6 Experiments for NL-storing of middle-size and large RDF-datasets

Abstract

In this chapter we will present results from series of experiments which are needed to estimate the storing time of NL-addressing for middle-size and large RDF-datasets.

The experiments for NL-storing of middle-size and large RDF-datasets are aimed to estimate possible further development of NL-ArM. We assume that its "software growth" will be done in the same grade as one of the known systems like Virtuoso, Jena, and Sesame. We will analyze what will be the place of NL-ArM in this environment. Our hypothesis is that NL-addressing will have good performance.

Chapter will start with describing the experimental storing models and algorithm used in this research. Further an estimation of experimental systems will be provided to make different configurations comparable. Special proportionality constants for hardware and software will be proposed. Using proportionality constants, experiments with middle-size and large datasets became comparable.

Experiments will be provided with both real and artificial datasets. Experimental results will be systematized in corresponded tables. For easy reading visualization by histograms will be given.

6.1 Experimental storing model

Our first experiments in this research were to realize a small multi-language dictionary. In this case, NL-storing model is simple because the one-one correspondence "word - definition". The storing models for thesauruses are more complicated due to existing more than one corresponding definitions for a given word. Because of this, we outlined and analyzed the storing model of WordNet thesaurus [WordNet, 2012]. The idea was to use NL-addressing to realize the WordNet lexical database and this way to avoid recompilation of its database after every update. The program used for the experiments was "WordArM" (see Appendix A).

The next step of experiments was storing graphs and ontologies which have one important aspect – the layers which correspond to types of relations between nodes of graph or ontology [Ivanova et al, 2013e]. To make experiments with real data, we have used the WordNet as ontology and its 45 types of relations (given by its files of different types) we have stored as 45 layers. To provide experiments in this case, we used program "OntoArM" (see Appendix A).

The goal of the experiments with different small datasets, like dictionary, thesaurus or ontology, was to discover regularities in the NL-addressing realization. More concretely, two regularities of time for storing by using NL-addressing were discovered:

- It depends on number of elements in the instances;
- It not depends on number of instances in datasets.

The experiments for NL-storing of middle-size and large RDF-datasets have another goal. We are interested to estimate possible further development of NL-ArM. We assume that its "software growth" will be done in the same grade as one of the known systems like Virtuoso, Jena, and Sesame. We will analyze what will be the place of NL-ArM in this environment. Our hypothesis is that NL-addressing will have good performance.

Now, our next step is to provide experiments to use NL-addressing for storing middle-size and large RDF-datasets. We have realized experimental program RDFArM (see Appendix A) for storing RDF-datasets.

Let remember from Chapter 1 that the primary goal of Resource Description Framework (RDF) is to handle *non regular or semi-structured data* [Muys, 2007]. RDF provides a general method to decompose any information into pieces called triples [Briggs, 2012]:

- Each triple is of the form "Subject", "Predicate", "Object";
- Subject and Object are the names for two things in the world. Predicate is the relationship between them;
- Subject, Predicate, Object may be given as URI's (stand-ins for things in the real world);
- Object can additionally be raw text.

The power of RDF relies on the flexibility in representing arbitrary structure without a priori schemas. Each edge in the graph is a single fact, a single statement, similar to the relationship between a single cell in a relational table and its row's primary key. RDF offers the ability to specify concepts and link them together into a graph of data [Faye et al, 2012].

Middle-size RDF-datasets are those which contain from several hundred thousand up to 10 millions of RDF-instances.

Large RDF-datasets may contain 50, 100, or more millions, as well as billions or trillions of RDF-instances.

Unfortunately, due to financial limitations, we have no proper hardware for making comprehensive program experiments with many gigabytes of source data. Because of this, storing of very large RDF structures by NL- addressing is planned as future work. Here we will outline partial experiments with limited quantity from several hundred thousand up to 100 millions of triples or quadruples due to small main and hard disk memory as well as computational possibilities for processing very large datasets both by our program and by other installations.

In the same time, due to constant complexity of NL-addressing, we may extrapolate the results and provide preliminary comparison with published benchmarks of known RDF-stores using corresponded normalizing the results.

➢ *Experimental datasets*

We will provide experiments with middle-size RDF-datasets, based on selected real middle-size datasets from DBpedia's homepages [DBpedia, 2007a; DBpedia, 2007b] and artificial middle-size datasets from Berlin SPARQL Bench Mark (BSBM) [Bizer & Schultz, 2008; Bizer & Schultz, 2009]. Real large datasets are taken from DBpedia's homepages [DBpedia, 2007c], BSBM [Bizer & Schultz, 2009] and Billion Triple Challenge (BTC) 2012 [BTC, 2012]. Artificial large datasets are taken from Berlin SPARQL Bench Mark (BSBM) [Bizer & Schultz, 2009].

The reason to make this choice is that we want to provide experiments with both real and artificial data. The artificial datasets like BSBM [BSBM, 2012] contain standard artificially generated data and it is possible to adapt the software to have best results just for this kind of data. The DBpedia and BTC datasets were crawled using several seed sets collected from multiple real sources. Data in BTC datasets are encoded in N-Quads format.

The N-Quads is a format that extends N-triples with context. Each triple in N-Quad's document can have an optional context value [N-Quads, 2013]:

<subject> <predicate> <object> <context>.

as opposed to N-triples, where each triple has the form:

<subject> <predicate> <object>.

The notion of provenance is essential when integrating data from different sources. Therefore, modern RDF repositories store "subject-predicate-object-context" quadruples, where the context typically denotes the provenance of a given statement. The SPARQL query language can query such RDF datasets or entire collections of RDF graphs [SPARQL, 2013]. The context element is also sometimes used to track a dimension such as time or geographic location.

Applications of N-Quads include:

- Exchange of RDF datasets between RDF repositories, where the fourth element is the URI of the graph that contains each statement;
- Exchange of collections of RDF documents, where the fourth element is the HTTP URI from which the document was originally retrieved;
- Publishing of complex RDF knowledge bases, where the original provenance of each statement has to be kept intact.

N-Quads inherit the practical advantages of N-Triples:

- Simple parsing;
- Succinctness compared to alternatives such as reification or multi-document archives;
- Effective streaming and processing with line-based tools.

Let see a quadruple from BTC extracted from the data set [datahub_data0, 2012]:

<http://nektar.oszk.hu/resource/auth/magyar_irodalom><http://www.w3.org/2004/02/skos/core#narrower><http://nektar.oszk.hu/resource/auth/hungarikum><http://nektar.oszk.hu/data/auth/magyar_irodalom>.

In this quadruple:

<subject>	=	<http://nektar.oszk.hu/resource/auth/magyar_irodalom>
<predicate>	=	<http://www.w3.org/2004/02/skos/core#narrower>
<object>	=	<http://nektar.oszk.hu/resource/auth/hungarikum>
<context>	=	<http://nektar.oszk.hu/data/auth/magyar_irodalom>

➢ ***Storing models***

The storing models we will use are multi-layer. First two models are for N-Triples and the third is for N-Quads format.

In the *first storing model*, values of *Predicates* may be names of the layers (archives), the *Subjects* will be the NL-addresses, and only *Objects* will be saved.

In the *second storing model*, values of *Subjects* and *Predicates* will be the NL-addresses in the same archive, and only *Objects* will be saved using couple (Subject, Relation) as co-ordinates of container where the Object will be saved.

In the *third storing model*, values of *Subjects* and *Predicates* will be the NL-addresses but *Objects* and *Context*s will be saved in different archives using couple (Subject, Relation) as co-ordinates.

6.2 Experimental storing algorithm

All storing models pointed above may be generalized in one common model where *Subjects, Predicates, Objects,* and *Context*s are numbered separately and these numbers are used to construct storing co-ordinates. For triple datasets the elements which contain context have to be omitted.

The experimental storing algorithm is illustrated on Figure 50. Main idea is to use NL-addressing for quick unique numbering of elements of triples/quadruples and after that to use these numbers as co-ordinates for storing information in the archives. In this case we have two kinds of archives (1) archive of counters and (2) archive of values.

In Figure 50 we illustrated storing of RDF – triple

(beer, is, proof that...)

First we assign a number to subject – in this case: “beer”.

The same we do for the relation – in this case: “is”.

And after that we used these numbers as coordinates of the object - in this case: “proof that ...”.

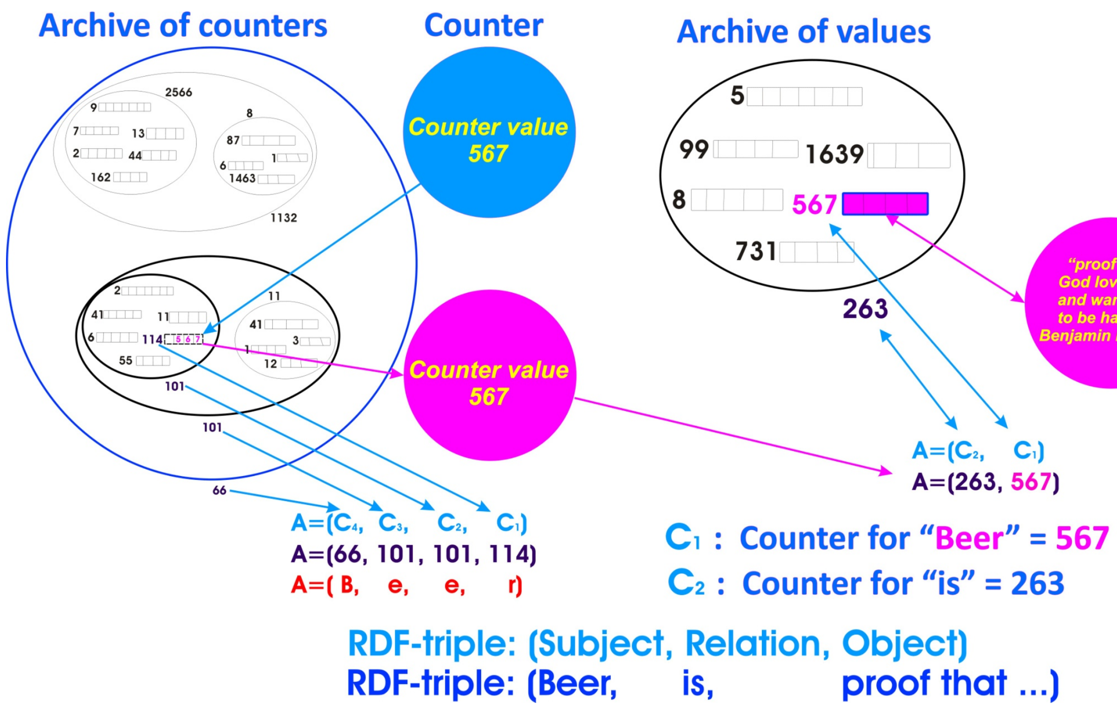

Archive of counters
Counter
Archive of values
Counter value 567
Counter value 567
"proof that God loves us and wants us to be happy." Benjamin Franklin
A=(C₂, C₁)
A=(263, 567)
A=(C₄, C₃, C₂, C₁)
A=(66, 101, 101, 114)
A=(B, e, e, r)
C₁ : Counter for "Beer" = 567
C₂ : Counter for "is" = 263
RDF-triple: (Subject, Relation, Object)
RDF-triple: (Beer, is, proof that ...)

Figure 50. Illustration of the experimental storing algorithm

➢ ***Common storing algorithm based on NL-addressing***

1. Read a quadruple from input file.
2. Assign unique numbers to the <subject>, <predicate>, <object>, and <context>, respectively denoted by NS, NP, NO, and NC. The algorithm of this step is given below.
3. Store the structures:

– {NO; NC} in the "object" index archive using the path (NS, NP);
– {NS; NC} in the "subject" index archive using the path (NP, NO);
– {NP; NC} in the "predicate" index archive using the path (NS, NO).

