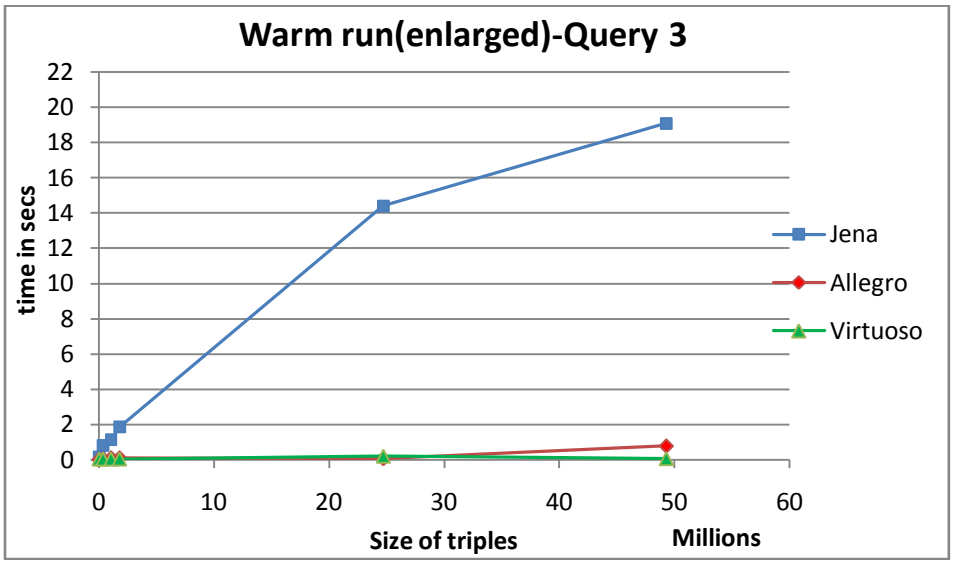
For query 1 Jena shows good scalability as there is a dip in the graph response time as the number of triples is increased. Virtuoso and AllegroGraph had the best performance with gradual and smooth. Ne04j could not process this query and hence no response time was obtained.

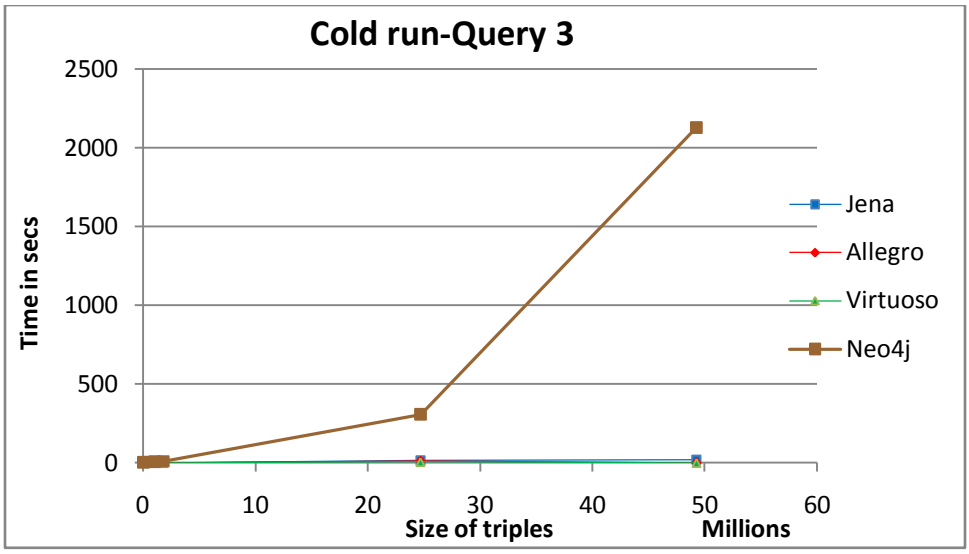


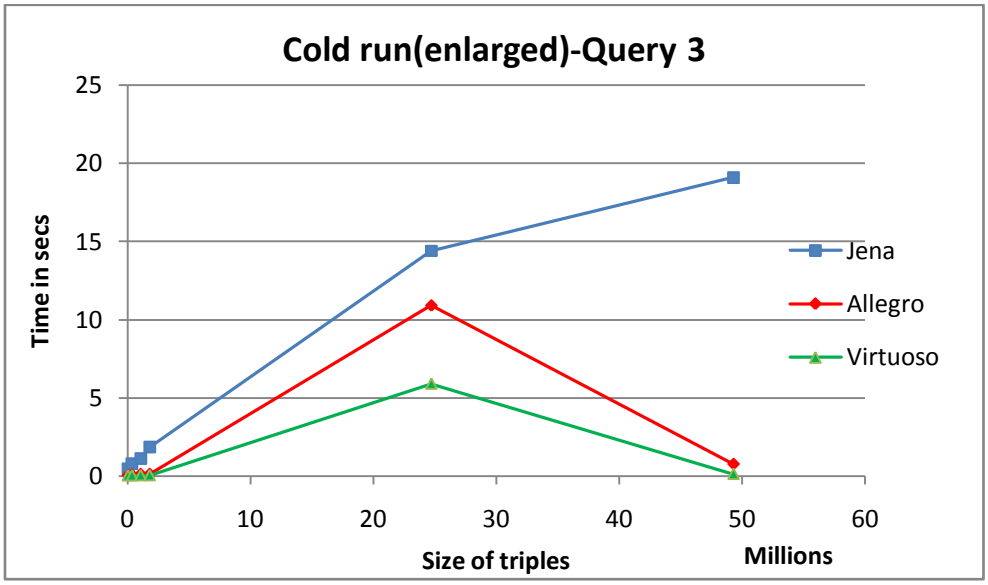


For query 2, it is clear that Virtuoso scales better than the other systems for larger databases. Allegro shows more or less a consistent performance or all the data sets irrespective of their size We could also infer that both Neo4j and Jena did not show good scalability since the response time kept increasing as the number of triples was increased. Jena was slow possibly because of the OPTIONAL operator where similar increases in the slope is seen with queries like 4, 7 and 8 that also contain OPTIONAL operator.

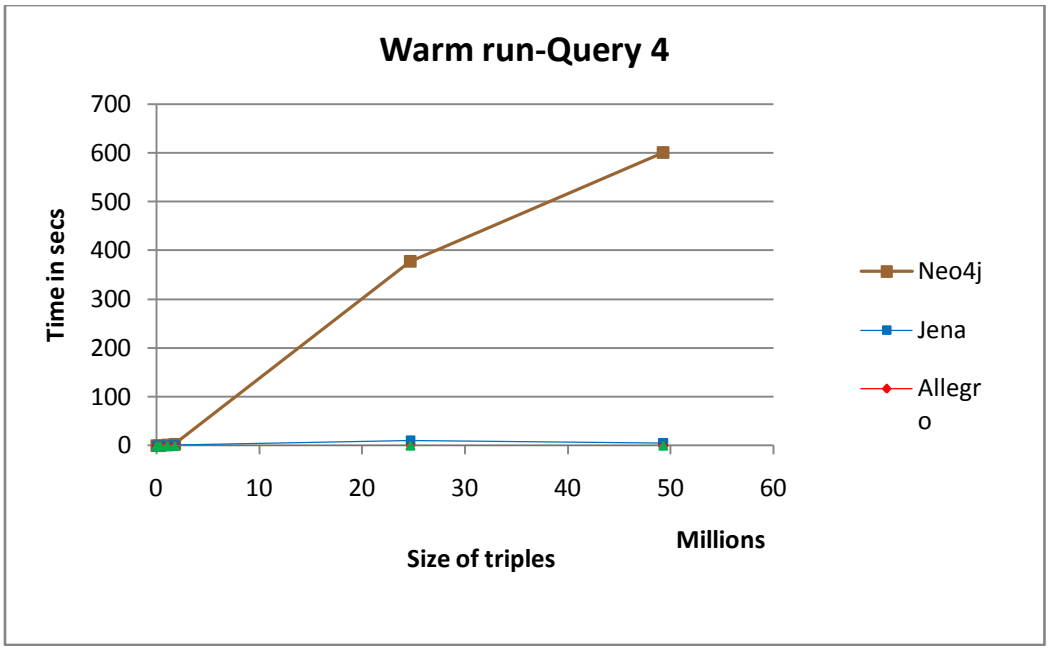

Warm run-Query 3
Time in secs
2000
1800
1600
1400
1200
1000
800
600
400
200
0
0
10
20
30
40
50
60
Size of triples
Millions
Neo4j
Jena
Allegro
Virtuoso

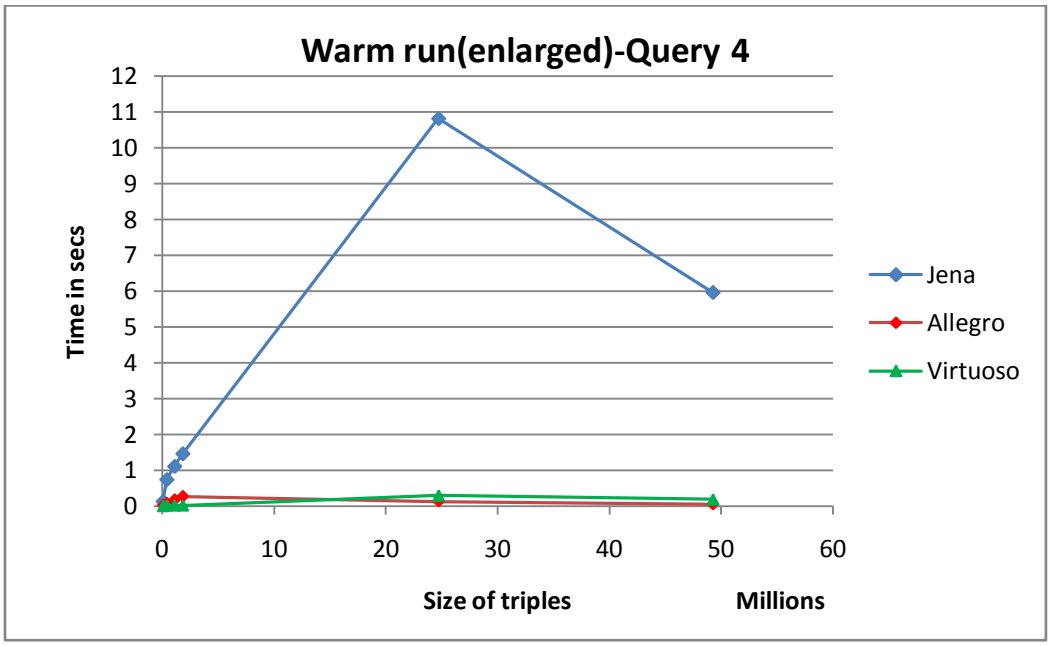

Warm run(enlarged)-Query 3
time in secs
22
20
18
16
14
12
10
8
6
4
2
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena
Allegro
Virtuoso

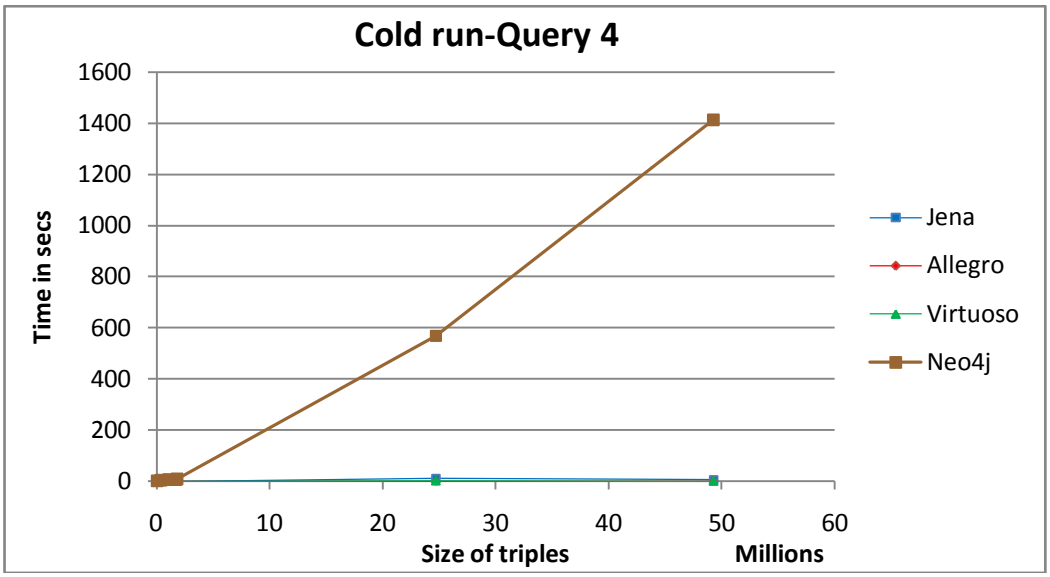


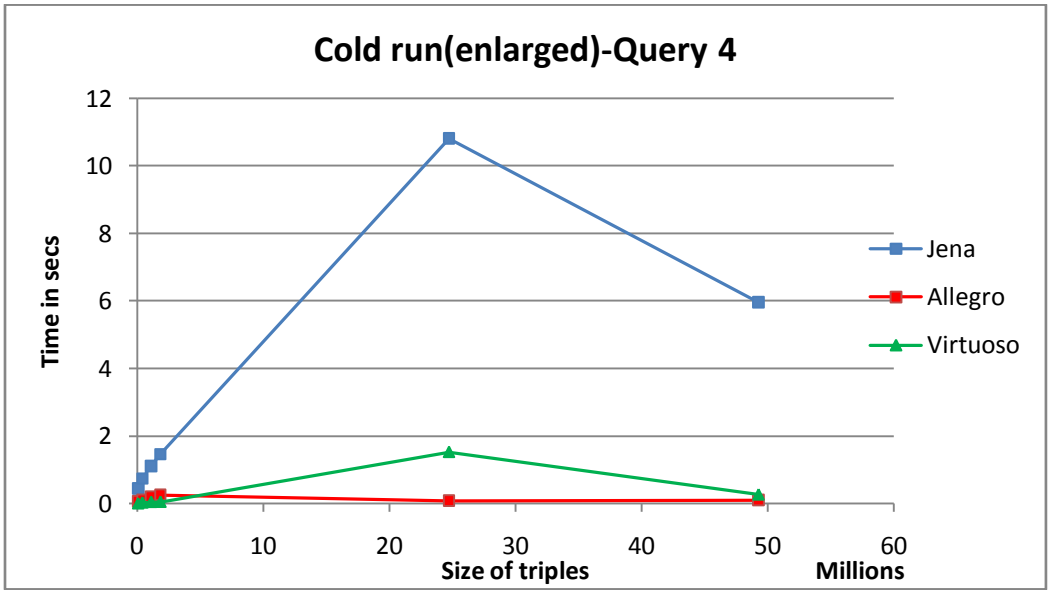


For query 3, both AllegroGraph and Virtuoso clearly show good scalability compared to Jena and Neo4j which seem to perform very slow with increasing number of triples.