4. Repeat from 1 until there are new quadruples, i.e. till end of file.
5. Stop.

➢ ***Algorithm for assigning unique numbers***

1. A separate counters for the <subject>, <predicate>, <object>, and <context> are used. Counters start from 1.
2. A separate NL-archives for the <subject>, <predicate>, <object>, and <context> are used.

3. In every NL-archive, using the values of respectively <subject>, <predicate>, <object>, and <context> as paths:

   IF no counter value exist at the corresponded path

   THAN

   - Store value of corresponded counter in the container located by the path;
   - Store the content of <subject>, <predicate>, <object>, or <context> respectively in corresponded data archive in hash table 1 (domain 1) using the value of the counter as path;
   - Increment the corresponded counter by 1.

   ELSE assign the existing value of counter as number of NS, NP, NO, and NC, respectively.
4. Return.

➢ *Algorithm for reading based on NL-addressing*

1. Read the request from screen form or file. The request may contain a part of the elements of the quadruple. Missing elements are requested to be found.
2. From every NL-archive, using the values of given respectively <subject>, <predicate>, <object>, or <context> as NL-addresses read the values of corresponded counters NS, NP, NO, or NC.
3. If the corresponded co-ordinate couple exist, read the structures:
   - {NO; NC} from the “object” index archive using path (NS, NP);
   - {NS; NC} from the “subject” index archive using path (NP, NO);
   - {NP; NC} from the “predicate” index archive using path (NS, NO).
4. **IF** all elements of the set {NS, NP, NO, NC} are given:

   THAN using the set {NS, NP, NO, NC} read the quadruple elements (from corresponded data archives).

   ELSE using given values of the elements of the set {NS, NP, NO, NC} scan all possible values of the unknown elements to reconstruct the set {NS, NP, NO, NC}. The result contains all possible quadruples for the requested values.
5. End.

Comment: If any of parameters are not given, i.e. <subject>, <predicate>, <object>, or <context>, as in SPARQL requests, the rest are used as constant addresses and omitted parameters scan all non empty co-ordinates for given position. This way all possible requests like (?S-?P-?O), (S-P-?O), (S-?P-O), (?S-P-O), etc., are covered (S stands for subject, P for property, O for object). For more information about SPARQL see [SPARQL, 2013] as well as short outline of it in the end of Appendix B.

No search indexes are needed and no recompilation of the data base is required after update or adding new information in the data base.

6.3 Estimation of experimental systems

The goal of experiments presented in this chapter is to compare loading times of the experimental program RDFArM with ones of several known systems measured for various datasets. For this purpose, due to different characteristics of the experimental computer configurations and software systems, we need to apply some proportionality constants to make results comparable.

Evaluation, comparison, and selection of modern computer and communication systems are complex decision problem. System evaluation techniques can be either qualitative or quantitative [Dujmovi'c, 1996]:

- Qualitative techniques are usually based on a list of features to be analyzed for each competitive system. The list includes technical characteristics, costs, and other components for evaluation. After a study of proposed systems the evaluator creates for each proposal a list of advantages and a list of disadvantages. The lists summarizing advantages and disadvantages are then intuitively compared and the final ranking of proposed systems is suggested. Such an approach is obviously attractive only when the decision problem is sufficiently simple. In cases with many decision criteria it is difficult to properly intuitively aggregate a number of components affecting the final decision, and it is not possible to precisely identify minor differences between similar proposals. In addition, it is extremely difficult to justify whether a given difference in total cost is commensurate to a corresponding difference in total performance. These difficulties can be reduced by introducing quantitative components in the decision process [Dujmovi'c, 1996].
- The aim of quantitative methods is to make the system evaluation process well structured, relatively simple, and accurate, providing global quantitative indicators which are used to find and to justify the optimum decision [Dujmovi'c, 1996].

For purposes of this research we will use simple evaluation system based on traditional scoring techniques. The basic idea is very simple [Dujmovi'c, 1996]: for a set of evaluated systems we first identify *n* relevant components (performance variables) that are individually evaluated. The results of evaluation are individual normalized scores $E_1, ..., E_n$, where $0 \leq E_i \leq 1$ (or $0 \leq E_i \leq 100\%$). The average score is then

$$E = (E_1 + ... + E_n)/n.$$

If all components are not equally important then we introduce positive normalized weights, which reflect the relative importance of individual components. $W_1,...,W_n$. Usually, $0 \leq W_i \leq 1,$ i = 1, 2,..., n, and $W_1 + ... + W_n = 1$.

The global score is defined as a weighted arithmetic mean:

$$E = W_1E_1 + W_2E_2 + ... W_nE_n,\ 0 \leq E \leq 1.$$

Below we will compare our benchmark hardware configuration with three others. The characteristics we will take in account are Processor, Physical Memory and Hard Disk capacity. We assume that the operating systems and service software are equivalent in all cases. For concrete computer systems used in the experiments we have respectively:

✓ **Configuration K** is our benchmark configuration:

- Processor: Intel Core2 Duo T9550 2.66GHz; CPU Launched: 2009, *Average CPU Mark:* ***1810*** (P_K=1810) [T9550, 2009] http://www.cpubenchmark.net/cpu.php?cpu=Intel+Core2+Duo+T9550+%40+2.66GHz &id=1011;
- Physical Memory: 4.00 GB (M_K=4);
- Hard Disk: 100 GB data partition; 2 GB swap (D_K=100);
- Operating System: 64-bit operating system Windows 7 Ultimate SP1.

Characteristic values of **Configuration K** are: **P_K=1810, M_K=4, D_K=100**.

✓ **Configuration A** is benchmark configuration of [Becker, 2008]:

- Processor: Intel Pentium Dual Core 2.8 GHz; CPU Launched: 2008; Average CPU Mark: 598 (PA =598) [Pentium Dual, 2008];
- Physical Memory: 1 GB (MA =1);
- Hard Disk: 40 GB data partition; 2 GB swap (DA =40);
- Operating System: Ubuntu Linux 7.10 64-bit.

Characteristic values of **Configuration A** are: **P_A =598, M_A =1, D_A =40**.

✓ **Configuration B**: is benchmark configuration of [BSBMv2, 2008] and [BSBMv3, 2009] DELL workstation:

- Processor: Intel Core2Quad Q9450 @ 2.66GHz, CPU Launched:2008, Average CPU Mark: 3791 (PB =3791) [Q9450, 2008];
- Physical Memory: 8GB DDR2 667 (4 x 2GB) (MB =8);
- Hard Disks: 160GB (10,000 rpm) SATA2, 750GB (7,200 rpm) SATA2 (DB = 160 + 750 = 910);
- Operating System: Ubuntu 8.04 64-bit, Kernel Linux 2.6.24-16-generic; Java Runtime: VM 1.6.0, HotSpot(TM) 64-Bit Server VM (build 10.0-b23); Separate partitions for application data (on 7,200 rpm HDD) and data bases (on 10,000 rpm HDD).

Characteristic values of **Configuration B** are: **P_B =3791, M_B =8, D_B =910**.

✓ **Configuration C** is benchmark configuration used for LDIF [LDIF Benchmarks, 2013; LDIF, 2013]:

- Processor: Intel i7 950, 3.07GHz (quad core); CPU Launched: 2009, *Average CPU Mark:* ***5664*** (P_C =5664) [i7 950, 2009];
- Physical Memory: 24GB (M_C =24);
- Hard Disks: 2 × 1.8TB (7,200 rpm) SATA2 (D_C =3600);
- Operating System: Ubuntu 11.04 64-bit, Kernel: 2.6.38-10; Java version: 1.6.0_22.

Characteristic values of **Configuration C** are: **P_C =5664, M_C =24, D_C =3600**.

➢ *Global scores of computer configurations*

Normalized estimation E_P of processors' power will be computed by formula:

$$E_p = \frac{P_i}{P_K}, i = A,B,C$$

where P_j, j=K,A,B,C is the processor's average CPU mark.

We assume that the processors' power is very important and because of this we will use processors weight as 0.5, i.e.

$$W_P = 0.5.$$

Normalized estimation E_M of physical memory will be computed by formula:

$$E_M = \frac{M_i}{M_K}, i = A,B,C$$

where M_j, j=K,A,B,C is the size of main memory in Giga bytes.

We assume that main memory is more important than hard disk memory and because of this we will use main memory weight as 0.3, i.e.

$$W_M = 0.3.$$

Normalized estimation E_{HD} of hard disk capacity will be computed by formula:

$$E_D = \frac{D_i}{D_K}, i = A,B,C$$

where D_j, j=K,A,B,C is the size of hard disk memory in Giga bytes.

We assume that the hard disk memory weight as 0.2, i.e.

$$W_D = 0.2.$$

Formula for computing the global score of computer configuration is defined as a weighted arithmetic mean:

$$E_i = W_P E_P + W_M E_M + W_D E_D$$

or

$$E_i = 0.5 E_P + 0.3 E_M + 0.2 E_D$$

➢ *Global scores of experimental computer configurations*

The global scores of experimental computer configurations are as follow.

✧ Global score $\boldsymbol{E_K}$ of configuration **K** is **1:**

$$P_K=1810;\ \mathbf{EK_P} = 1810/1810 = \mathbf{1}$$

$$M_K=4;\ \mathbf{EK_M} = 4/4 = \mathbf{1}$$

$$D_K=100;\ \mathbf{EK_D} = 100/100 = \mathbf{1}$$

$$\boldsymbol{E_K} = 0.5EK_P + 0.3EK_M + 0.2EK_D = 0.5*1+0.3*1+0.2*1 =$$

$$= 0.5+0.3+0.2 = \mathbf{1}$$

✧ Global score $\boldsymbol{E_A}$ of configuration **A** is **0.32**:

$$P_A=598;\ \mathbf{EA_P} = 598/1810 = \mathbf{0.33}$$

$$M_A=1;\ \mathbf{EA_M} = 1/4 = \mathbf{0.25}$$

$$D_A=40;\ \mathbf{EA_D} = 40/100 = \mathbf{0.40}$$

$$\boldsymbol{E_A} = 0.5EA_P+0.3EA_M+0.2EA_D= 0.5*0.33+0.3*0.25+0.2*0.40 =$$

$$= 0.165+0.075+0.08 = \mathbf{0.32}$$

✧ Global score $\boldsymbol{E_B}$ of configuration **B** is **3.465**:

P_B=3791	$\mathbf{EB_P}$ =	3791/1810	= **2.09**
M_B=8	$\mathbf{EB_M}$ =	8/4	= **2**
D_B=910	$\mathbf{EB_D}$ =	910/100	= **9.1**

$$\boldsymbol{E_B} = 0.5EB_P+0.3EB_M+0.2EB_D = 0.5*2.09+0.3*2+0.2*9.1 =$$
$$= 1.045+0.6+1.82 = \mathbf{3.465}$$

✧ Global score $\boldsymbol{E_C}$ of configuration **C** is **10.565**:

P_C=5664;	$\mathbf{EC_P}$ =	5664/1810	= **3.13**
M_C=24;	$\mathbf{EC_M}$ =	24/4	= **6**
D_C=3600;	$\mathbf{EC_H}$ =	3600/100	= **36**

$$\boldsymbol{E_C} = 0.5EC_P+0.3EC_M+0.2EC_D = 0.5*3.13+0.3*6+0.2*36=$$
$$= 1.565+1.8+7.2 = \mathbf{10.565}$$

✓ *Hardware proportionality constants*

The hardware proportionality constants H_i, i = A, B, C, for normalizing our results to be comparable with results received on other computer configurations are as follow:

K∝A :	H_A =	E_K/E_A =	1 / 0.32	= **3.125**
K∝B :	H_B =	E_K/E_B =	1 / 3.465	= **0.289**
K∝C :	H_C =	E_K/E_C =	1 / 10.565	= **0.095**

➢ *Comparing software systems' performance*

Enhancing the hardware power does not cause linear enhancing of the software performance. To discover the value of growth one has to test both source and enhanced systems running equal or similar software.