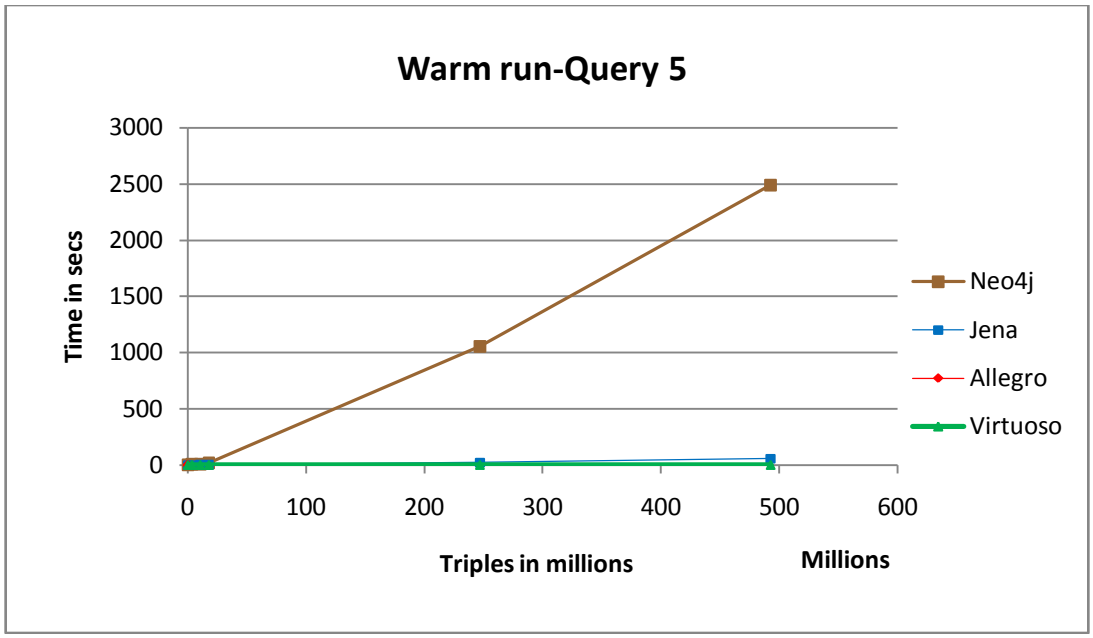

Warm run-Query 4
Time in secs
700
600
500
400
300
200
100
0
0
10
20
30
40
50
60
Millions
Size of triples
Neo4j
Jena
Allegr
o

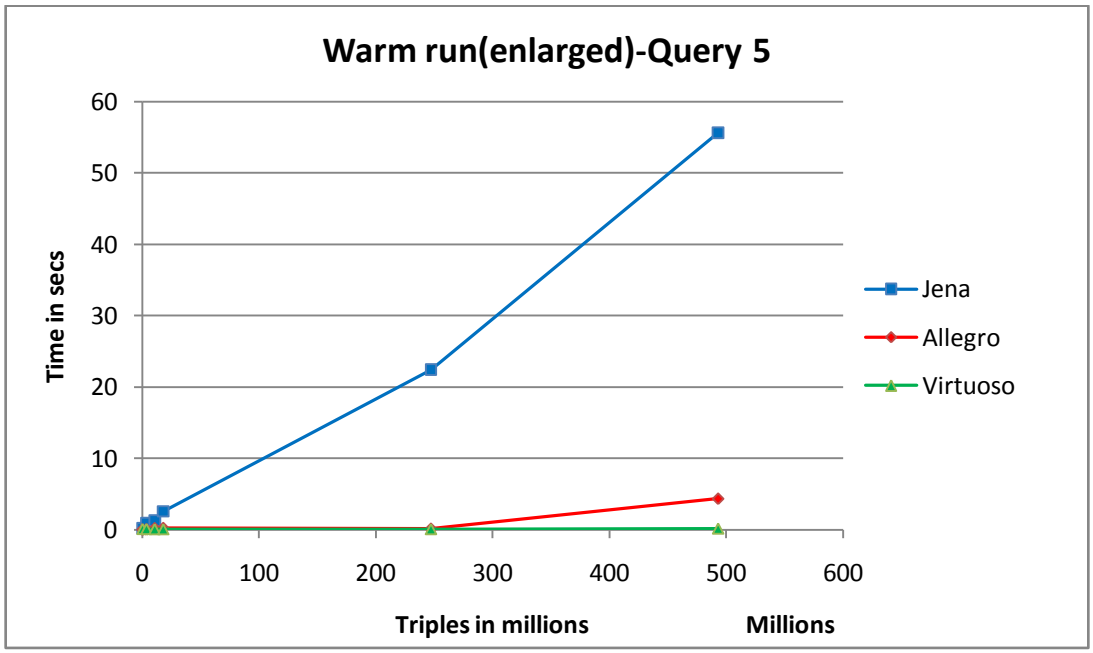

Warm run(enlarged)-Query 4
Time in secs
12
11
10
9
8
7
6
5
4
3
2
1
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena
Allegro
Virtuoso

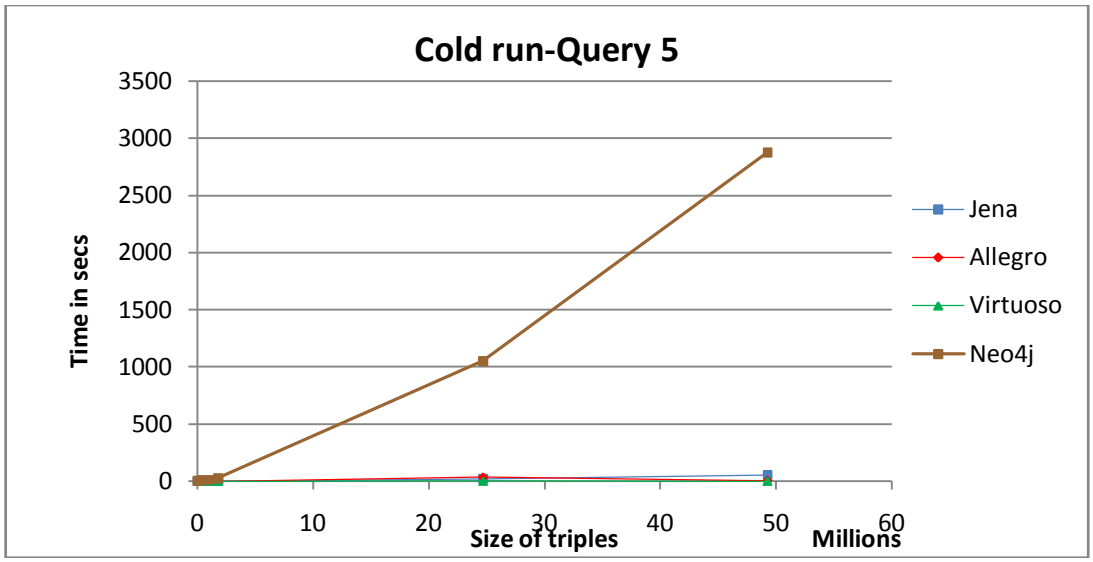


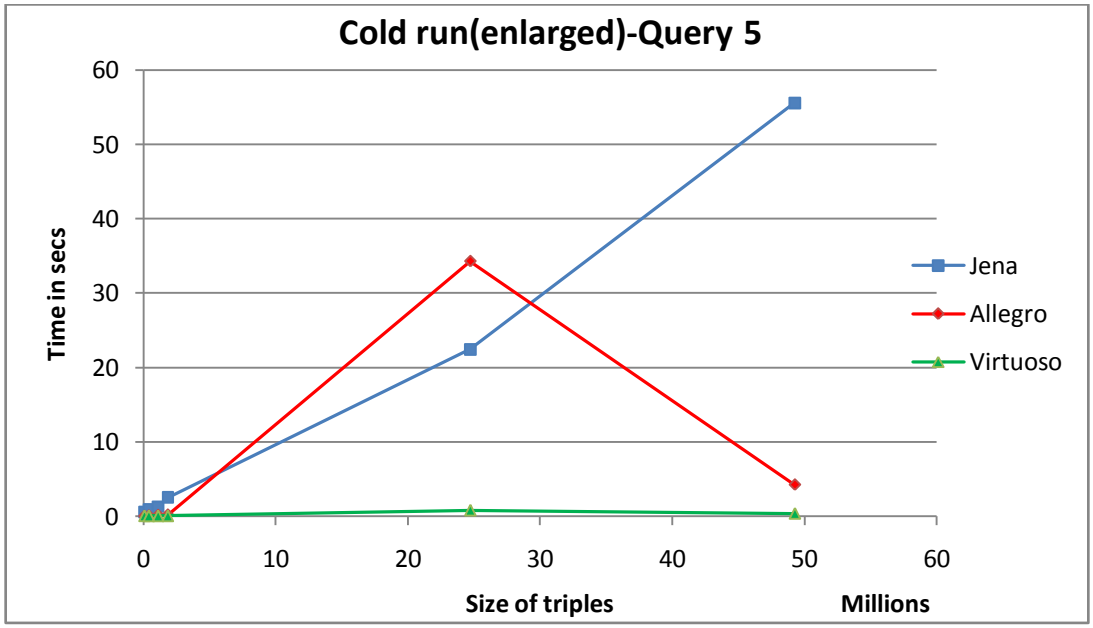


Neo4j does not show good scalability with respect to query 4, while Jena and Virtuoso scaled well as the number of triples increased, since the slope was dropping down.


Warm run-Query 5
Time in secs
3000
2500
2000
1500
1000
500
0
0
100
200
300
400
500
600
Triples in millions
Millions
Neo4j
Jena
Allegro
Virtuoso

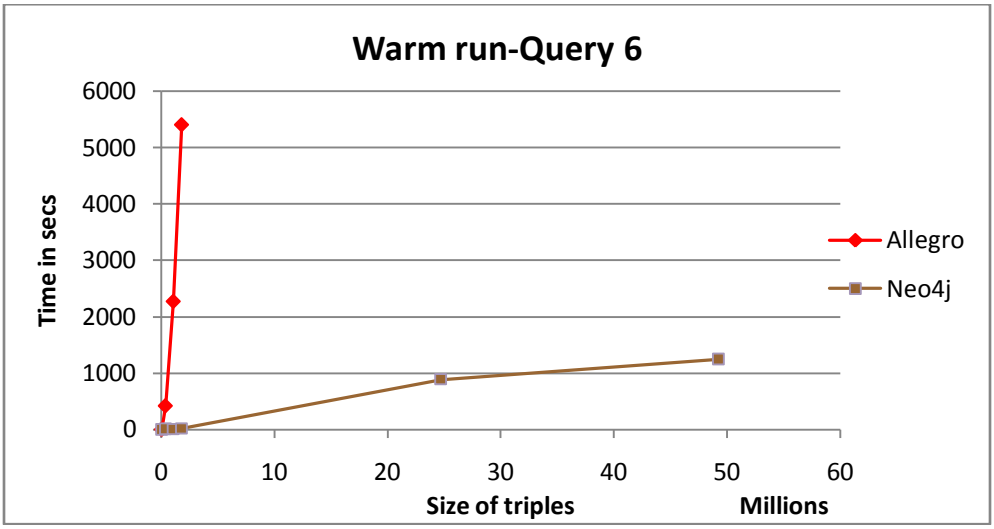

Warm run(enlarged)-Query 5
Time in secs
60
50
40
30
20
10
0
0
100
200
300
400
500
600
Triples in millions
Millions
Jena
Allegro
Virtuoso