In our case we have the same problem. Configurations A, K, B, and C, may be ordered by their Average CPU Marks as well as their General scores. In both cases we need to discover the growth of software performance for different configurations. This is needed because we want to have common basis for comparing our load time with those of other systems which are tested on different computer configurations.

For this purpose we will follow simple algorithm.

Let program system **X** is tested on two computer configurations: **U** and **W**, where **W** is enhanced configuration; and program system **Y** is tested on different computer configuration **V** of the same class and similar characteristics as **U**. We have couples (**X**,**U**), (**X**,**W**), and (**Y**,**V**).

Computer configurations **U** and **W** are not available for testing and all work has to be done on computer configuration **V**.

Computer configurations' global scores are respectively:

$$\mathbf{E_U} = 0.3,\ \mathbf{E_V} = 1, \text{ and } \mathbf{E_W} = 3.$$

X is tested on **U** by dataset **S1** with 200 instances and on **W** with similar dataset **S2** with 250 instances.

Y is tested on configuration **V** by datasets **S1** and **S2**.

Loading times are respectively:

$$\mathbf{L}_{(X,U,S1)}=1000 \text{ sec.}, \mathbf{L}_{(X,W,S2)}=5 \text{ sec.};$$

$$\mathbf{L}_{(Y,V,S1)}=400 \text{ sec.}, \mathbf{L}_{(Y,V,S2)}=500 \text{ sec.}$$

The problem we have to solve is:

"What will be the loading time of system **Y** if it will be run on computer configuration **W** with dataset **S2**?" i.e. $\mathbf{L}_{(Y,W,S2)} = ?$.

Firstly we will illustrate the algorithm and after that we will give it in details.

We have the diagram (Figure 51):

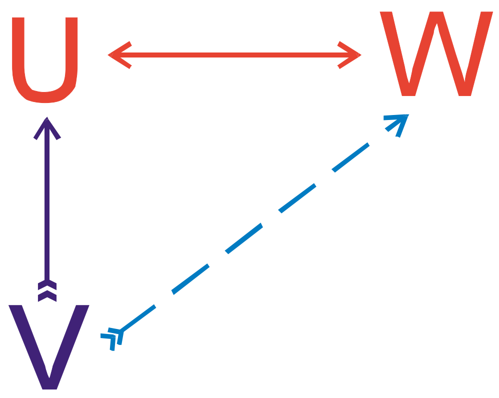


Figure 51. Interrelations between computer configurations

Using published data we may estimate interrelations between computer configurations **U** and **W** as well as between two versions of system **X** run on **U** and **W**. We have to use hardware proportionality constants to make data comparable and to compute the ratio coefficient of software growth by dividing the loading time on **W** by one on **U**.

To make data from experiments on **V** comparable with these on **U** and **W** we assume that **V** and **U** are from the same class of computer power and there is no software growth for a system **Y** in the transition from **V** to **U**. In other words, to estimate interrelations between computer configurations **V** and **U** we need only hardware proportionality constant. After this step we will have data from experiments on **V** transferred for the **U**, i.e. we will have results from system **Y** as if the system **Y** is tested on configuration **U**.

We assume that the possible software growth of system **Y** from computer **U** to **W** is the same as for the system **X**, i.e. we can use the same coefficient for software growth for systems **X** and **Y**. This way we will have comparable data for computer configuration **W**.

Below the algorithm is given in details:

1. Reduce loading time $\mathbf{L_{(X,W,S2)}}$ of program system **X**, run on computer configuration **W** and dataset **S2** with $|\mathbf{S2}|=250$ instances, to loading time $\mathbf{L_{(X,W,S2')}}$ of **X** for hypothetical dataset **S2'** with $|\mathbf{S2'}|=|\mathbf{S1}|=200$ instances, using the formula:

$$\mathbf{L_{(X,W,S2')}} = |\mathbf{S2'}| * (\mathbf{L_{(X,W,S2)}}/ |\mathbf{S2}|) =$$
$$= |\mathbf{S1}| * (\mathbf{L_{(X,W,S2)}}/ |\mathbf{S2}|) = 200*(5/250) = \mathbf{4}$$

2. Compute ratio coefficient of growth $\mathbf{G_{UW}}$ from (**X**,**U**) to (**X**,**W**) by equation:

$$\mathbf{G_{UW}} = \mathbf{L_{(X,U,S1)}}/\mathbf{L_{(X,W,S2')}} = 1000/4 = \mathbf{250}$$

3. Compute loading time $\mathbf{L_{(Y,U,S2)}}$ of system **Y** with dataset **S2** if it is hypothetically ran on configuration **U**, using hardware proportionality constant $\boldsymbol{H_{VU}}$:

$$\mathbf{V \propto U} : \quad \mathbf{H_{VU}} = \quad \mathbf{E_V}/\mathbf{E_U} = \quad 1 / 0.3 \quad = \mathbf{3.33}$$

and formula:

$$\mathbf{L_{(Y,U,S2)}} = \mathbf{H_{VU}}*\mathbf{L_{(Y,V,S2)}} = 3.33*\mathbf{L_{(Y,V,S2)}} = 3.33*500 = \mathbf{1665}$$

4. Compute loading time $\mathbf{L_{(Y,W,S2)}}$ of system **Y** with dataset **S2** if it is hypothetically ran on configuration **W,** using ratio coefficient of growth $\mathbf{G_{UW}}$, hypothetical loading time $\mathbf{L_{(Y,U,S2)}}$, and formula:

$$\mathbf{L_{(Y,W,S2)}} = \mathbf{L_{(Y,U,S2)}}/\mathbf{G_{UW}} = \mathbf{L_{(Y,U,S2)}} / 250 = 1665/250 = \mathbf{6.66}$$

This way we have received comparable value of loading time of system **Y** with system **X** for computer configuration **W**, i.e.

$$\mathbf{L_{(X,W,S2)}}=5 \text{ sec. and } \mathbf{L_{(Y,W,S2)}} = 6.66 \text{ sec.}$$

and we may conclude that system **X** will have a little better loading time than system **Y** if both are run on computer configuration **W** with dataset **S2**.

One may suppose that we may use directly proportionality constant $\boldsymbol{H_{WV}}$:

$$\mathbf{W \propto V} : \quad \boldsymbol{H_{WV}} = \quad \mathbf{E_W}/\mathbf{E_V} = \quad 3 / 1 \quad = \mathbf{3}$$

and to reduce $\mathbf{L_{(Y,V,S2)}}=500$ sec. three times, i.e. 500/3 = 166.66.

This is not correct because the *software growth* is not taken in account.

We have to calculate possible software growth from **V** to **W** again going through **U** and using $\mathbf{G_{UW}}$ to calculate possible $\mathbf{G_{VW}}$. This may be done by using the proportionality constant $\mathbf{H_{VU}}$ because we need to calibrate growth from **U** to **W** by hardware proportionality of **V** and **U**. In other words, to receive value of growth $\mathbf{G_{VW}}$ from **V** to **W** we have to compute:

$$\mathbf{G_{VW}} = \mathbf{G_{UW}}/\mathbf{H_{VU}}$$

Finally:

$$\mathbf{L_{(Y,W,S2)}} = \mathbf{L_{(Y,V,S2)}}/\mathbf{G_{VW}}$$

Let see it for concrete values:

$$\mathbf{G_{UW}} = \mathbf{L_{(X,U,S1)}}/\mathbf{L_{(X,W,S2')}} = 1000/4 = \mathbf{250}$$
$$\mathbf{H_{VU}} = \mathbf{E_V}/\mathbf{E_U} = \quad 1 / 0.3 \quad = \mathbf{3.33}$$
$$\mathbf{G_{VW}} = (\mathbf{G_{UW}}/\mathbf{H_{VU}}) = (250/3.33) = \mathbf{75.07}$$
$$\mathbf{L_{(Y,W,S2)}} = \mathbf{L_{(Y,V,S2)}}/\mathbf{G_{VW}} = 500 / 75.07 = \mathbf{6.66}$$

We received the same result as algorithm above. This proves that we have equivalent approaches.

The algorithm may be presented by a formula:

$$L_{(Y,W,S2)} = R_{YVW} * L_{(Y,V,S2)}$$

where

$$R_{YVW} = \frac{E_v * | S1 | * L_{(X,W,S2)}}{E_U * | S2 | * L_{(X,U,S1)}}$$

i.e.

$$L_{(Y,W,S2)} = \frac{E_v * | S1 | * L_{(X,W,S2)}}{E_U * | S2 | * L_{(X,U,S1)}} * L_{(Y,V,S2)}$$

where:

- **X**, **Y** - program systems;
- **U**, **V**, **W** – computer configurations;
- (**X**,**U**), (**X**,**W**), (**Y**,**V**) – couples “program system – computer configuration”;
- $\mathbf{E_U}$, $\mathbf{E_V}$, $\mathbf{E_W}$ - computer configurations’ global scores;
- **S1**, **S2** – datasets;
- $\mathbf{L_{(X,U,S1)}}$, $\mathbf{L_{(X,W,S2)}}$, $\mathbf{L_{(Y,V,S1)}}$, $\mathbf{L_{(Y,V,S2)}}$, $\mathbf{L_{(Y,W,S2)}}$ - loading times of given program system, computer configuration, and dataset;
- $\mathbf{H_{VU}}$ – computer configurations’ proportionality constant;
- $\mathbf{G_{UW}}$ – ratio coefficient of growth of software system during migration from a computer configuration to enhanced one.

➢ *Experimental environment*

Our experimental environment includes program systems, computer configurations, datasets and experimental data like published benchmark results, different constants, ratio coefficients, etc. The main concrete elements of our experimental environment are:

- Program systems to be compared are:
  - RDFArM;
  - Virtuoso;
  - Jena;
  - Sesame.

  Virtuoso, Jena and Sesame have several variants depending of database used. These variants have different loading times on the same computer configurations. In our comparisons we will take the best result from the all benchmarks on given configuration.
- Computer configurations used for benchmarking are **A**, **K**, **B**, **C**;
- Couples “program system – computer configuration” are:
  - (RDFArM, **K**);
  - (Virtuoso, **A**), (Virtuoso, **B**), (Virtuoso, **C**);
  - (Jena, **A**), (Jena, **B**), (Jena, **C**);
  - (Sesame, **A**), (Sesame, **B**), (Sesame, **C**).
- Computer configurations’ global scores are $\mathbf{E_A}$, $\mathbf{E_K}$, $\mathbf{E_B}$, and $\mathbf{E_C}$;

- Middle-size datasets are:
  - *BSBM 50K;*
  - *homepages-fixed.nt;*
  - *BSBM 250K;*
  - *geocoordinates-fixed.nt;*
  - *BSBM 1M;*
  - *BSBM 5M.*
- Large size datasets are:
  - *infoboxes-fixed.nt;*
  - *BSBM 25M;*
  - *BSBM 100M.*
- Proportionality constant between computer configurations **K** and **A** is $\mathbf{H_{KA}}$;
- Ratio coefficient of growth of software systems during migration from computer configuration A to enhanced ones B and C are $\mathbf{G_{AB}}$ and $\mathbf{G_{AC}}$;
- Corresponded loading times **L** will be presented at the places where they will be used.

➢ *Software proportionality constants*

To provide concrete comparisons of our experimental loading time data, we have to compute $\mathbf{H_{KA}}$, $\mathbf{G_{AB}}$, and $\mathbf{G_{AC}}$.

For purposes of this research it is enough to compute average constants $\mathbf{H_{KA}}$, $\mathbf{G_{AB}}$, and $\mathbf{G_{AC}}$ based on average loading data for all chosen systems. We will use published benchmark results done by Freie Universität Berlin, Web-based Systems Group (BSBM team) and available both as printed publication and free accessible data in the Internet.

✓ *Software proportionality for configurations K, A, and B*

Benchmark results for dataset **S1** (homepages-fixed.nt; 200 036 triples) used for benchmarks on **Configuration A** are published in [Becker, 2008] and reproduced in Table 35.

Table 35. Benchmark results for dataset S1 (homepages-fixed.nt)

system	**loading time in seconds**	**the best time in seconds**
Virtuoso (ogps, pogs, psog, sopg)	1327	1327
Jena SDB MySQL Layout 2 Index	5245	
Jena SDB Postgre SQL Layout 2 Index	3557	3557
Jena SDB Postgre SQL Layout 2 Hash	9681	
Sesame Native (spoc, posc)	2404	2404
	Total average time in seconds:	**2429.333**

Benchmark results for dataset **S2** (*BSBM 250K*; 250 030 triples) used for benchmarks on **Configuration B** are published in [BSBMv2, 2008] and reproduced in Table 36.

Table 36. Benchmark results for dataset S2 (BSBM 250K)

system	loading time in seconds
Virtuoso	33
Jena SDB	24
Sesame	18
Total average time in seconds:	**25**

Due to equal systems and range of their loading times on the same computer configuration, we will use total average times as loading times of virtual system **X**, i.e. $\mathbf{L_{(X,A,S1)}}$ = **2429.333** and $\mathbf{L_{(X,B,S2)}}$ = **25**.

Following our algorithm, we reduce loading time $\mathbf{L_{(X,B,S2)}}$ of virtual system **X**, run on computer configuration **B** and dataset **S2** with |**S2**|=250 030 triples, to loading time $\mathbf{L_{(X,B,S2')}}$ of **X** for hypothetical dataset **S2'** with |**S2'**|=|**S1**|=200 036 instances, using the formula

$$\mathbf{L_{(X,B,S2')}} = |\mathbf{S1}| * (\mathbf{L_{(X,B,S2)}}/ |\mathbf{S2}|) = 200036*(25/250030) = \mathbf{20.00}.$$

We compute ratio coefficient of growth $\mathbf{G_{AB}}$ from (**X**,**A**) to (**X**,**B**) by equation:

$$\mathbf{G_{AB}} = \mathbf{L_{(X,A,S1)}}/\mathbf{L_{(X,B,S2')}} = 2429.333/20 = \mathbf{121.46665}.$$

Hardware proportionality constant $\mathbf{H_{AK}}$ is:

$$\mathbf{A \propto K}: \quad \mathbf{H_{AK}} = \quad \mathbf{E_K/E_A} = \quad 1 / 0.32 \quad = \mathbf{3.125}$$

Really measured **RDFArM** loading time on Configuration **K** for dataset **S2** is **575.069** sec. We compute loading time $\mathbf{L_{(RDFArM,A,S2)}}$ using formula:

$$\mathbf{L_{(RDFArM,A,S2)}} = \mathbf{H_{AK}}*\mathbf{L_{(RDFArM,K,S2)}} = 3.125*575.069 = 1797.09.$$

At the end, we compute loading time $\mathbf{L_{(RDFArM,B,S2)}}$ of system **RDFArM** with dataset **S2** if it is hypothetically run on configuration **B,** using ratio coefficient of growth $\mathbf{G_{AB}}$, hypothetical loading time $\mathbf{L_{(RDFArM,A,S2)}}$, and formula:

$$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,A,S2)}}/\mathbf{G_{AB}} = 1797.09 / 121.46665 = \mathbf{14.796}$$

To **verify** our computations and to show *the easiest way* to find $\mathbf{L_{(RDFArM,B,S2)}}$, we will use our formula

$$L_{(RDFArM,B,S2)} = R_{RDFArM,K,B} * L_{(RDFArM,K,S2)}$$

i.e. we have to compute $\mathbf{R_{RDFArM,K,B}}$ one time and to use it in benchmarks for all datasets. $\mathbf{R_{RDFArM,K,B}}$ may be computed by formula:

$$R_{RDFArM,A,B} = \frac{E_K * | S1 | * L_{(X,B,S2)}}{E_A * | S2 | * L_{(X,A,S1)}}$$

or in linear view:

$$\mathbf{R_{RDFArM,K,B}} = (\mathbf{E_K} * |\mathbf{S1}| * \mathbf{L_{(X,B,S2)}}) / (\mathbf{E_A} * |\mathbf{S2}| * \mathbf{L_{(X,A,S1)}}) =$$
$$= (1 * 200036 * 25) / (0.32 * 250030 * 2429.333) =$$
$$= 5000900 / 194369961.5968 = \mathbf{0.025729}.$$

We compute loading time $\mathbf{L_{(RDFArM,B,S2)}}$ of system **RDFArM** with dataset **S2** if it is hypothetically run on configuration **B,** using ratio coefficient $\mathbf{R_{RDFArM,K,B}}$:

$$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}} = 575.069 * 0.025729 = \mathbf{14.796}.$$

We receive the same result.

✓ ***Software proportionality for configurations K, A, and C***

Software proportionality for configurations **K**, **A**, and **C** will be computed based on the performance of systems Virtuoso and Jena because missing information about Sesame in the benchmark publications.

Benchmark results for dataset **S1** (*infoboxes-fixed.nt*; 15,472,624 triples) used for benchmarks on **Configuration A** are published in [Becker, 2008] and reproduced in Table 37.

Table 37. Benchmark results for dataset S1 (infoboxes-fixed.nt)

system	loading time in seconds	the best time in seconds
Virtuoso	7017	7017
Jena SDB MySQL Layout 2 Index	70851	70851
Jena SDB Postgre SQL Layout 2 Index	73199	
Jena SDB Postgre SQL Layout 2 Hash	734285	
	Total average time:	**38934**

Benchmark results for dataset **S2** (*BSBM 100M*; 100 000 748 triples) used for benchmarks on **Configuration C** are published in [BSBMv6, 2011] and reproduced in Table 38.

Table 38. Benchmark results for dataset S2 (BSBM 100M)

system	loading time in seconds
Virtuoso	6566
Jena TDB	4488
Total average time:	**5527**

Following our algorithm, we reduce loading time $\mathbf{L_{(X,C,S2)}}$ of virtual system **X**, run on computer configuration **C** and dataset **S2** with |**S2**|=100 000 748 triples, to loading time $\mathbf{L_{(X,C,S2')}}$ of **X** for hypothetical dataset **S2'** with |**S2'**|=|**S1**|=15 472 624 instances, using the formula:

$$\mathbf{L_{(X,C,S2')}} = |\mathbf{S1}| * (\mathbf{L_{(X,C,S2)}} / |\mathbf{S2}|) =$$

$$= 15472624*(5527/100000748) = \mathbf{855.166}.$$

We compute ratio coefficient of growth $\mathbf{G_{AC}}$ from (**X**,**A**) to (**X**,**C**) by equation:

$$\mathbf{G_{AC}} = \mathbf{L_{(X,A,S1)}}/\mathbf{L_{(X,C,S2')}} = 38934/855.166 = \mathbf{45.528}.$$

Hardware proportionality constant $\boldsymbol{H_{AK}}$ is:

$$\mathbf{A \propto K}: \quad \mathbf{H_{AK}} = \mathbf{E_K/E_A} = 1 / 0.32 = \mathbf{3.125}.$$

Really measured **RDFArM** loading time on Configuration **K** for dataset **S2** is 43652.528 sec. We compute loading time $\mathbf{L_{(RDFArM,A,S2)}}$ using formula:

$$\mathbf{L_{(RDFArM,A,S2)}} = \mathbf{H_{AK}} * \mathbf{L_{(RDFArM,K,S2)}} = 3.125*43652.528 = \mathbf{136414.15}.$$

At the end, we compute loading time $\mathbf{L_{(RDFArM,C,S2)}}$ of system **RDFArM** with dataset **S2** if it is hypothetically run on configuration **C,** using ratio coefficient of growth $\mathbf{G_{AC}}$, hypothetical loading time $\mathbf{L_{(RDFArM,A,S2)}}$, and formula:

$$\mathbf{L_{(RDFArM,C,S2)}} = \mathbf{L_{(RDFArM,A,S2)}}/\mathbf{G_{AC}} = 136414.15/45.528 = \mathbf{2996.27} \text{ sec.}$$

To **verify** our computations and to show *the easiest way* to find $\mathbf{L_{(RDFArM,C,S2)}}$, we will use our formula

$$L_{(RDFArM,C,S2)} = R_{RDFArM,K,C} * L_{(RDFArM,K,S2)}$$

i.e. we have to compute $\mathbf{R_{RDFArM,K,C}}$ one time and to use it in benchmarks for all datasets. $\mathbf{R_{RDFArM,K,C}}$ may be computed by formula:

$$R_{RDFArM,A,C} = \frac{E_K * | S1 | * L_{(X,C,S2)}}{E_A * | S2 | * L_{(X,A,S1)}}$$

or in linear view:

$$\mathbf{R_{RDFArM,K,C}} = (\mathbf{E_K} * |\mathbf{S1}| * \mathbf{L_{(X,C,S2)}}) / (\mathbf{E_A} * |\mathbf{S2}| * \mathbf{L_{(X,A,S1)}}) =$$

$$= (1 * 15472624 * 5527) / (0.32 * 100000748 * 38934) =$$

$$= 85517192848 / 1245897319242.24 = \mathbf{0.068639}.$$

We compute loading time $\mathbf{L_{(RDFArM,C,S2)}}$ of system **RDFArM** with dataset **S2** if it is hypothetically run on configuration **C,** using ratio coefficient $\mathbf{R_{RDFArM,K,C}}$:

$$\mathbf{L_{(RDFArM,C,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,C}} = 43652.528 * 0.068639 = \mathbf{2996.27}.$$

We receive same result.

✓ *Ratio coefficients*

To compare our results from experiments on computer configuration **K** we will use ratio coefficients:

- For published results received on computer configuration **A**:

$$\mathbf{L_{(RDFArM,A,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{3.125};$$

- For published results received on computer configuration **B**:

$$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{0.025729};$$

- For published results received on computer configuration **C**:

$$\mathbf{L_{(RDFArM,C,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{0.068639}.$$

6.4 Experiments with middle-size datasets

We will compare RDFArM with RDF-stores:

– OpenLink Virtuoso Open-Source Edition 5.0.2 [Virtuoso, 2013];
– Jena SDB Beta 1 on PostgreSQL 8.2.5 and MySQL 5.0.45 [Jena, 2013];
– Sesame 2.0 [Sesame, 2012],

tested by Berlin SPARQL Bench Mark (BSBM) team and connected to it research groups [Becker, 2008; BSBMv2, 2008; BSBMv3, 2009]. More information about latest versions of these systems is given in Appendix B.