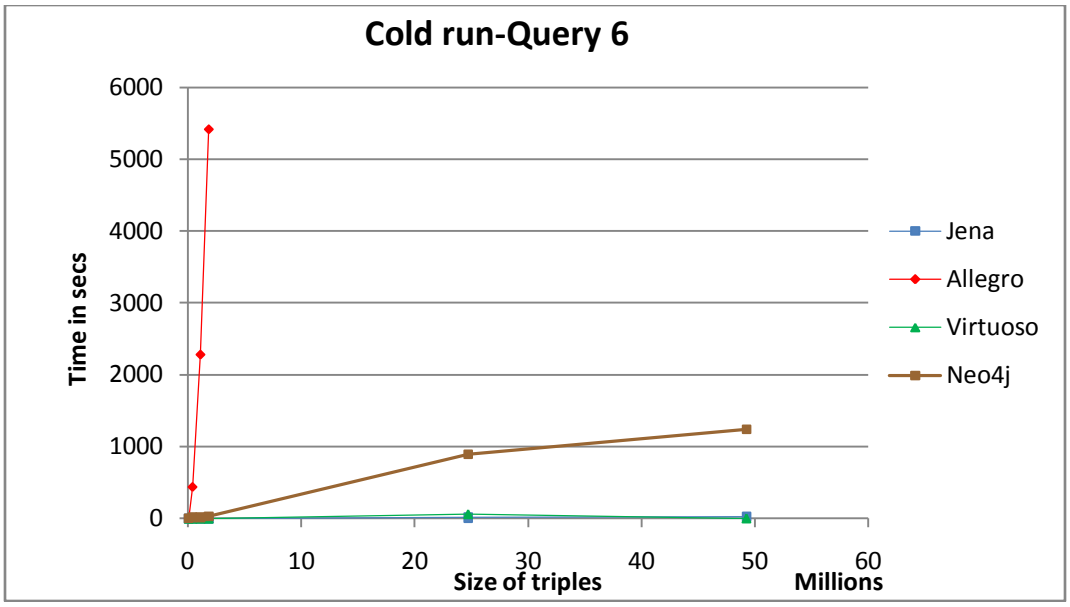




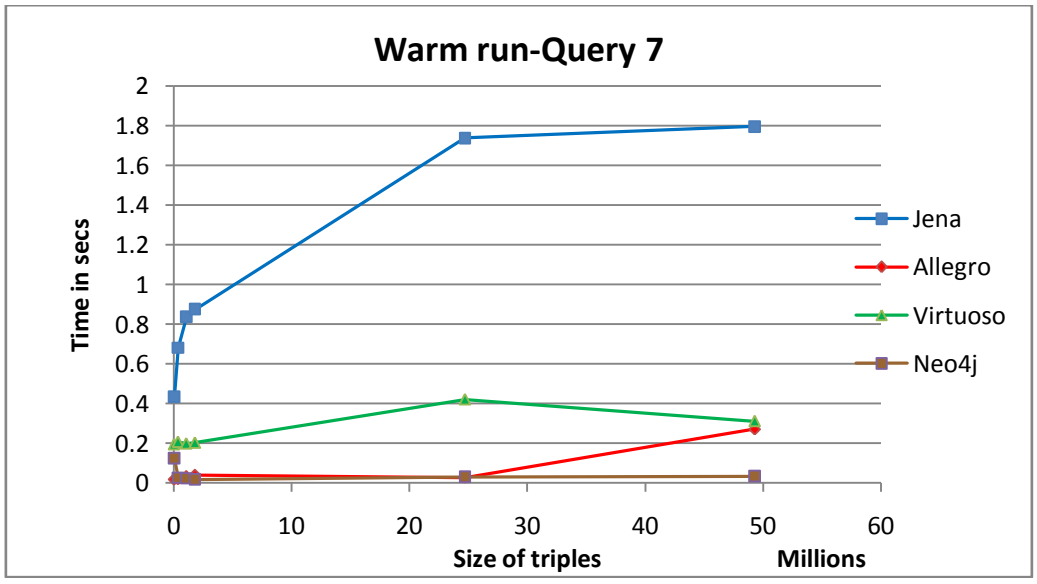
With respect to query 5 both Jena and Ne4j clearly did not scale with the size of the dataset. Virtuoso showed more or less he same performance irrespective of the size of the dataset and scaled very well. AllegroGraph also scaled well. Jena was possibly slower because of the ORDER clause in query 5, a similar increase in Jena's slope is observed in query 8 and 10, probably for the same reason.

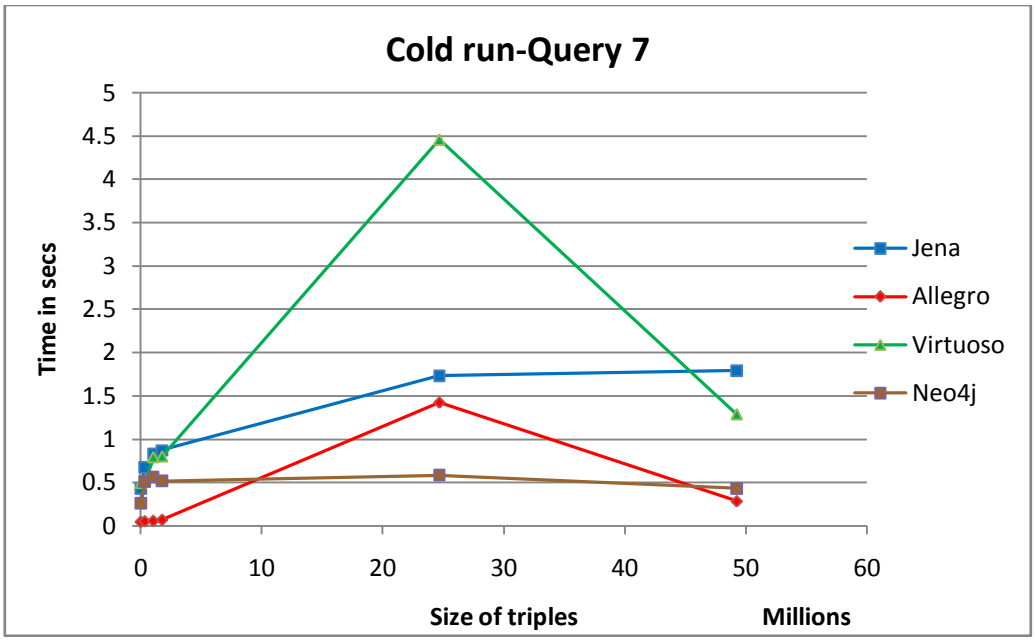

Warm run-Query 6
Time in secs
4
3.5
3
2.5
2
1.5
1
0.5
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena
Virtuoso


Warm run-Query 6
Time in secs
6000
5000
4000
3000
2000
1000
0
0
10
20
30
40
50
60
Size of triples
Millions
Allegro
Neo4j

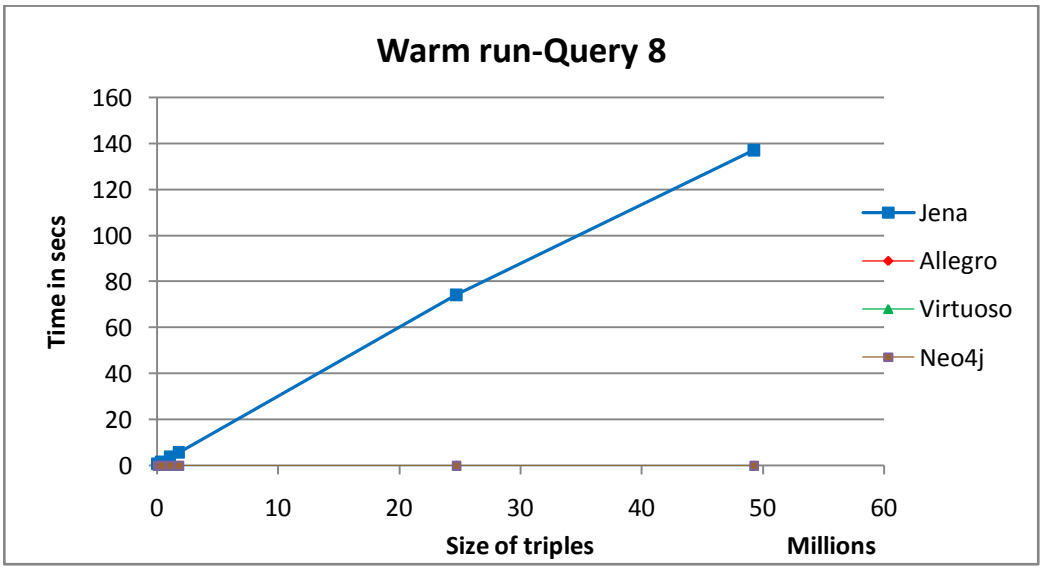


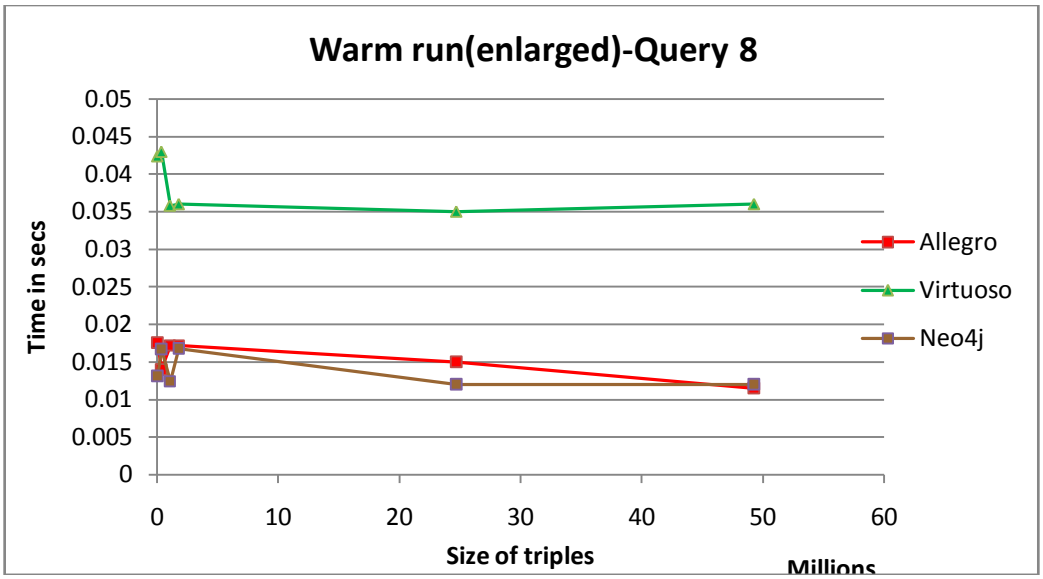
For query 6 AllegroGraph and Neo4j scaled less compared to Jena and Virtuoso. Jena clearly scaled best. Virtuoso was taking a longer response time as the size of the dataset increased .

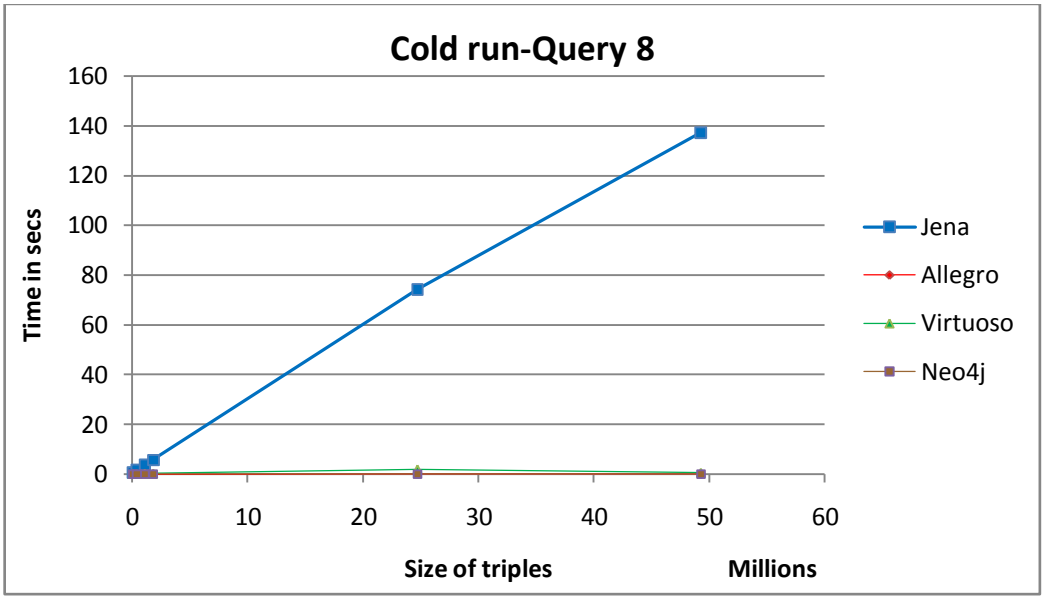


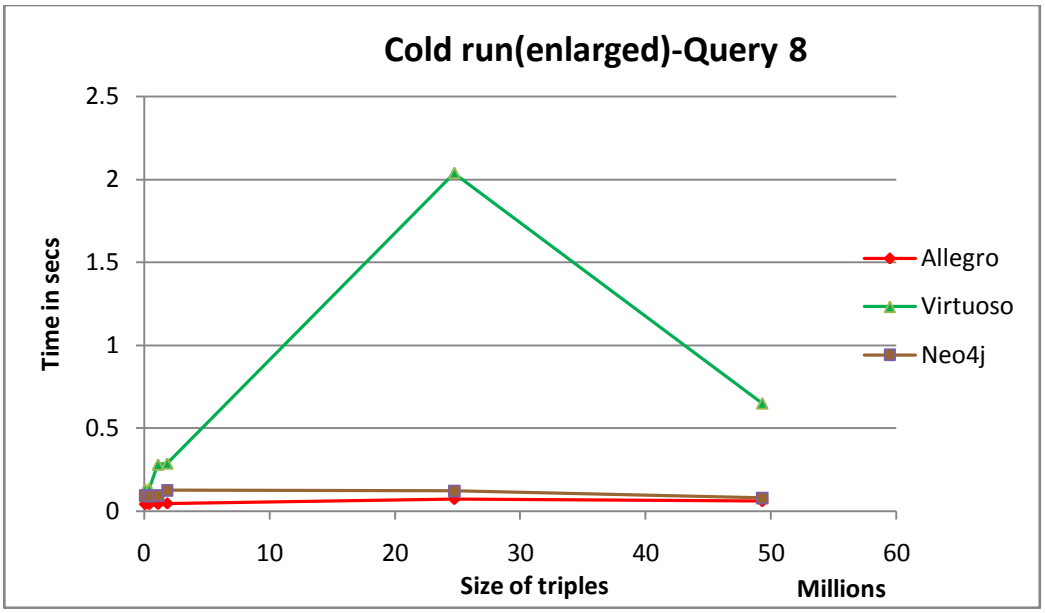


With respect to query 7 Jena does not show good scalability. Allegro and Neo4j performs better than Virtuoso, but Virtuoso has the best scalability trend for larger databases.

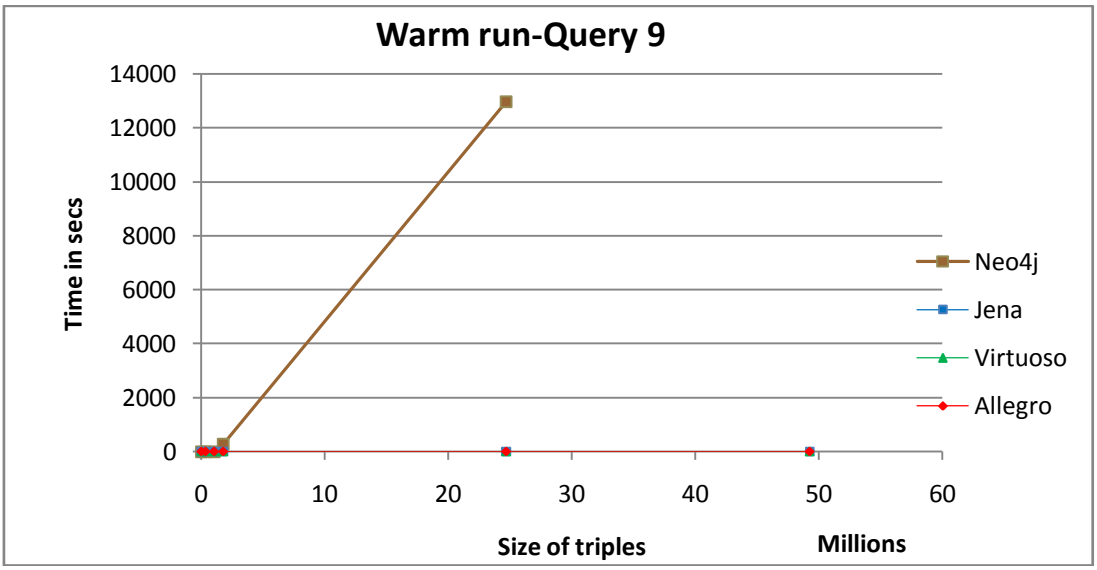


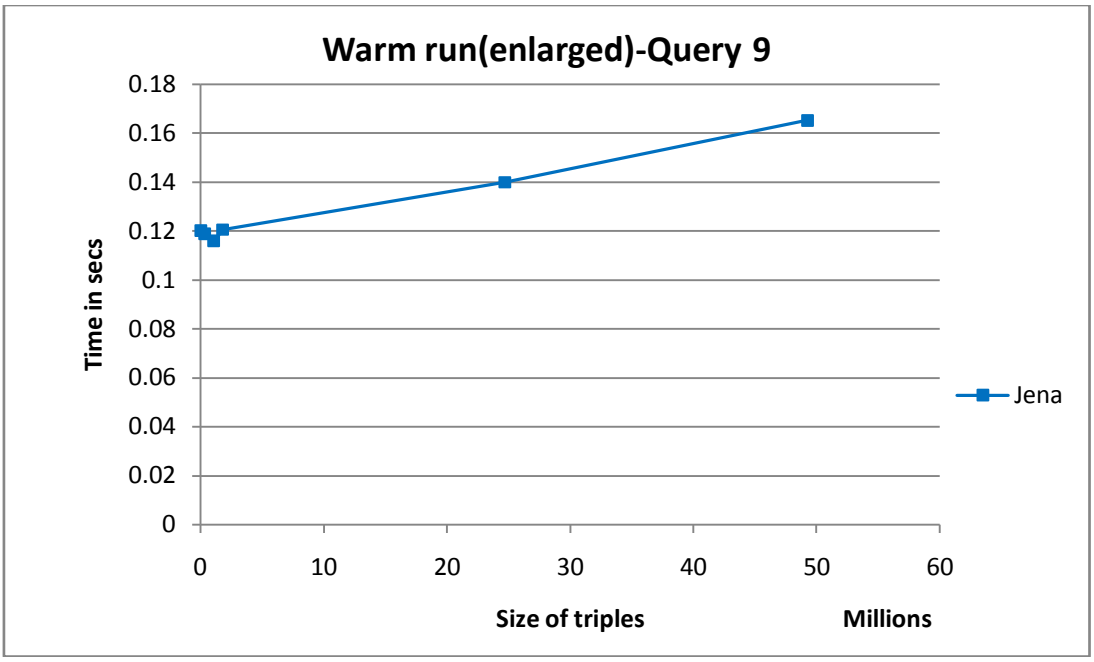

Warm run(enlarged)-Query 8
Time in secs
0.05
0.045
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
10
20
30
40
50
60
Size of triples
Millions
Allegro
Virtuoso
Neo4j

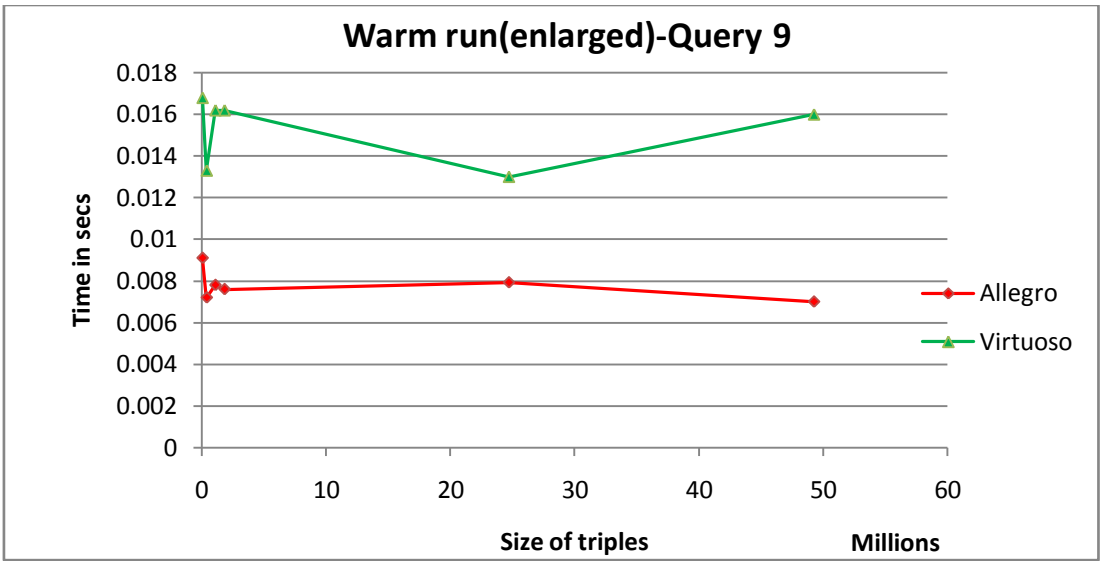

Cold run-Query 8
Time in secs
160
140
120
100
80
60
40
20
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena
Allegro
Virtuoso
Neo4j

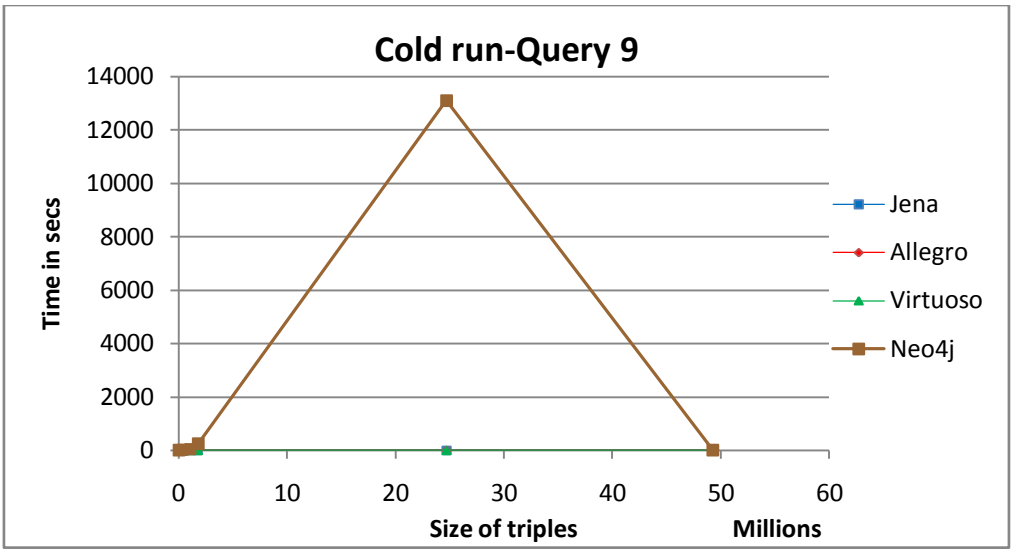


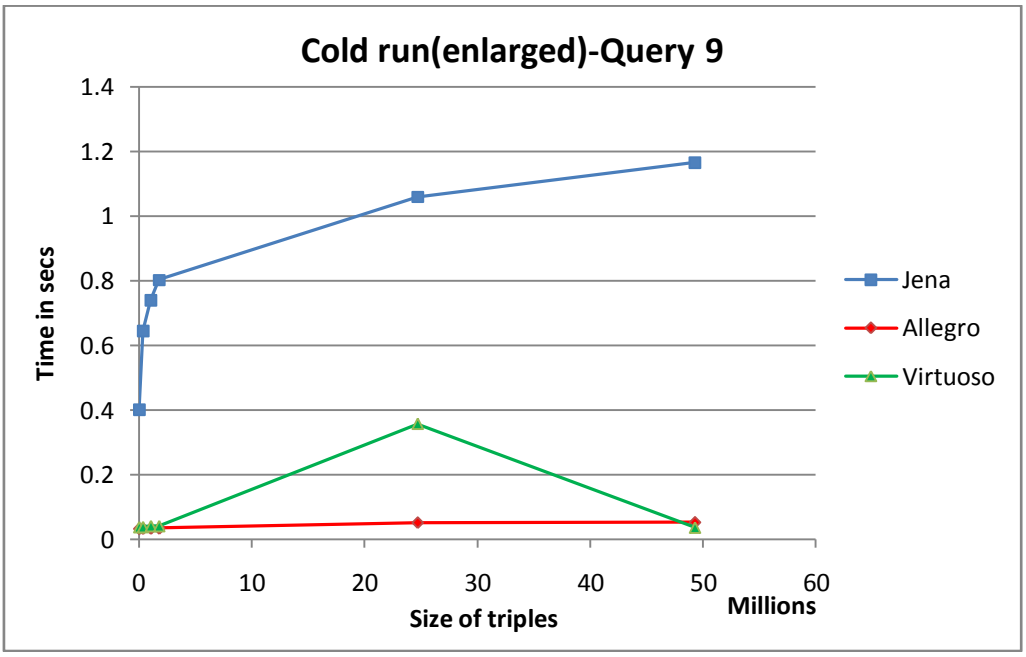
For query 8, clearly Jena did not scale well. AllegroGraph and Neo4j showed a consistent performance for all the datasets irrespective of their size. Again Virtuoso showed good scalability with increasing database size.