We will provide experiments with *middle-size RDF-datasets*, based on selected real datasets from DBpedia [DBpedia, 2007a; DBpedia, 2007b] and artificial datasets created by BSBM Data Generator [BSBM DG, 2013; Bizer & Schultz, 2009].

The real middle-size RDF-datasets which we will use consist of DBpedia's homepages and geocoordinates datasets with minor corrections [Becker, 2008]:

– *Homepages-fixed.nt* (200,036 triples; 24 MB) Based on DBpedia's homepages.nt dated 2007-08-30 [DBpedia, 2007a]. 3 URLs that included line breaks were manually corrected (fixed for DBpedia 3.0);
– *Geocoordinates-fixed.nt* (447,517 triples; 64 MB) Based on DBpedia's geocoordinates.nt dated 2007-08-30 [DBpedia, 2007b]. Decimal data type URI was corrected (DBpedia bug #1817019; resolved);

The RDF stores feature different indexing behaviors: Sesame automatically indexes after each import, while SDB and Virtuoso allow for selective index activation which caouse corresponded limitations or advantages. In order to make load times comparable, the data import by [Becker, 2008] was performed as follows:

– *Homepages-fixed.nt* was imported with indexes enabled;
– *Geocoordinates-fixed.nt* was imported with indexes enabled.

In the case with RDFArM no parameters are needed. The data sets were loaded directly from the source N-triple files.

The artificial middle-size RDF-datasets are generated by BSBM Data Generator [BSBM DG, 2013] and published in N-triple as well as in Turtle format [BSBMv1, 2008; BSBMv2, 2008; BSBMv3, 2009]. We converted Turtle format in N-triple format using “rdf2rdf” program developed by Enrico Minack [Minack, 2010].

We have use four BSBM datasets – 50K, 250K, 1M, and 5M. Details about these datasets are summarized in following Table 39.

Table 39. Details about used artificial middle-size RDF-datasets

Name of RDF-dataset:	**50K**	**250K**	**1M**	**5M**
Exact Total Number of Triples:	**50,116**	**250,030**	**1,000,313**	**5,000,453**
Number of Products	91	666	2,785	9,609
Number of Producers	2	14	60	199
Number of Product Features	580	2,860	4,745	3,307
Number of Product Types	13	55	151	73
Number of Vendors	2	8	34	196
Number of Offers	1,820	13,320	55,700	192,180
Number of Reviewers	116	339	1432	12,351
Number of Reviews	2,275	6,660	27,850	240,225
Total Number of Instances	4,899	23,922	92,757	*458,140*
File Size Turtle (unzipped)	14 MB	22 MB	86 MB	*1,4 GB*

✓ *Loading of BSBM 50K*

RDFArM has loaded all **50116** triples from BSBM 50K for about **113 seconds** (112851 ms) or average time of **2.3 ms** per triple (Figure 52).

Number of Subjects in this dataset was S=4900; number of relations R=40; and number of objects O=50116.

This means that practically we had 40 layers with 4900 NL-locations (containers) which contain 50116 objects. The loading time' results from our experiment and [Bizer & Schultz, 2008] are given in Table 40 and shown on Figure 53.

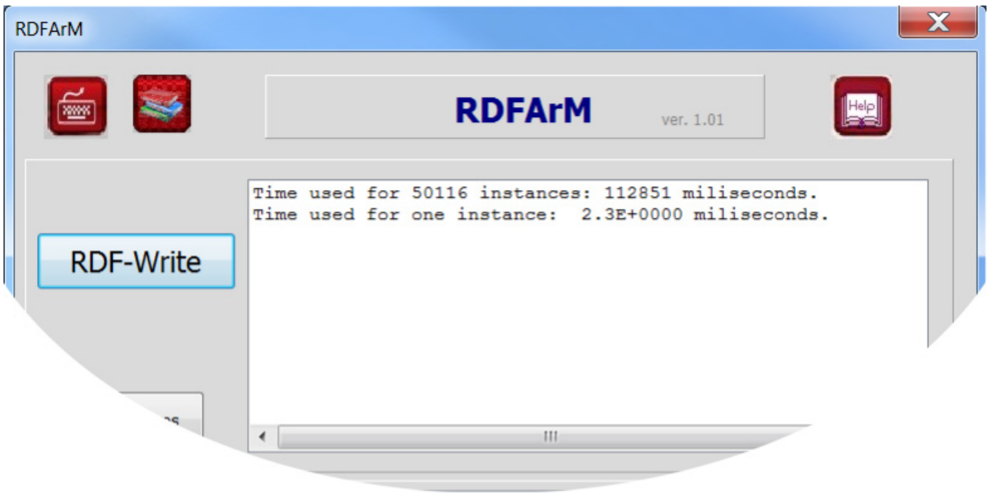
Benchmark configuration used by [Bizer&Schultz, 2008] is **Configuration B**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}}, \text{ and } \mathbf{R_{RDFArM,K,B}} = \mathbf{0.025729};$$

and we compute final loading time as follow: 113 * 0.025729= **2.91 sec**.



```
Time used for 50116 instances: 112851 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 52. Screenshot of the report of RDFArM for BSBM 50K

Table 40. Benchmark results for BSBM 50K

system	loading time in seconds
Sesame	3
Jena SDB	5
Virtuoso	2
RDFArM	**3**

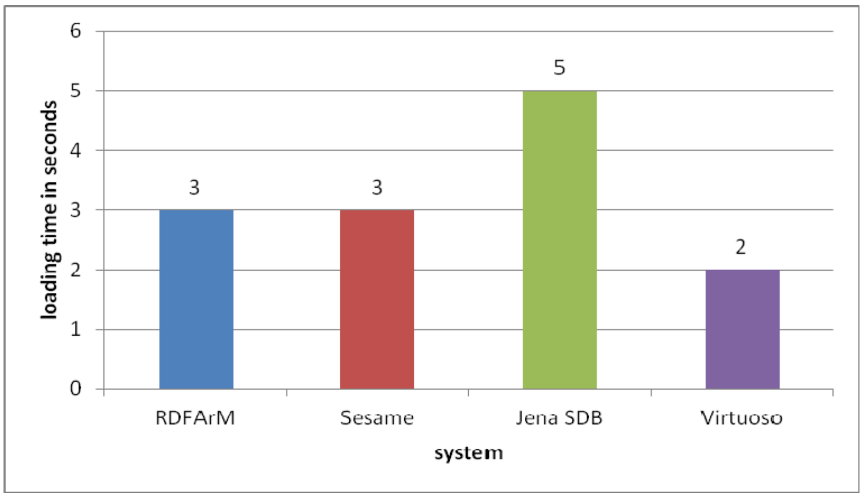
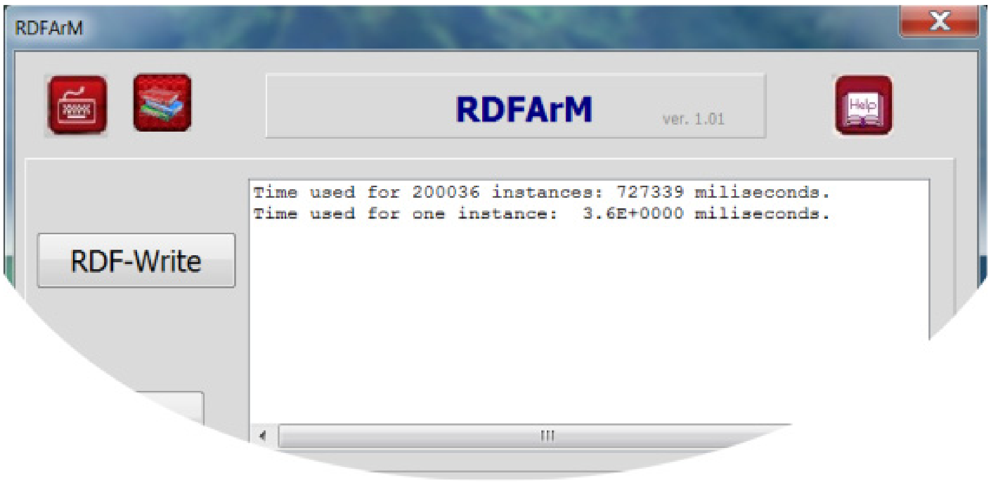


Figure 53. Benchmark results for BSBM 50K

From Table 40 and Figure 53 we may conclude that for **BSBM 50K** Virtuoso has the best time, RDFArM has same loading time as Sesame and 40% better performance than Jena.

✓ ***Loading of homepages-fixed.nt***

RDFArM has loaded all **200036** triples from *homepages-fixed.nt* for about **727 seconds** (727339 ms) or average time of **3.6 ms** per triple (Figure 54).