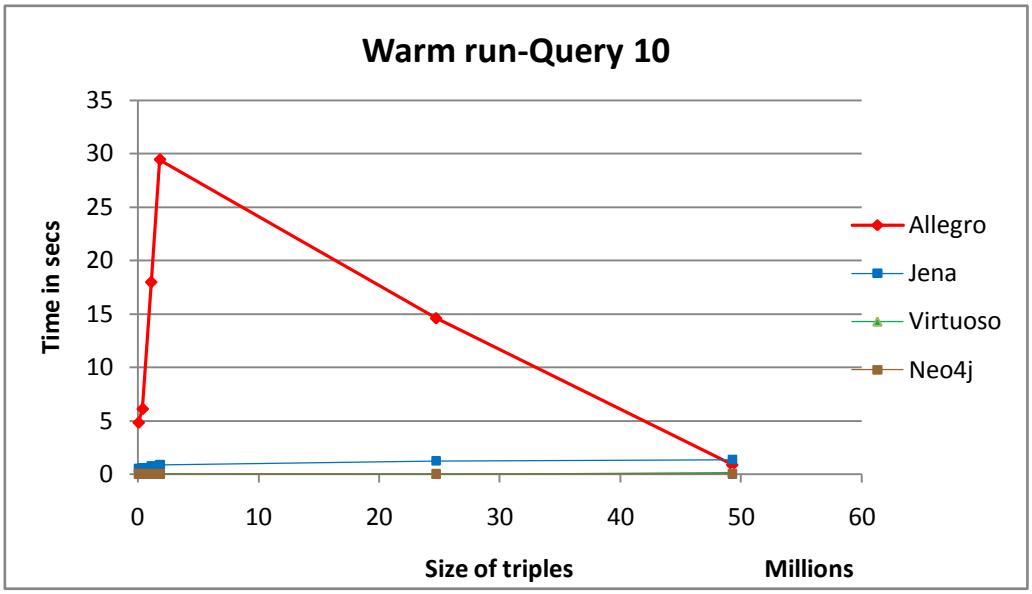

Warm run(enlarged)-Query 9
Time in secs
0.18
0.16
0.14
0.12
0.1
0.08
0.06
0.04
0.02
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena

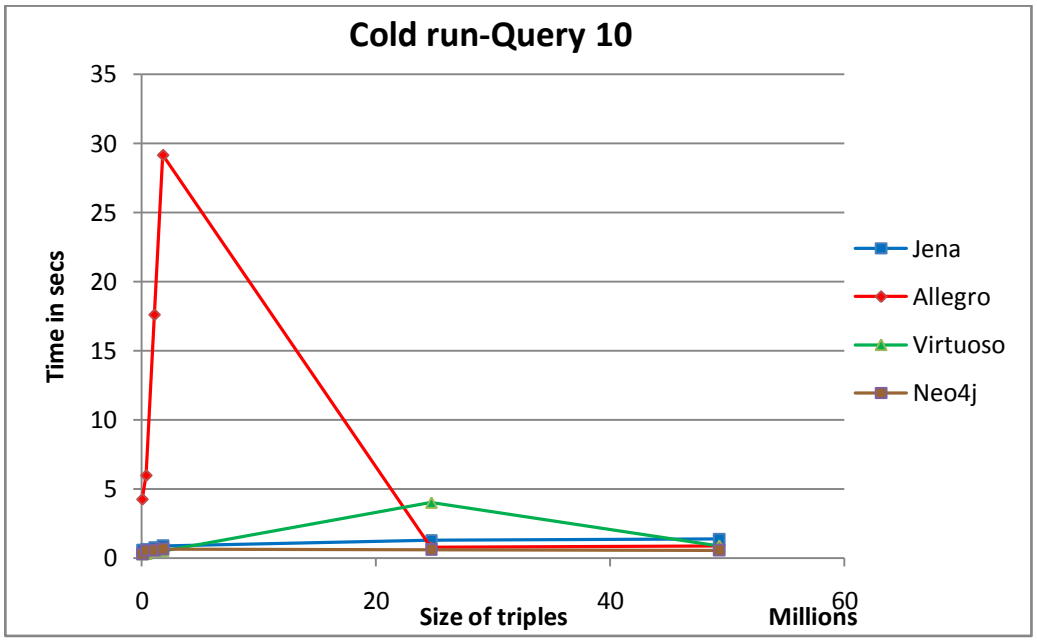

Warm run(enlarged)-Query 9
Time in secs
0.018
0.016
0.014
0.012
0.01
0.008
0.006
0.004
0.002
0
0
10
20
30
40
50
60
Size of triples
Millions
Allegro
Virtuoso

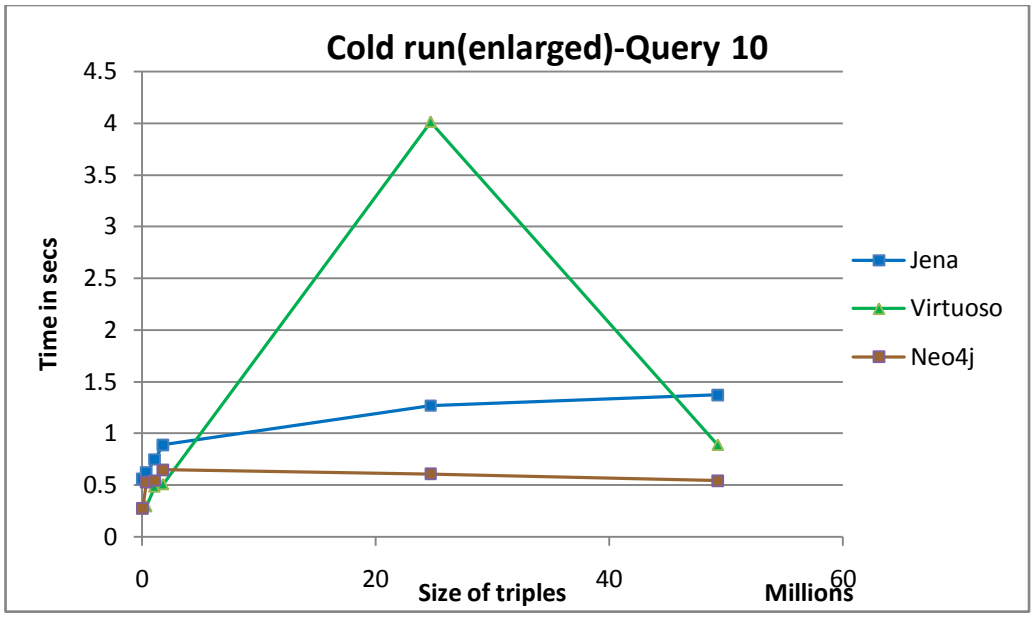




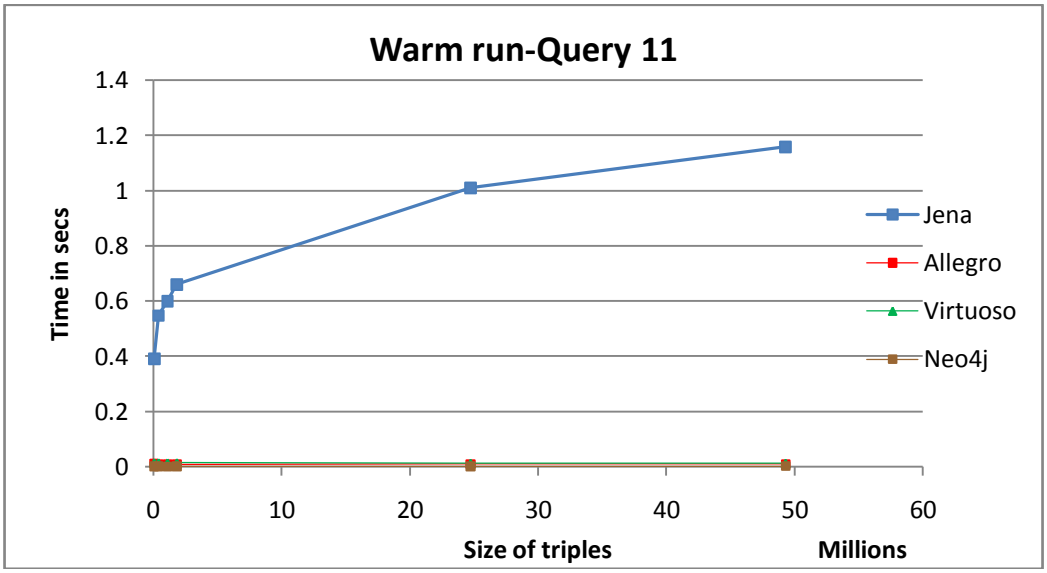
For query 9, Virtuoso, and AllegroGraph scaled well compared to other systems. Neo4J in this case scales better for larger databases. Jena was showing increase in response time as the size of the dataset was increased.

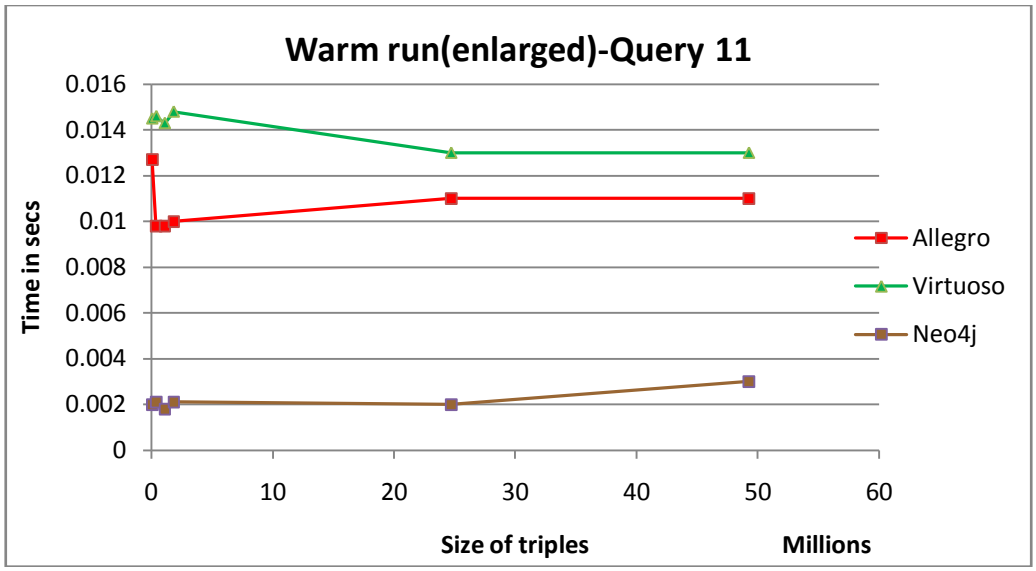

Warm run-Query 10
Time in secs
35
30
25
20
15
10
5
0
0
10
20
30
40
50
60
Size of triples
Millions
Allegro
Jena
Virtuoso
Neo4j

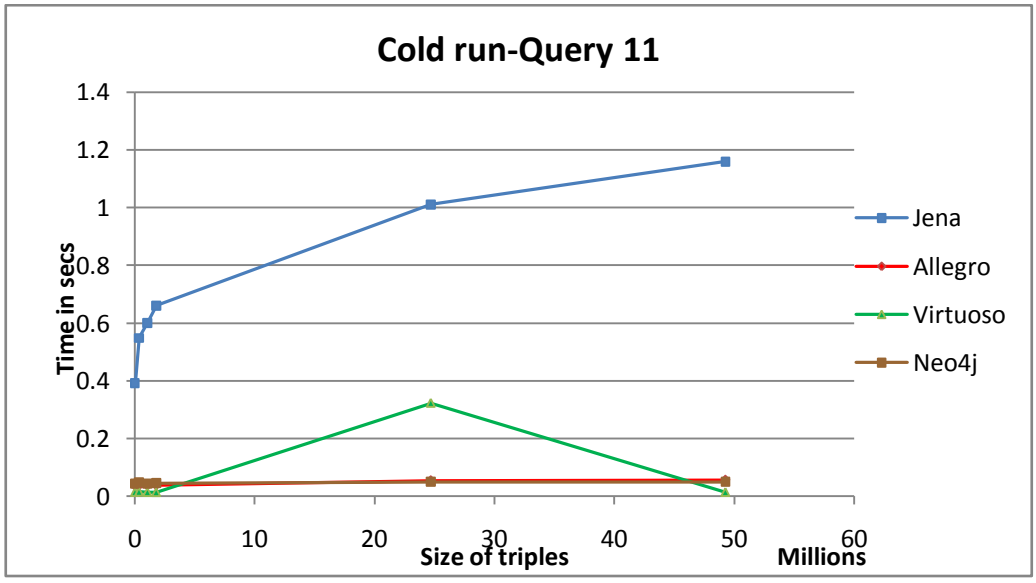

Cold run-Query 10
Time in secs
35
30
25
20
15
10
5
0
0
20
40
60
Size of triples
Millions
Jena
Allegro
Virtuoso
Neo4j



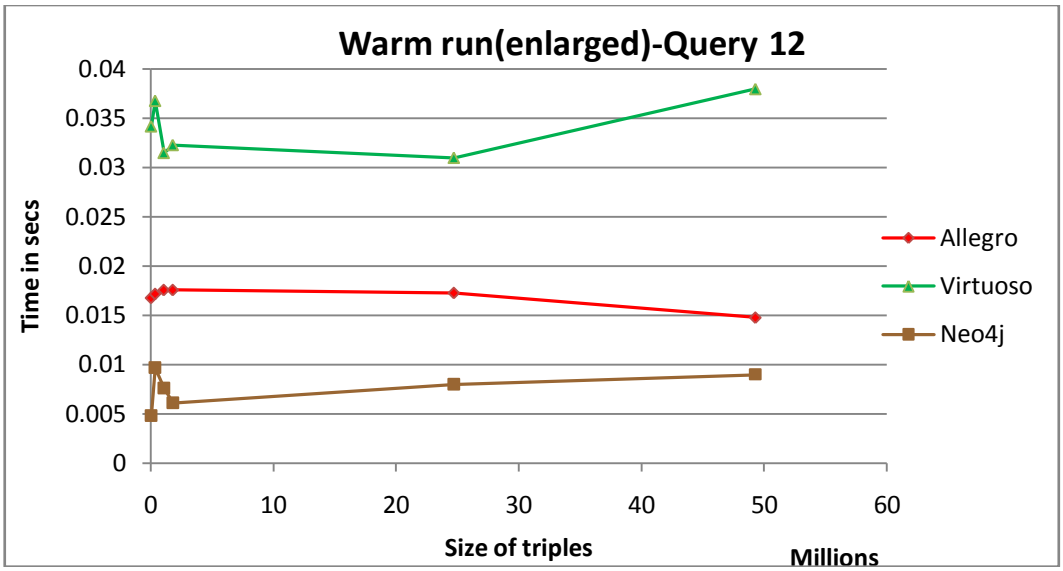
Virtuoso scales better for query 10 especially for the larger dataset which consists of 50 million triples. Jena and Neo4j's response time kept increasing when the size of the dataset increased thus showing poor scalability. It is also observed that AllegroGraph could not show quick response time for smaller datasets but it performed well for larger datasets which contains like ~25 and ~50 millions triples



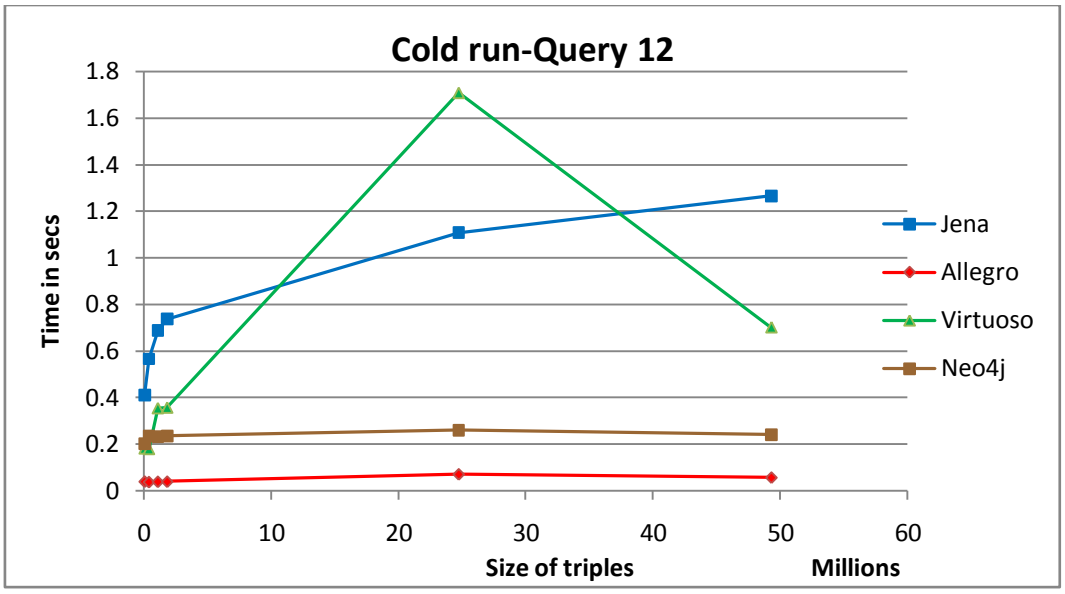




Virtuoso seems to scaled better comparing with increasing data size. Jena clearly did not scale well for query 11. AllegroGraph and Neo4j showed more or less the same performance for all the datasets irrespective of their sizes.


Warm run(enlarged)-Query 12
Time in secs
0.04
0.035
0.03
0.025
0.02
0.015
0.01
0.005
0
0
10
20
30
40
50
60
Size of triples
Millions
Allegro
Virtuoso
Neo4j


Warm run-Query 12
Time in secs
1.4
1.2
1
0.8
0.6
0.4
0.2
0
0
10
20
30
40
50
60
Size of triples
Millions
Jena
Allegro
Virtuoso
Neo4j



Virtuoso showed good scalability with respect query 12, while Jena showed a poor scalability. The response time of AllegroGraph and Neo4j were almost consistent or all their datasets. This is observed through their smooth curve.

8. Conclusion

Experiments were conducted to compare the loading and querying performance of four different RDF storage systems. The used queries and datasets are based on the standard e-commerce Berlin SPARQL benchmark. It was learnt that overall, Virtuoso was better in terms of scalability. Jena SDB was not able to compete with any of the other systems and showed a poor performance (even after tuning MySQL) over. Virtuoso exhibited more or less a consistent performance irrespective of the size of the dataset. Though Neo4j suffered a lot with the loading time, it showed a pretty good performance with querying. In addition it is observed that Neo4j was able to function better with smaller datasets compared to the larger datasets of ~25 and ~50 million triples. As far as loading is concerned both Virtuoso and AllegroGraph were equally good for the smaller datasets where as for the larger one clearly AllegroGraph outperformed Virtuoso. AllegroGraph was better than the other systems in few of the queries, for instance it handled the DESCRIBE query much better than other systems. So of all the systems that were studied, Virtuoso showed good overall performance with very large data sets, while AllegroGraph was better for smaller data sets. Each of these triple stores, considered for study had their own relative merits. Thus we can conclude that Virtuoso and AllegroGraph were distinctly the best systems under the conditions evaluated in this study.

The following table gives a quick overview of scalability of different systems for all the 12 queries

	Jena	AllegroGraph	Virtuoso	Neo4j
Query 1	Good	Average	Average	NR
Query 2	Poor	Average	Good	Poor
Query 3	Poor	Good	Good	Poor
Query 4	Good	Average	Good	Poor
Query 5	Good	Average	Good	Poor
Query 6	Average	Poor	Good	Poor
Query 7	Average	Poor	Good	Average
Query 8	Poor	Average	Good	Average
Query 9	Poor	Average	Good	Good
Query 10	Poor	Good	Good	Poor
Query 11	Poor	Average	Good	Average
Query 12	Poor	Average	Good	Average

Table 3 : Scalability of systems under test

Future Work

To study more about the scalability of the systems, larges database have to be evaluated. This will help us understand how each of the systems perform with very large databases. It is quite natural that the systems show a quick response time for smaller datasets but the real problem arises when the load is too much. In our experiments we limited the size of the triples to 50 million triples.

APPENDIX I - COLD RESULTS

Tabular values of the query response time

Query 1

query 1	Jena	Allegro	Virtuoso	Neo4j
40377	0.409	0.041	0.02	NR
374911	0.676	0.057	0.0216	NR
1075626	0.935	0.0856	0.0286	NR
1809874	1.339	0.0896	0.03	NR
24711725	6.92	0.613	0.198	NR
4927923(	4.136	0.845	0.452	NR

NR – The query did not give any response time

Query 2

query 2	Jena	Allegro	Virtuoso	Neo4j
40377	0.438	0.0563	0.3436	0.225
374911	0.618	0.056	0.3526	0.186
1075626	0.733	0.0536	0.3466	0.186
1809874	0.863	0.056	0.3353	0.2406
24711725	1.187	0.111	2.104	0.41
49279230	1.401	0.065	0.836	0.642

Query 3

query 3	Jena	Allegro	Virtuoso	Neo4j
40377	0.452	0.0532	0.03	1.406
374911	0.795	0.0786	0.035	3.814
1075626	1.125	0.1323	0.044	5.402
1809874	1.842	0.1396	0.044	8.3833
24711725	14.398	10.917	5.894	307.145
49279230	19.098	0.779	0.129	2132.296

Query 4

query 4	Jena	Allegro	Virtuoso	Neo4j
40377	0.44	0.0533	0	1.688
374911	0.736	0.1013	0.0256	4.766
1075626	1.107	0.193	0.0423	6.849

1809874	1.459	0.251	0.046	10.384
24711725	10.817	0.08	1.524	568.911
49279230	5.963	0.09	0.263	1414.526

Query 5

query 5	Jena	Allegro	Virtuoso	Neo4j
40377	0.547	0.0663	0.108	4.302
374911	0.886	0.1063	0.0973	11.19
1075626	1.284	0.19	0.0956	15.369
1809874	2.565	0.2403	0.0983	28.419
24711725	22.455	34.327	0.856	1053.683
49279230	55.613	4.287	0.427	2880.55

Query 6

query 6	Jena	Allegro	Virtuoso	Neo4j
40377	0.444	5.0903	0.014	1.5116
374911	0.828	438	0.0433	13.211
1075626	1.205	2280	0.0866	12.858
1809874	1.914	5414	0.1236	26.062
24711725	12.388	NR	60.143	891.346
49279230	24.187	NR	3.369	1239.272

Query 7

query 7	Jena	Allegro	Virtuoso	Neo4j
40377	0.433	0.0476	0.4936	0.263
374911	0.68	0.0536	0.5155	0.5103
1075626	0.836	0.059	0.7923	0.5593
1809874	0.875	0.0713	0.801	0.5156
24711725	1.738	1.426	4.463	0.584
49279230	1.796	0.285	1.288	0.431

Query 8

query 8	Jena	Allegro	Virtuoso	Neo4j
40377	0.736	0.0406	0.1326	0.095
374911	1.769	0.0406	0.1355	0.0896
1075626	3.894	0.0416	0.2813	0.0953
1809874	5.763	0.045	0.288	0.126

24711725	74.33	0.072	2.038	0.122
49279230	137.26	0.059	0.65	0.078

Query 9

query 9	Jena	Allegro	Virtuoso	Neo4j
40377	0.401	0.0333	0.0383	2.987
374911	0.542	0.0336	0.0386	15.218
1075626	0.669	0.034	0.0416	43.752
1809874	0.721	0.0343	0.0416	245.576
24711725	1.059	0.052	0.357	13115.42
49279230	1.165	0.053	0.037	NR

NR – The response time was unacceptably large

Query 10

query 10	Jena	Allegro	Virtuoso	Neo4j
40377	0.562	4.2656	0.276	0.2756
374911	0.623	5.9893	0.2992	0.5323
1075626	0.7513	17.606	0.489	0.5473
1809874	0.892	29.144	0.512	0.6513
24711725	1.271	0.773	4.013	0.611
49279230	1.374	0.866	0.892	0.546

Query 11

query 11	Jena	Allegro	Virtuoso	Neo4j
40377	0.392	0.037	0.015	0.0426
374911	0.549	0.037	0.016	0.048
1075626	0.601	0.038	0.015	0.0436
1809874	0.661	0.0383	0.0147	0.0453
24711725	1.011	0.055	0.323	0.049
49279230	1.16	0.057	0.015	0.05

Query 12

query 12	Jena	Allegro	Virtuoso	Neo4j

40377	0.411	0.04	0.1825	0.203
374911	0.567	0.038	0.18	0.2366
1075626	0.689	0.0393	0.3553	0.233
1809874	0.739	0.0396	0.358	0.236
24711725	1.11	0.071	1.71	0.261
49279230	1.268	0.057	0.703	0.241

APPENDIX II - WARM RESULTS

Tabular values of the query response time

Query 1

query 1	Jena	Allegro	Virtuoso	Neo4j
40377	0.1285	0.0152	0.012	NR
374911	0.676	0.0273	0.0132	NR
1075626	0.935	0.0577	0.0143	NR
1809874	1.339	0.0661	0.0143	NR
24711725	6.92	0.050367	0.15	NR
49279230	4.136	0.7564	0.25	NR

NR – The query did not give any response time

Query 2

query 2	Jena	Allegro	Virtuoso	Neo4j
40377	0.1435	0.0288	0.0405	0.0414
374911	0.618	0.0244	0.0399	0.0366
1075626	0.733	0.0266	0.0369	0.035
1809874	0.863	0.026	0.0384	0.0419
24711725	1.187	0.02645	0.037	0.039
49279230	1.401	0.0647	0.129	0.041

Query 3

query 3	Jena	Allegro	Virtuoso	Neo4j
40377	0.1333	0.016	0.0248	0.1381
374911	0.795	0.0417	0.0255	0.4923
1075626	1.125	0.0999	0.0278	0.7503
1809874	1.842	0.101	0.026	1.417
24711725	14.398	0.06465	0.187	276.92
49279230	19.09	0.7694	0.046	1820.662

Query 4

query 4	Jena	Allegro	Virtuoso	Neo4j
40377	0.1305	0.0157	0	0.1743
374911	0.736	0.0727	0.0163	0.7933

1075626	1.107	0.1707	0.0187	1.487
1809874	1.459	0.2661	0.018	2.9
24711725	10.817	0.1313	0.291	377.83
49279230	5.963	0.0483	0.1815	601.275

Query 5

query 5	Jena	Allegro	Virtuoso	Neo4j
40377	0.1813	0.0434	0.0937	1.588
374911	0.886	0.074	0.0945	6.8626
1075626	1.284	0.2168	0.0627	9.5485
1809874	2.565	0.2521	0.0641	20.5471
24711725	22.455	0.146575	0.07	1053.683
49279230	55.61	4.334	0.153	2488.837

Query 6

query 6	Jena	Allegro	Virtuoso	Neo4j
40377	0.124	4.97	0.0093	0.269
374911	0.1808	424	0.0336	9.833
1075626	0.2845	2276	0.0709	8.1353
1809874	0.3671	5409	0.1024	18.9203
24711725	2.256	NR	1.937	891.346
49279230	24.187	NR	3.337	1248.981

NR – The response time was unacceptably large

Query 7

query 7	Jena	Allegro	Virtuoso	Neo4j
40377	0.433	0.0173	0.1953	0.1246
374911	0.68	0.019	0.2078	0.0257
1075626	0.836	0.0327	0.1995	0.0231
1809874	0.875	0.0393	0.2035	0.0175
24711725	1.738	0.027075	0.421	0.031
49279230	1.796	0.2705	0.312	0.033

Query 8

query 8	Jena	Allegro	Virtuoso	Neo4j
40377	0.736	0.0176	0.0424	0.0131
374911	1.769	0.014	0.043	0.0167

1075626	3.894	0.0172	0.0358	0.0124
1809874	5.763	0.0172	0.036	0.0168
24711725	74.33	0.015	0.035	0.012
49279230	137.26	0.0115	0.036	0.012

Query 9

query 9	Jena	Allegro	Virtuoso	Neo4j
40377	0.401	0.0091	0.0168	0.179
374911	0.542	0.0072	0.0133	1.658
1075626	0.669	0.0078	0.0162	4.854
1809874	0.721	0.0076	0.0162	285.348
24711725	1.059	0.007925	0.013	12967.18
49279230	1.165	0.007	0.016	NR

NR – The response time was unacceptably large

Query 10

query 10	Jena	Allegro	Virtuoso	Neo4j
40377	0.562	4.8483	0.0353	0.0205
374911	0.623	6.128	0.0418	0.0243
1075626	0.751	17.9746	0.036	0.0225
1809874	0.903	29.4353	0.0415	0.0231
24711725	1.271	14.59655	0.0114	0.02
49279230	1.374	0.8876	0.164	0.028

Query 11

query 11	Jena	Allegro	Virtuoso	Neo4j
40377	0.392	0.0127	0.0145	0.002
374911	0.549	0.0098	0.0146	0.0021
1075626	0.601	0.0098	0.0143	0.0018
1809874	0.661	0.01	0.0148	0.0021
24711725	1.011	0.011	0.013	0.002
49279230	1.16	0.011	0.013	0.003