```
Time used for 200036 instances: 727339 miliseconds.
Time used for one instance:  3.6E+0000 miliseconds.
```

Figure 54. Screenshot of the report of RDFArM for homepages-fixed.nt

More detailed information is given in Table 41. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 200036 and Table 41 contains three rows.

Table 41. RDFArM results for homepages-fixed.nt

part	triples stored	ms for all	ms for one	Subjects	Relations	Objects
1	100000	360955	3.6	100000	1	100000
2	100000	366275	3.7	100000	1	100000
3	36	109	3.0	36	1	36
Total:	**200036**	**727339**	**3.6**	**200036**	**1**	**200036**

Number of Subjects in this dataset was S=200036; number of relations R=1; and number of objects O=200036.

This means that practically we had only one layer with 200036 NL-locations (containers) which contain the same number of objects. The loading time' results from our experiment and [Becker, 2008] are given in Table 42 and Figure 55.

Benchmark configuration used by [Becker, 2008] is **Configuration A**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$$\mathbf{L}_{\mathbf{(RDFArM,A,S2)}} = \mathbf{H}_{\mathbf{AK}}{*}\mathbf{L}_{\mathbf{(RDFArM,K,S2)}},\ \text{where } \mathbf{H}_{\mathbf{AK}} = \mathbf{3.125};$$

and we compute final loading time as follow: 727 x 3.125 = **2271.875** sec.

Table 42. Benchmark results for homepages-fixed.nt

system	loading time in seconds
Virtuoso (ogps, pogs, psog, sopg)	1327
Jena SDB MySQL Layout 2 Index	5245
Jena SDB Postgre SQL Layout 2 Index	3557
Jena SDB Postgre SQL Layout 2 Hash	9681
Sesame Native (spoc, posc)	2404
RDFArM	**2272**

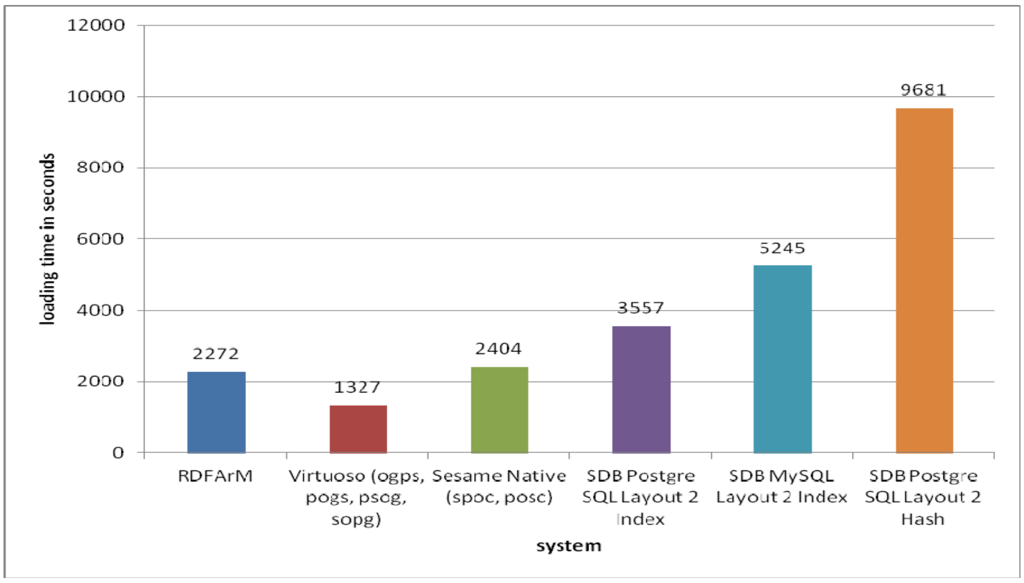


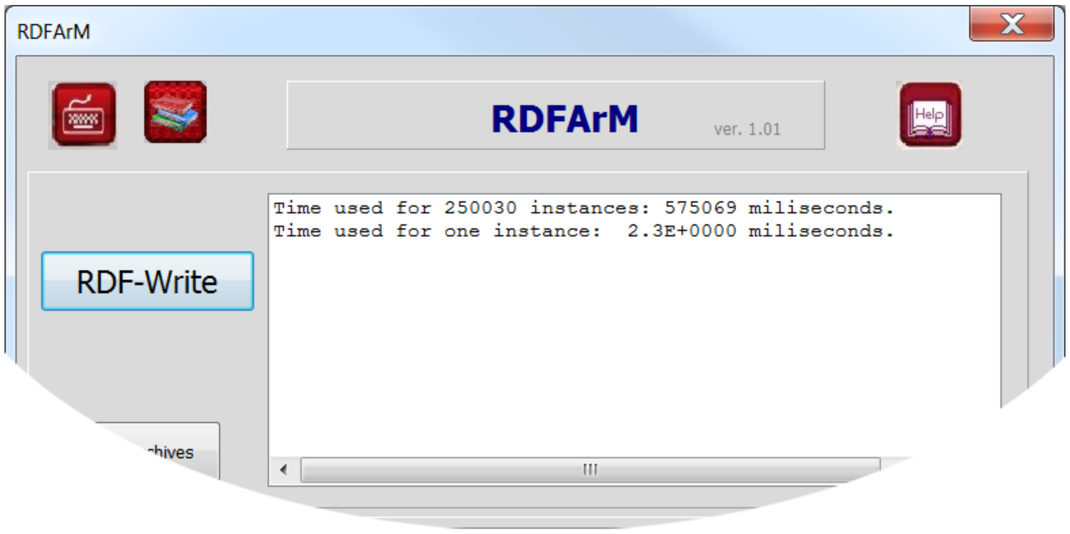
Figure 55. Benchmark results for homepages-fixed.nt

From Table 42 we may conclude that Virtuoso has the best time (about 42% better result than RDFArM); RDFArM has about 5% better time than Sesame and 36% better time than Jena (we take in account only the best results of compared systems, in this case – Jena).

✓ *Loading of BSBM 250K*

RDFArM has loaded all **250030** triples from BSBM 250K for about **575 seconds** (575069 ms) or average time of **2.3 ms** per triple (Figure 56).

More detailed information is given in Table 43. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 250030 and Table 43 contains three rows.



```
Time used for 250030 instances: 575069 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 56. Screenshot of the report of RDFArM for BSBM 250K

Table 43. RDFArM results for BSBM 250K

part	triples stored	ms for all	ms for one	Subjects	Relations	Objects
1	100000	238525	2.4	19854	6	100000
2	100000	228854	2.3	26505	22	100000
3	50030	107690	2.1	14525	22	50030
Total:	**250030**	**575069**	**2.3**	**60884**	**22**	**250030**

Number of Subjects in this dataset was S=60884; number of relations R=22; and number of objects O=250030.

This means that practically we had 22 layers with 60884 NL-locations (containers) which contain 250030 objects. The loading time’ results from our experiment and [BSBMv2, 2008] are given in Table 44 and shown on Figure 57.

Benchmark configuration used by [BSBMv2, 2008] is **Configuration B**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}}$, and $\mathbf{R_{RDFArM,K,B}} = \mathbf{0.025729}$;

and we compute final loading time as follow: 575 x 0.025729= **14.79 sec**.

Table 44. Benchmark results for BSBM 250K

system	**loading time in seconds**
Sesame	19
Jena TDB	13
Virtuoso TS	05
Virtuoso RDF views	09
Virtuoso SQL	09
RDFArM	**14.79**

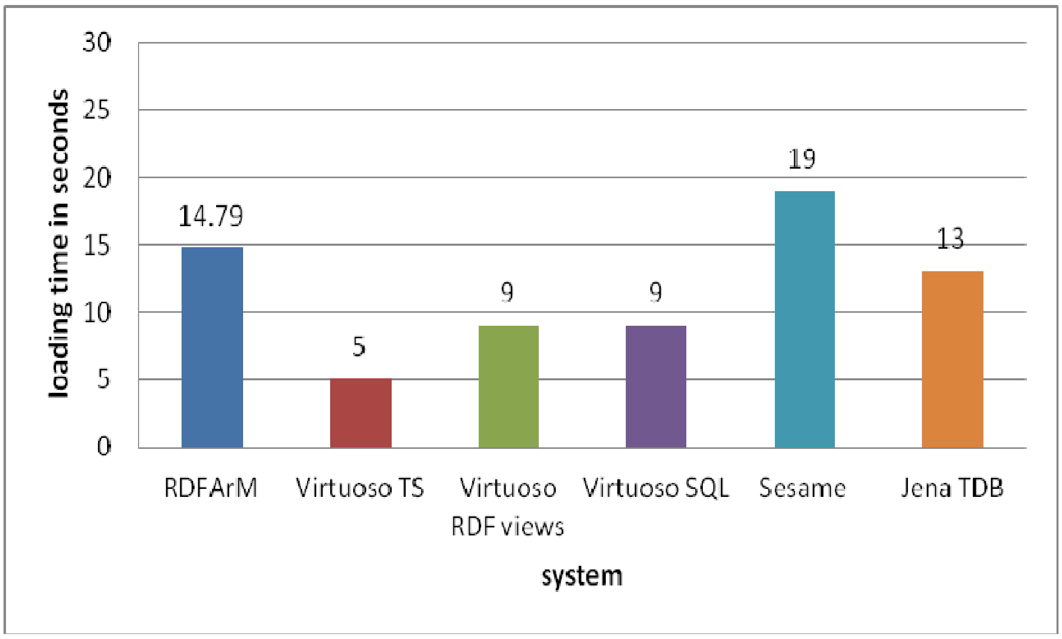


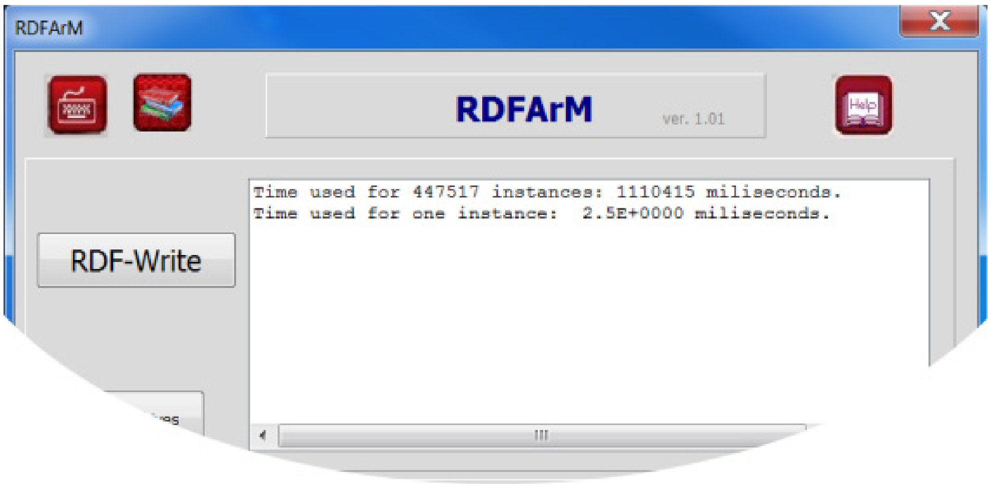
Figure 57. Benchmark results for BSBM 250K

From Table 44 and Figure 57 we may conclude that Virtuoso has 66% and Jena has 12% better performance than RDFArM. RDFArM has 22% better performance than Sesame.

✓ ***Loading of geocoordinates-fixed.nt***

RDFArM has loaded all 447517 triples from geocoordinates-fixed.nt for about 1110 seconds (1110415 ms) or average time of 2.5 ms per triple (Figure 58).

More detailed information is given in Table 45. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 447517 and Table 45 contains five rows.



```
Time used for 447517 instances: 1110415 miliseconds.
Time used for one instance:  2.5E+0000 miliseconds.
```

Figure 58. Screenshot of the report of RDFArM for geocoordinates-fixed.nt

Table 45. RDFArM results for geocoordinates-fixed.nt

part	triples stored	ms for all	ms for one	Subjects	Relations	Objects
1	100000	244453	2.4	34430	6	100000
2	100000	246747	2.5	34909	6	100000
3	100000	245530	2.5	33863	6	100000
4	100000	248198	2.5	33678	6	100000
5	47517	47517	2.6	16095	6	47517
Total:	**447517**	**1110415**	**2.5**	**152975**	**6**	**447517**

Number of Subjects in this dataset was S=152975; number of relations R=6; and number of objects O=447517.

This means that practically we had six layers with 152975 NL-locations (containers) which contain 447517 objects, i.e. some containers in some layers are empty. The loading time' results from our experiment and [Becker, 2008] are given in Table 46 and Figure 59.

Benchmark configuration used by [Becker, 2008] is **Configuration A**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$$\mathbf{L}_{\mathbf{(RDFArM,A,S2)}} = \mathbf{H}_{\mathbf{AK}} * \mathbf{L}_{\mathbf{(RDFArM,K,S2)}}, \text{ where } \mathbf{H}_{\mathbf{AK}} = \mathbf{3.125};$$

and we compute final loading time as follow: 1110 x 3.125= **3468.75 sec**.

Table 46. Benchmark results for geocoordinates-fixed.nt

system	loading time in seconds
Virtuoso (ogps, pogs, psog, sopg)	1235
Jena SDB MySQL Layout 2 Index	6290
Jena SDB Postgre SQL Layout 2 Index	3305
Jena SDB Postgre SQL Layout 2 Hash	9640
Sesame Native (spoc, posc)	2341
RDFArM	**3469**

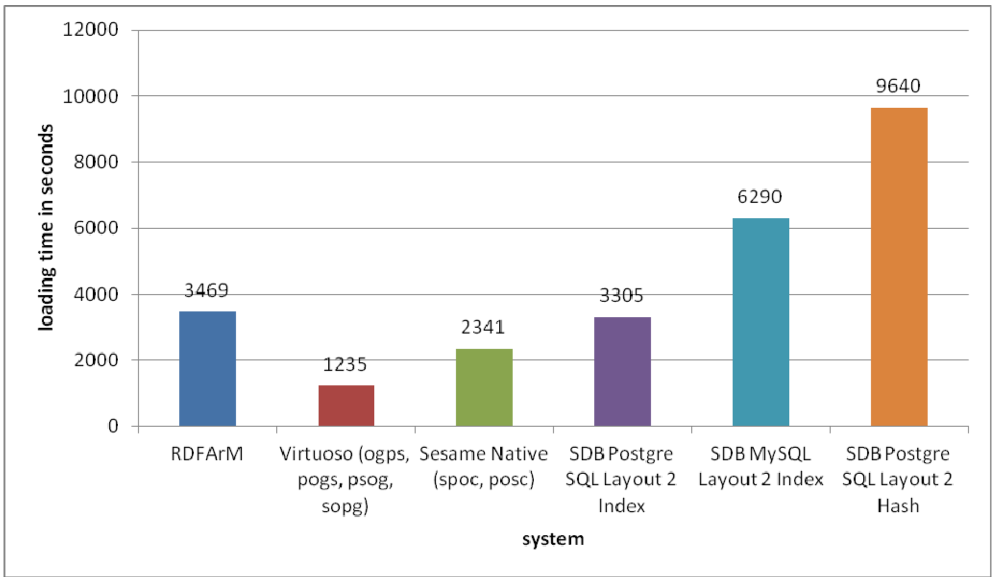
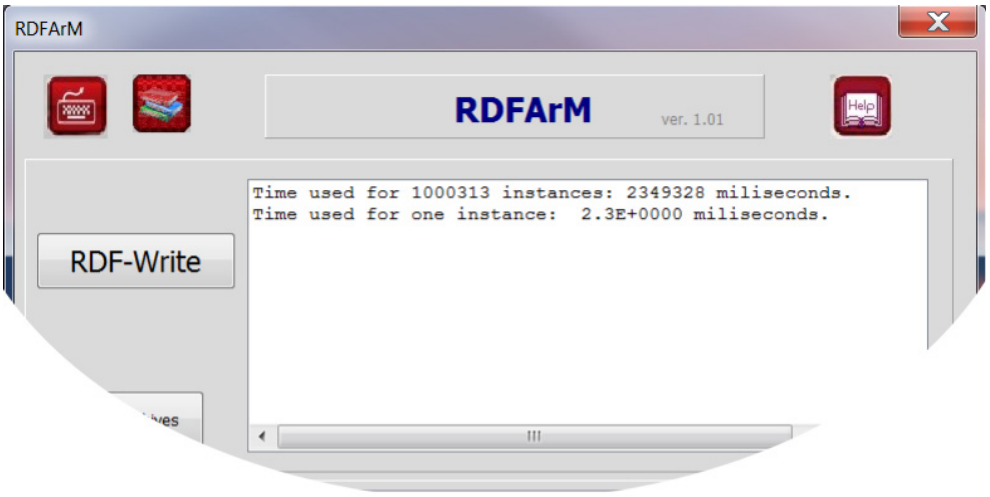


Figure 59. Benchmark results for geocoordinates-fixed.nt

From Table 46 and Figure 59 we may conclude that RDFArM has the worst performance (we take the best time of Jena). Virtuoso has 64%, Sesame has 33%, and Jena has 5% better performance.

✓ ***Loading of BSBM 1M***

RDFArM has loaded all **1000313** triples from BSBM 1M for about **2349 seconds** (2349328 ms) or average time of **2.3 ms** per triple (Figure 60).



```
Time used for 1000313 instances: 2349328 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 60. Screenshot of the report of RDFArM for BSBM 1M

More detailed information is given in Table 47. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 1000313 and Table 47 contains 11 rows. This table has new structure. It contains number of stored triples to corresponded part including it and in separate columns the time for storing the last 100000 triples and average time for one triple from this part.

Table 47. RDFArM results for BSBM 1M

part	triples stored	ms for all	ms for one	ms for last 100000	ms for one	Subjects	Relations	Objects
1	100000	241099	2.4	241099	2.4	6859	22	100000
2	200000	480265	2.4	239166	2.4	14363	29	200000
3	300000	714453	2.4	234188	2.3	24365	29	300000
4	400000	962994	2.4	248541	2.5	34366	29	400000
5	500000	1194344	2.4	231350	2.3	44368	29	500000
6	600000	1423665	2.4	229321	2.3	54370	29	600000

part	triples stored	ms for all	ms for one	ms for last 100000	ms for one	Subjects	Relations	Objects
7	700000	1655420	2.4	231755	2.3	64324	40	700000
8	800000	1892074	2.4	236654	2.4	73799	40	800000
9	900000	2116590	2.4	224516	2.2	83269	40	900000
10	1000000	2348501	2.3	231911	2.3	92729	40	1000000
11	1000313	2349328	2.3	827	2.6	92757	40	1000313

Number of Subjects in this dataset was S=92757; number of relations R=40; and number of objects O=1000313.

This means that practically we had 40 layers with 92757 NL-locations (containers) which contain 1000313 objects. The loading time' results from our experiment and [BSBMv2, 2008; BSBMv3, 2009] are given in Table 48 and shown on Figure 61.

Benchmark configuration used by [BSBMv2, 2008; BSBMv3, 2009] is **Configuration B**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}}$, and $\mathbf{R_{RDFArM,K,B}} = \mathbf{0.025729}$;

and we compute final loading time as follow: 2349 x 0.025729 = **60.437421 sec**.

Table 48. Benchmark results for BSBM 1M

system	**loading time in min:sec**	
	(a) [BSBMv2, 2008]	(b) [BSBMv3, 2009]
Sesame	02:59	03:33
Jena TDB	00:49	00:41
Jena SDB	02:09	-
Virtuoso TS	00:23	00:25
Virtuoso RV	00:34	00:33
Virtuoso SQL	00:34	00:33
RDFArM	**01:00**	**01:00**

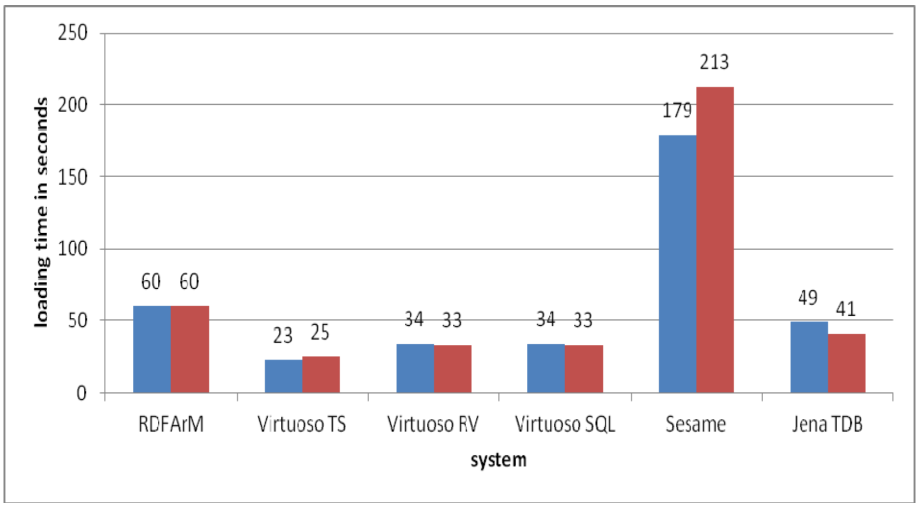
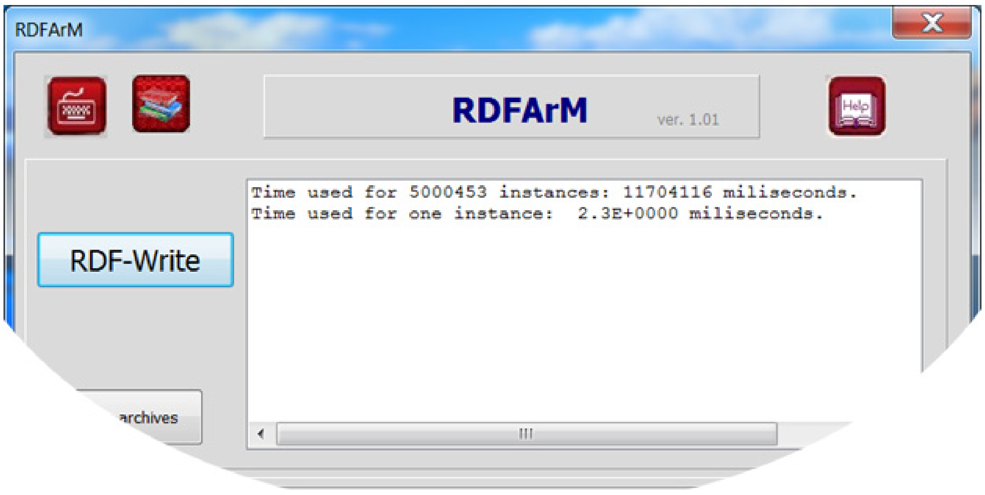


Figure 61. Benchmark results for BSBM 1M

From Table 48 and Figure 61 we may conclude that Virtuoso has 62% and Jena has 32% better performance than RDFArM. RDFArM has 67% better performance than Sesame.

✓ ***Loading of BSBM 5M***

RDFArM has loaded all **5000453** triples from BSBM 5M for about **11704 sec.** (11704116 ms) or average time of **2.3 ms** per triple (Figure 62).



```
Time used for 5000453 instances: 11704116 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 62. Screenshot of the report of RDFArM for BSBM 5M

Number of Subjects in this dataset was S=458142; number of relations R=55; and number of objects O=5000453.

This means that practically we had 55 layers with 458142 NL-locations (containers) which contain 5000453 objects. The loading time' results from our experiment and [Bizer & Schultz, 2008] are given in Table 50 and shown on Figure 63.

More detailed information is given in Table 49. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 5000453 and Table 49 contains 51 rows.

This table contains number of stored triples to corresponded part including it and in separate columns the time for storing the last 100000 triples and average time for one triple from this part.

Table 49. RDFArM results for BSBM 5M

part	triples stored	ms for all	ms for one	ms for last 100000	ms for one	Subjects	Relations	Objects
1	100000	250023	2.5	250023	2.5	5463	22	100000
2	200000	506660	2.5	256637	2.6	7973	22	200000
3	300000	751254	2.5	244594	2.4	10471	22	300000
4	400000	983196	2.5	231942	2.3	12974	22	400000
5	500000	1227104	2.5	243908	2.4	22353	29	500000
6	600000	1468063	2.4	240959	2.4	32357	29	600000
7	700000	1708663	2.4	240600	2.4	42360	29	700000
8	800000	1956034	2.4	247371	2.5	52363	29	800000
9	900000	2190644	2.4	234610	2.3	62366	29	900000
10	1000000	2430043	2.4	239399	2.4	72369	29	1000000
11	1100000	2666041	2.4	235998	2.4	82372	29	1100000
12	1200000	2910230	2.4	244189	2.4	92375	29	1200000
13	1300000	3143529	2.4	233299	2.3	102377	29	1300000
14	1400000	3371618	2.4	228089	2.3	112381	29	1400000
15	1500000	3605136	2.4	233518	2.3	122384	29	1500000
16	1600000	3838139	2.4	233003	2.3	132387	29	1600000
17	1700000	4070830	2.4	232691	2.3	142390	29	1700000
18	1800000	4298155	2.4	227325	2.3	152393	29	1800000