Query 12

query 12	Jena	Allegro	Virtuoso	Neo4j
40377	0.411	0.0168	0.0342	0.0048
374911	0.567	0.0172	0.0368	0.0097
1075626	0.689	0.0176	0.0315	0.0076
1809874	0.739	0.0176	0.0323	0.0061
24711725	1.11	0.0173	0.031	0.008
49279230	1.268	0.0148	0.038	0.009

APPENDIX III
FORMULATED SET OF QUERIES

Query 1

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> SELECT DISTINCT
?product ?label
WHERE {
?product rdfs:label ?label .
?product rdf:type <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType2> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12>.
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
?product bsbm:productPropertyNumeric1 ?value1 .
FILTER (?value1 > 348)}
ORDER BY ?label
LIMIT 10
```

Query 2

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX inst:
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#> PREFIX dc:
<http://purl.org/dc/elements/1.1/>
SELECT ?label ?comment ?producer ?productFeature ?propertyTextual1
?propertyTextual2 ?propertyTextual3 ?propertyNumeric1
?propertyNumeric2 ?propertyTextual4 ?propertyTextual5
?propertyNumeric4
WHERE {
inst:Product5  rdfs:label ?label .
inst:Product5  rdfs:comment ?comment . inst:Product5
            bsbm:producer ?p .
?p rdfs:label ?producer .
inst:Product5  dc:publisher ?p . inst:Product5
            bsbm:productFeature ?f .
?f rdfs:label ?productFeature .
inst:Product5  bsbm:productPropertyTextual1 ?propertyTextual1 .
inst:Product5  bsbm:productPropertyTextual2 ?propertyTextual2 .
inst:Product5  bsbm:productPropertyTextual3 ?propertyTextual3 .
inst:Product5  bsbm:productPropertyNumeric1 ?propertyNumeric1 .
inst:Product5  bsbm:productPropertyNumeric2 ?propertyNumeric2 .
OPTIONAL {inst:Product5    bsbm:productPropertyTextual4 ?propertyTextual4 }
OPTIONAL {inst:Product5    bsbm:productPropertyTextual5 ?propertyTextual5 }
OPTIONAL {inst:Product5    bsbm:productPropertyNumeric4 ?propertyNumeric4 }}
```

Query 3

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> SELECT ?product ?label
WHERE {
?product rdfs:label ?label .
?product rdf:type <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType2> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12> .
?product bsbm:productPropertyNumeric1 ?p1 .
FILTER ( ?p1 > 214 )
?product bsbm:productPropertyNumeric3 ?p3 . FILTER (?p3 < 698 )
OPTIONAL {
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
?product rdfs:label ?testVar } FILTER
(!bound(?testVar)) } ORDER BY ?label
LIMIT 10
```

Query 4

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
SELECT ?product ?label
WHERE {
{ ?product rdfs:label ?label .
?product rdf:type <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType2> .
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12>.
?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
?product bsbm:productPropertyNumeric1 ?value1 .
FILTER (?value1 > 348)}
UNION
{
?product rdfs:label ?label .
?product rdf:type    <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType1>.
?product bsbm:productFeature     <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12> .
?product bsbm:productFeature     <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature8> .
?product bsbm:productPropertyNumeric2 ?p2 . FILTER ( ?p2>759 )
}
}
ORDER BY ?label
LIMIT 10
OFFSET 10
```

Query 5

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX rdf:
<http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> PREFIX inst:
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/>
SELECT DISTINCT ?product ?productLabel
WHERE {
?product rdfs:label ?productLabel .
FILTER (inst:Product6      != ?product)
inst:Product6  bsbm:productFeature ?prodFeature .
?product bsbm:productFeature ?prodFeature .
inst:Product6  bsbm:productPropertyNumeric1 ?origProperty1 .
?product bsbm:productPropertyNumeric1 ?simProperty1 .
FILTER (?simProperty1 < (?origProperty1 + 120) && ?simProperty1 > (?origProperty1 - 120))
inst:Product6  bsbm:productPropertyNumeric2 ?origProperty2 .
?product bsbm:productPropertyNumeric2 ?simProperty2 .
FILTER (?simProperty2 < (?origProperty2 + 170) && ?simProperty2 >
(?origProperty2 - 170)) } ORDER BY ?productLabel
LIMIT 5
```

Query 6

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> SELECT ?product ?label
WHERE {
?product rdfs:label ?label .
?product rdf:type bsbm:Product .
FILTER regex(?label, "r")}
```

Query 7

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX foaf:
<http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#> PREFIX dc:
<http://purl.org/dc/elements/1.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
SELECT ?productLabel ?offer ?price ?vendor ?vendorTitle ?review
?revTitle ?reviewer ?revName ?rating1 ?rating2
WHERE {
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product5> rdfs:label
?productLabel .
OPTIONAL {
?offer bsbm:product <http://www4.wiwiss.fu-
```

```
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product5> .
?offer bsbm:price ?price .
?offer bsbm:vendor ?vendor .
?vendor rdfs:label ?vendorTitle .
?vendor bsbm:country <http://downlode.org/rdf/iso-3166/countries#DE>.
?offer dc:publisher ?vendor .
?offer bsbm:validTo ?date .
FILTER (?date > 2001-09-16 ) }
OPTIONAL {
?review bsbm:reviewFor <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product5> .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?revName .
?review dc:title ?revTitle .
OPTIONAL { ?review bsbm:rating1 ?rating1 . }
OPTIONAL { ?review bsbm:rating2 ?rating2 . } } }
```

Query 8

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX
inst:<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX rev: <http://purl.org/stuff/rev#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?title ?text ?reviewDate ?reviewer ?reviewerName ?rating1
?rating2 ?rating3 ?rating4
WHERE {
?review bsbm:reviewFor inst:Product6   .
?review dc:title ?title .
?review rev:text ?text .
FILTER langMatches( lang(?text), "EN" )
?review bsbm:reviewDate ?reviewDate .
?review rev:reviewer ?reviewer .
?reviewer foaf:name ?reviewerName .
OPTIONAL { ?review bsbm:rating1 ?rating1 . }
OPTIONAL { ?review bsbm:rating2 ?rating2 . }
OPTIONAL { ?review bsbm:rating3 ?rating3 . } OPTIONAL { ?review
bsbm:rating4 ?rating4 . } }
ORDER BY DESC(?reviewDate) LIMIT 20
```

Query 9

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#> PREFIX rev:
<http://purl.org/stuff/rev#>
DESCRIBE ?x
WHERE {
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromRatingSite1/Review4>      rev:reviewer
```

```
?x }
```

Query 10

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
SELECT DISTINCT ?offer ?price
WHERE {
?offer bsbm:product <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromProducer1/Product1> .
?offer bsbm:vendor ?vendor .
?offer dc:publisher ?vendor .
?vendor bsbm:country                                    <http://downlode.org/rdf/iso-
3166/countries#GB> .
?offer bsbm:deliveryDays ?deliveryDays .
FILTER (?deliveryDays <= 3)
?offer bsbm:price ?price .
?offer bsbm:validTo ?date .
FILTER (?date >     "2008-04-19T00:00:00"^^xsd:dateTime)
}
ORDER BY xsd:double(str(?price))
LIMIT 10
```

Query 11

```
           SELECT ?property ?hasValue ?isValueOf
WHERE {
{ <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>     ?property
?hasValue }
UNION
{ ?isValueOf ?property <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>     } }
```

Query 12

```
PREFIX bsbm: <http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> PREFIX rdfs:
<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/> PREFIX bsbm-export:
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/export/>

CONSTRUCT {
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>
bsbm- export:product ?productURI .
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>
bsbm-
export:productlabel ?productlabel .
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm-export:vendor
?vendorname .
```

```
<Error! Hyperlink reference not valid.> bsbm- export:vendorhomepage ?vendorhomepage .
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>
bsbm- export:offerURL ?offerURL .
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm-
export:price
?price .
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm-
export:deliveryDays ?deliveryDays .
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm- export:validuntil
?validTo }
WHERE {
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>bsbm:product
?productURI .
?productURI rdfs:label ?productlabel .
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm:vendor
?vendorURI .
?vendorURI rdfs:label ?vendorname .
?vendorURI foaf:homepage ?vendorhomepage .
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm:offerWebpage
?offerURL .
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm:price
?price
.
<http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1> bsbm:deliveryDays
?deliveryDays .
<http://www4.wiwiss.fu- berlin.de/bizer/bsbm/v01/instances/dataFromVendor1/Offer1>
bsbm:validTo
?validTo }
```

APPENDIX I V
SOURCE CODE SAMPLE
Loading and indexing of data in AllegroGraph

```
package com.franz.agbase.examples;
import com.franz.agbase.*;
public class loadNT_100 {

               /**
* Demonstrates how to load a triple store from N-Triples files.
*
* @param args unused
* @throws AllegroGraphException
*/
public static void main(String[] args) throws AllegroGraphException {

// Connect to server, which must already be running. AllegroGraphConnection ags =
new AllegroGraphConnection();
                  try {
ags.enable();
}
catch (Exception e)
{
throw new AllegroGraphException("Server connection problem", e);
}

// Create fresh triple-store.
AllegroGraph ts = ags.renew("load100ntriples",AGPaths.TRIPLE_STORES);

// Load a file in N-Triples format
String ntripleFile = AGPaths.dataSources("dataset_100.nt");
System.out.println("Loading N-Triples " + ntripleFile);
String report;
String result;
long start = System.currentTimeMillis() ;
ts.loadNTriples(ntripleFile);

report=AGUtils.elapsedTime(start); System.out.println("Loading Time " + report);
System.out.println("Loaded " + ts.numberOfTriples() + " triples.");


// Index the triple store for faster querying long starts =
System.currentTimeMillis() ; ts.indexAllTriples();
result=AGUtils.elapsedTime(starts); System.out.println("Indexing Time
" + result);

// Close the store and disconnect from the server.
ts.closeTripleStore();
ags.disable();
              }
}
```

Querying in AllegroGraph

```
package com.franz.agbase.examples;

import com.franz.agbase.*;

/**
 * Demonstrates issuing a simple SPARQL SELECT query and showing results.
 *
 */
public class AGSparqlDistinct {

public static void main(String[] args) throws AllegroGraphException {
// Connect to server, which must already be running.
AllegroGraphConnection ags = new AllegroGraphConnection();
try {
      ags.enable();
    }
catch (Exception e)
{
 throw new AllegroGraphException("Server connection problem", e);
}

AllegroGraph ts = ags.access("load140000ntriples", AGPaths.TRIPLE_STORES);

//A simple SPARQL SELECT query

String query = "PREFIX bsbm: <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/> "+
"PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> "+
"PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> "+
"SELECT DISTINCT ?product ?label "+
"WHERE { "+
"?product rdfs:label ?label . "+
"?product rdf:type <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType2> . "+
"?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12>."+
"?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> . "+
"?product bsbm:productPropertyNumeric1 ?value1 . "+
"FILTER (?value1 > 348)} "+
"ORDER BY ?label "+
"LIMIT 10";

// Set up the SPARQLQuery object
            SPARQLQuery sq = new SPARQLQuery();
            sq.setTripleStore(ts);
            sq.setQuery(query);


// Do the SELECT query and show results
            doSparqlSelect(sq);
}
```

```
       * A convenience method for showing SPARQL SELECT queries to the user,
it prints
       * the given query and its results.
       *
       * @param sq a SPARQLQuery object with any optional parameters set.
       *
       * @throws AllegroGraphException
       */
public static void doSparqlSelect(SPARQLQuery sq) throws
AllegroGraphException
{
            String report;
            long start = System.currentTimeMillis() ;
            ValueSetIterator it = sq.select();
            report=AGUtils.elapsedTime(start);
            System.out.println("Exec time: " + report);
            AGUtils.showResults(it);

      }
}
```

Querying in Virtuoso

```
import com.hp.hpl.jena.query.*;
import com.hp.hpl.jena.rdf.model.RDFNode;
import com.hp.hpl.jena.graph.Triple;
import com.hp.hpl.jena.graph.Node;
import com.hp.hpl.jena.graph.Graph;
import com.hp.hpl.jena.rdf.model.*;
import java.util.Iterator;
import java.lang.System;
import java.util.*;

import virtuoso.jena.driver.*;

public class Query1 {

      /**
       * Executes a SPARQL query against a virtuoso url and prints results.
       */
      public static void main(String[] args) {

            String url;
            int numresults;
            if(args.length == 0)
                url = "jdbc:virtuoso://localhost:1111";
            else
                url = args[0];

/*                STEP 1                    */
            VirtGraph set = new VirtGraph (url, "dba", "dba");



/*          Select all data in virtuoso   */
            Query sparql = QueryFactory.create("PREFIX bsbm:
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/vocabulary/> "+
"PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> "+
"PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#> "+
"SELECT DISTINCT ?product ?label "+
"FROM <http://dataset_1000.com> "+
"WHERE { "+
"?product rdfs:label ?label. "+
"?product rdf:type <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductType2>. "+
"?product <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/productFeature> <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature12>."+
"?product bsbm:productFeature <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/instances/ProductFeature20> . "+
"?product bsbm:productPropertyNumeric1 ?value1 . "+
"FILTER (?value1 > 348)} "+
"ORDER BY ?label "+
"LIMIT 10");
```

```
VirtuosoQueryExecution vqe = VirtuosoQueryExecutionFactory.create (sparql,
set);
numresults=0;
Query1 ext=new Query1();

long start = System.currentTimeMillis();
ResultSet results = vqe.execSelect();


while (results.hasNext())
{
                  QuerySolution result = results.nextSolution();
                    RDFNode graph = result.get("http://dataset_1000.com");
                  RDFNode s = result.get("product");
                RDFNode p = result.get("label");
                  numresults=numresults+1; System.out.println(" { " + s + " .
" + p +"+}");

}

            String exetime=ext.elapsedTime(start);
            System.out.println("Exec time: " + exetime);
            System.out.println("Num of results :" + numresults);


      }

String elapsedTime(long start)
  {
             long total = System.currentTimeMillis() - start;
             long min = total/60000;
             long msec = total%60000;
             double sec = msec/1000.0;
             String report;
            if (min > 0)
                   {
                  report = min + ":" + sec + " minutes:seconds";
   }
else
   {
      report = sec + " seconds";
   }
return report;

       }

}
```

Loading and indexing in Neo4j

```
import java.io.File;
import java.util.HashMap;
import java.util.Map;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.impl.EmptyBindingSet;
import org.openrdf.query.parser.ParsedQuery;
import org.openrdf.query.parser.QueryParser;
import org.openrdf.query.parser.sparql.SPARQLParserFactory;
import org.openrdf.repository.RepositoryException;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.sail.SailRepositoryConnection;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.Sail;
import org.openrdf.sail.SailConnection;
import org.neo4j.graphdb.GraphDatabaseService;
import com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jGraph;
import com.tinkerpop.blueprints.pgm.oupls.sail.GraphSail;

public class LoadDatasetTest
{
   public static void main( String[] args )

    {
    LoadDatasetTest ext2=new LoadDatasetTest();
    try {
              ext2.loadTriples();
        }
    catch(Exception e)
       {
          e.printStackTrace();
        }

    }



   public void loadTriples() throws Exception
       {
         String dB_DIR = "db6/berlindb";
        Neo4jGraph neo = new Neo4jGraph( dB_DIR );
        neo.setMaxBufferSize( 20000 );
        Sail sail = new GraphSail( neo );
        sail.initialize();
        SailRepositoryConnection connection;
        LoadDatasetTest ext1=new LoadDatasetTest();
        try
        {

         long start = System.currentTimeMillis();
        connection = new SailRepository( sail ).getConnection();
        File file = new File( "berlin_nt_140000.nt" );
        System.out.println( "Loading " + file + ": " );
        connection.add( file, null, RDFFormat.NTRIPLES );
```

```
        connection.close();
         String exetime=ext1.elapsedTime(start);
        System.out.println("Loading time: " + exetime);

        }
        catch ( RepositoryException e1 )
        {
            e1.printStackTrace();
        }
        System.out.print( "Done." );
        sail.shutDown();
        neo.shutdown();
    }


 String elapsedTime(long start)
                {
             long total = System.currentTimeMillis() - start;
             long min = total/60000;
             long msec = total%60000;
             double sec = msec/1000.0;
             String report;
            if (min > 0)
                   {
                 report = min + ":" + sec + " minutes:seconds";
               }
                 else
                     {
                 report = sec + " seconds";
                }
               return report;

              }


}
```

Querying in Neo4j

```
import java.io.File;
import java.util.HashMap;
import java.util.Map;
import org.openrdf.query.BindingSet;
import org.openrdf.query.QueryEvaluationException;
import org.openrdf.query.impl.EmptyBindingSet;
import org.openrdf.query.parser.ParsedQuery;
import org.openrdf.query.parser.QueryParser;
import org.openrdf.query.parser.sparql.SPARQLParserFactory;
import org.openrdf.repository.RepositoryException;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.repository.sail.SailRepositoryConnection;
import org.openrdf.rio.RDFFormat;
import org.openrdf.sail.Sail;
import org.openrdf.sail.SailConnection;
import info.aduna.iteration.CloseableIteration;


import com.tinkerpop.blueprints.pgm.impls.neo4j.Neo4jGraph;
import com.tinkerpop.blueprints.pgm.oupls.sail.GraphSail;

public class QueryDataset_1
{

    public static void main( String[] args )

   {

    QueryDataset_1 ext2=new QueryDataset_1();
    try
    {
         ext2.berlinQuery();
    }
    catch(Exception e)
     {
          e.printStackTrace();
      }

    }

    public void berlinQuery() throws Exception
    {
        String dB_DIR = "target/berlindb";

        Sail sail = new GraphSail( new Neo4jGraph( dB_DIR ) );
        sail.initialize();
        String q1=
        "PREFIX bsbm: <http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/> "
                        + "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> "
                        + "PREFIX rdfs: <http://www.w3.org/2000/01/rdf-
schema#> "
                        + "SELECT DISTINCT ?product ?label "
```

```
                        + "WHERE { "
                        + "?product rdfs:label ?label . "
                        + "?product rdf:type <http://www4.wiwiss.F-
berlin.de/bizer/bsbm/v01/instances/ProductType2> . "
                        + "?product bsbm:productFeature
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature12>.
"
                        + "?product bsbm:productFeature
<http://www4.wiwiss.fu-berlin.de/bizer/bsbm/v01/instances/ProductFeature20> .
"
                        + "?product bsbm:productPropertyNumeric1 ?value1 . "
                        + "FILTER (?value1 > 348)} " + "ORDER BY ?label "
                        + "LIMIT 10 " ;

QueryDataset_1 ext1=new QueryDataset_1();
QueryParser parser = new SPARQLParserFactory().getParser();
ParsedQuery query = null;
CloseableIteration<? extends BindingSet, QueryEvaluationException>
sparqlResults;
SailConnection conn = sail.getConnection();


try
  {

      query = parser.parseQuery( q1,
                        "http://www4.wiwiss.fu-
berlin.de/bizer/bsbm/v01/vocabulary/" );

for(int i=0;i<10;i++)
{
  long start = System.currentTimeMillis();
  sparqlResults = conn.evaluate( query.getTupleExpr(),
  query.getDataset(), new EmptyBindingSet(), false );

    while ( sparqlResults.hasNext() )
     {

        sparqlResults.next();

      }

String querytime=ext1.elapsedTime(start);
System.out.println("Querying time: " + querytime);

  }
}

catch ( Throwable e )
   {
         e.printStackTrace();
   }

        conn.close();
        sail.shutDown();
 }
```

```
String elapsedTime(long start)
     {
            long total = System.currentTimeMillis() - start;
            long min = total/60000;
            long msec = total%60000;
           double sec = msec/1000.0;
            String report;
            if (min > 0)
              {
                  report = min + ":" + sec + " minutes:seconds";
               }
           else
              {
                   report = sec + " seconds";
               }
            return report;

      }

}
```

APPENDIX V

Number of results returned by each query

Query/Triples	40 K	300 K	1 M	1.8 M	25 M	50 M
query 1	2	10	10	10	10	0
query 2	23	15	15	16	21	20
query 3	3	10	10	10	10	20
query 4	0	9	5	5	10	0
query 5	1	3	5	5	5	5
query 6	67	729	2176	3697	51521	102994
query 7	2	1	2	1	4	5
query 8	1	1	2	2	2	1
query 9	21	21	5	6	21	21
query 10	5	1	1	1	0	0
query 11	10	10	10	10	10	10
query 12	8	8	8	8	8	8

REFERENCES

[1] W3C. SPARQL Query Language for RDF. World Wide Web Consortium (W3C). [Online] January 15, 2008 http://www.w3.org/TR/rdf-sparql-query/

[2] Christian Bizer and Andreas Schultz, Berlin SPARQL Benchmark(2011) http://www4.wiwiss.fu- berlin.de/bizer/BerlinSPARQLBenchmark/

[3] Open Jena, SDB, [online] June 16, 2011, http://openjena.org/wiki/SDB

[4] Franz Inc .AllegroGraph 3.3 Introduction, Franz Incorporation,[online] Mach 18, 2011 http://www.franz.com/agraph/support/documentation/3.3/agraph-introduction.html

[5] OpenLink Software, OpenLink Virtuoso Universal Server: Documentation,[online] 1999-2000 http://docs.openlinksw.com/pdf/virtdocs.pdf

[6] Neo4j Wiki, RDF / SPARQL Quickstart Guide, [online] February 26, 2011 http://wiki.neo4j.org/content/RDF_/_SPARQL_Quickstart_Guide

[7] W3C. W3C Semantic Web Activity. World Wide Web Consortium (W3C). [Online] 2010. http://www.w3.org/2001/sw/.

[8] Wikipedia, Tim Berners-Lee. [online] December2, 2011, http://sv.wikipedia.org/wiki/Tim_Berners-Lee

[9] W3C, Literals. World Wide Web Consortium (W3C). [online] October 10, 2003 http://www.w3.org/TR/rdf-concepts/#section-Literals

[10] Wikipedia, Uniform Resource Identifier. [online] December 8, 2011 http://en.wikipedia.org/wiki/Uniform_resource_identifier

[11] W3C, Resource Description Framework (RDF): Concepts and Abstract Syntax, World Wide Web Consortium. [online] February 10, 2004 http://www.w3.org/TR/rdf-concepts/

[12] W3C, Resource Description Framework (RDF): Concepts and Abstract Syntax, World Wide Web Consortium. [online] February 10, 2004 http://www.w3.org/TR/rdf-concepts/#section-triples

[13] Wikipedia, Uniform Resource Locator. [online] December 8, 2011 http://en.wikipedia.org/wiki/Uniform_resource_locator

[14] W3C, RDF/XML Syntax Specification (Revised) World Wide Web Consortium. [Online] February 10, 2004 http://www.w3.org/TR/REC-rdf-syntax/

[15] Wikipedia, Notation 3.[Online] October 5, 2011 http://en.wikipedia.org/wiki/Notation3

[16] Wikipedia, Turtle.[Online] October 5, 2011 http://en.wikipedia.org/wiki/Turtle_(syntax)

[17] Wikipedia, N-Triples.[Online] October 5, 2011 http://en.wikipedia.org/wiki/N-Triples

[18] Lefteris Sidirourgos, Vassilis Christophides,The Validating RDF Parser (VRP). [Online] June 23, 2004 http://139.91.183.30:9090/RDF/VRP/index.html

[19] Wikipedia, Notation 3. [Online] October 5, 2011 http://en.wikipedia.org/wiki/Notation3

[20] Wikipedia, Notation 3. [Online] October 5, 2011 http://en.wikipedia.org/wiki/Notation3

[21] Wikipedia, Turtle(Syntax). [Online] November 27, 2011 http://en.wikipedia.org/wiki/Turtle_(syntax)

[22] Wikipedia, N-Triples. [Online] April 5, 2011 http://en.wikipedia.org/wiki/N-Triples

[23] Wikipedia, Notation 3. [Online] October 5, 2011 http://en.wikipedia.org/wiki/Notation3

[24] Comparison of Triple stores http://www.bioontology.org/wiki/images/6/6a/Triple_Stores.pdf

[25] Jena, TDB.[Online] December 10, 2011 http://openjena.org/TDB/

[26] Mulgara Semantic Store,[Online] December 8, 2011 http://docs.mulgara.org/overview/index.html

[27] LargeTripleStores, Garlik 4Store [Online] August 19, 2011 http://www.w3.org/wiki/LargeTripleStores#Garlik_4store_.2815B.29

[28] Jena – A Semantic Web Framework for Java, OpenJena [online] http://jena.sourceforge.net/documentation.html

[29] Christian Bizer and Andreas Schultz, *The Berlin SPARQL Benchmark.* Freie Universität Berlin, Web-based Systems Group, October 26, 2011

[30] Semantic Web,SPARQL Enpoint [Online]November 26,2011 http://semanticweb.org/wiki/SPARQL_endpoint

[31] Jena, Jena – A Semantic Web Framework for Java [Online] December 8,2011 http://jena.sourceforge.net/

[32] Wikipedia, Java Database Connectivity, [Online] October 31, 2011 http://en.wikipedia.org/wiki/Java_Database_Connectivity

[33] Wikipedia Jena , SDB/Database Layout.[Online] October 30, 2010
http://openjena.org/wiki/SDB/Database_Layouts

[34] Wikipedia Jena, SDB/Loading data .[Online] October 30 2011
http://openjena.org/wiki/SDB/Loading_data

[35] Franz Inc, Semantic Web Technologies. http://www.franz.com/agraph/allegrograph/

[36] Franz Inc, Gruff: A Grapher-Based Triple-Store Browser for AllegroGraph
Franz Incorporation. http://www.franz.com/agraph/gruff/

[37] http://www.franz.com/agraph/allegrograph/agraph_performance_tuning.lhtml

[38] OpenLink Software, RDF Performance Tuning. [Online] December 8, 2011
http://ods.openlinksw.com/wiki/main/Main/VirtRDFPerformanceTuning

[39] OpenLink Software, RDF Index Scheme. [Online] December 8, 2011
http://docs.openlinksw.com/virtuoso/rdfperformancetuning.html#rdfperfrdfscheme

[] OpenLink Software, Virtuoso Functions Guide.[Online] April 1, 2012
http://docs.openlinksw.com/virtuoso/fn_ttlp_mt.html

[40] NoSQLZone, RDF data in Neo4J - the Tinkerpop story

[Online] October 24, 2011 http://java.dzone.com/news/rdf-data-neo4j-tinkerpop-story

[41] Virtuoso Jena provider[online] October 24,2011
http://ods.openlinksw.com/wiki/main/Main/VirtJenaProvider

[42] Tinker pop- open source project, October 24, 2011 http://tinkerpop.com/ .

[43] Neo4j – NoSQL database February 1, 2012 [online] http://neo4j.org/.

[44] Named graphs, Semantic Web Activity, February 1, 2012 http://www.w3.org/2004/03/trix/

[45] WAMP Server Package, February 1, 2012 http://www.wampserver.com/en/