19	1900000	4527367	2.4	229212	2.3	162396	29	1900000
20	2000000	4758030	2.4	230663	2.3	172399	29	2000000
21	2100000	4985698	2.4	227668	2.3	182402	29	2100000
22	2200000	5212742	2.4	227044	2.3	192405	29	2200000
23	2300000	5439692	2.4	226950	2.3	202408	29	2300000
24	2400000	5685347	2.4	245655	2.5	212043	40	2400000
25	2500000	5922328	2.4	236981	2.4	221512	40	2500000
26	2600000	6155331	2.4	233003	2.3	230972	40	2600000
27	2700000	6391610	2.4	236279	2.4	240447	40	2700000
28	2800000	6630417	2.4	238807	2.4	249912	40	2800000
29	2900000	6855511	2.4	225094	2.3	259371	40	2900000
30	3000000	7078545	2.4	223034	2.2	268831	40	3000000
31	3100000	7305979	2.4	227434	2.3	278290	40	3100000
32	3200000	7533928	2.4	227949	2.3	287754	40	3200000
33	3300000	7773608	2.4	239680	2.4	297240	40	3300000
34	3400000	8006782	2.4	233174	2.3	306704	40	3400000
35	3500000	8239629	2.4	232847	2.3	316145	40	3500000
36	3600000	8464536	2.4	224907	2.2	325609	40	3600000
37	3700000	8693202	2.3	228666	2.3	335077	40	3700000
38	3800000	8919248	2.3	226046	2.3	344557	40	3800000
39	3900000	9150254	2.3	231006	2.3	354009	40	3900000
40	4000000	9383912	2.3	233658	2.3	363472	40	4000000
41	4100000	9616120	2.3	232208	2.3	372924	40	4100000
42	4200000	9850090	2.3	233970	2.3	382383	40	4200000
43	4300000	10073842	2.3	223752	2.2	391847	40	4300000
44	4400000	10305832	2.3	231990	2.3	401308	40	4400000
45	4500000	10536619	2.3	230787	2.3	410763	40	4500000
46	4600000	10769997	2.3	233378	2.3	420233	40	4600000
47	4700000	11004030	2.3	234033	2.3	429699	40	4700000

48	4800000	11242836	2.3	238806	2.4	439169	40	4800000
49	4900000	11474107	2.3	231271	2.3	448643	40	4900000
50	5000000	11702852	2.3	228745	2.3	458099	40	5000000
51	5000453	11704116	2.3	1264	2.8	458142	55	5000453

Benchmark configuration used by [Bizer & Schultz, 2008] is **Configuration B**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}}$, and $\mathbf{R_{RDFArM,K,B}} = \mathbf{0.025729}$;

and we compute final loading time as follow: 11704 * 0.025729= **301.13** sec.

Table 50. Benchmark results for BSBM 5M

system	**loading time in seconds**
Sesame	1988
Jena SDB	1053
Virtuoso	609
RDFArM	**301**

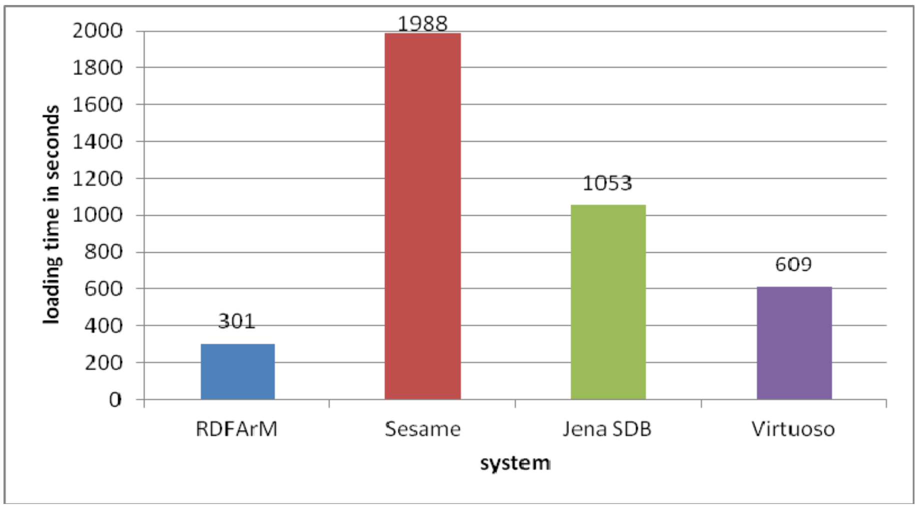


Figure 63. Benchmark results for BSBM 5M

From Table 50 and Figure 63 we may conclude that RDFArM has best loading time (better about 85% than Sesame, 71% than Jena, and 51% than Virtuoso).

6.5 Experiments with large datasets

We provided experiments with real large datasets which were taken from DBpedia's homepages [DBpedia, 2007c] and Billion Triple Challenge (BTC) 2012 [BTC, 2012].

The real dataset from DBpedia's *infoboxes-fixed.nt* (15,472,624 triples; 2.1 GB) is based on DBpedia's infoboxes.nt dated 2007-08-30 [DBpedia, 2007c]. 166 triples from the original set were excluded because they contained excessively large URIs *(> 500 characters)* that caused importing problems with Virtuoso (DBpedia bug #1871653). RDFArM has no such limitation. Infoboxes-fixed.nt was imported with indexes initially disabled in SDB and Virtuoso. Indexes were then activated and the time required for index creation time was factored into the import time. In the case with RDFArM no parameters are needed. The datasets were loaded directly from the source file.

The RDF Stores, tested by [Becker, 2008], are:

- OpenLink Virtuoso Open-Source Edition 5.0.2 [Virtuoso, 2013];
- Jena SDB Beta 1 on PostgreSQL 8.2.5 and MySQL 5.0.45 [Jena, 2013];
- Sesame 2.0 beta 6 [Sesame, 2012].

The RDF stores feature different indexing behaviors: Sesame automatically indexes after each import, while SDB and Virtuoso allow for selective index activation. More information about latest versions of these systems is given in Appendix B.

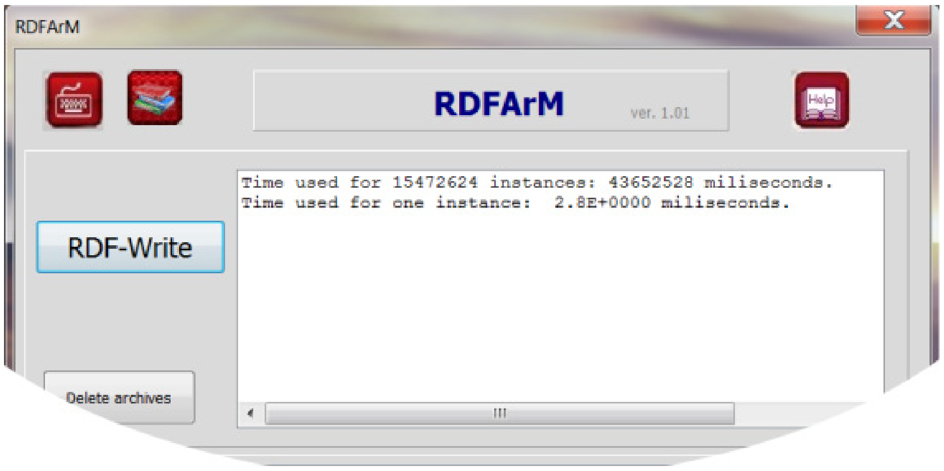
Artificial large datasets are taken from Berlin SPARQL Bench Mark (BSBM) [Bizer & Schultz, 2009; BSBMv3, 2009; BSBMv5, 2009; BSBMv6, 2011]. Details about the benchmark artificial datasets are summarized in the following Table 51:

Table 51. Details about artificial large RDF-datasets

Number of Triples	**25M**	**100M**
Exact Total Number of Triples	**25000244**	**100000112**
Number of Products	70812	284826
Number of Producers	1422	5618
Number of Product Features	23833	47884
Number of Product Types	731	2011
Number of Vendors	722	2854
Number of Offers	1416240	5696520
Number of Reviewers	36249	146054
Number of Reviews	708120	2848260
Total Number of Instances	2258129	9034027
File Size Turtle (unzipped)	*2.1 GB*	8.5 GB

✓ ***Loading of infoboxes-fixed.nt***

RDFArM has loaded all 15 472 624 triples from infoboxes-fixed.nt for about 43652 seconds (43652528 ms) or average time of 2.8 ms per triple (Figure 64).



```
Time used for 15472624 instances: 43652528 miliseconds.
Time used for one instance:  2.8E+0000 miliseconds.
```

Figure 64. Screenshot of the report of RDFArM for infoboxes-fixed.nt

More detailed information is given in Table 70 in Appendix A4. Every row of this table contains data for storing of one hundred thousand triples. Total stored triples were 15,472,624 and Table 70 contains 155 rows.

Number of Subjects in this dataset was S=1354298; number of relations R=56338; and number of objects O=15472624.

This means that practically we had 56338 layers with 1354298 NL-locations (containers) which contain 15472624 objects, i.e. some containers in some layers are empty. The loading time' results from our experiment and [Becker, 2008] are given in Table 52 and Figure 65.

Benchmark configuration used by [Becker, 2008] is **Configuration A**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is
$L_{(RDFArM,A,S2)} = H_{AK}*L_{(RDFArM,K,S2)}$, where $H_{AK} = 3.125$;
and we compute final loading time as follow: 43652 x 3.125= **136412.5** sec.

Table 52. Benchmark results for infoboxes-fixed.nt

system	loading time in seconds
Virtuoso (ogps, pogs, psog, sopg)	7017
Jena SDB MySQL Layout 2 Index	70851
Jena SDB Postgre SQL Layout 2 Index	73199
Jena SDB Postgre SQL Layout 2 Hash	734285
Sesame Native (spoc, posc)	21896
RDFArM	**136412**

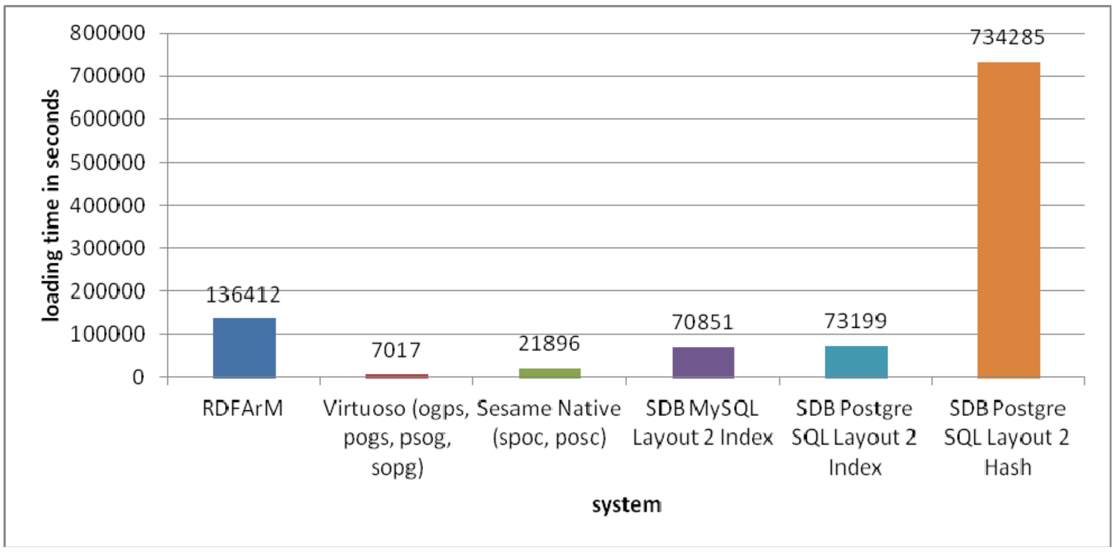


Figure 65. Benchmark results for infoboxes-fixed.nt

From Table 52 and Figure 65 we may conclude that RDFArM has the worst loading time. Virtuoso is 95%, Sesame is 84%, and Jena is 48% better than RDFArM (we take in account only the best results of compared systems).

For large datasets it is very important to support multi-processors' parallel loading of data. RDFArM is developed to support such work. We simulate four processors' configuration by separating dataset on portions of 5 million triples and loading them separately (Table 70).

Table 53 presents final times for different processors.

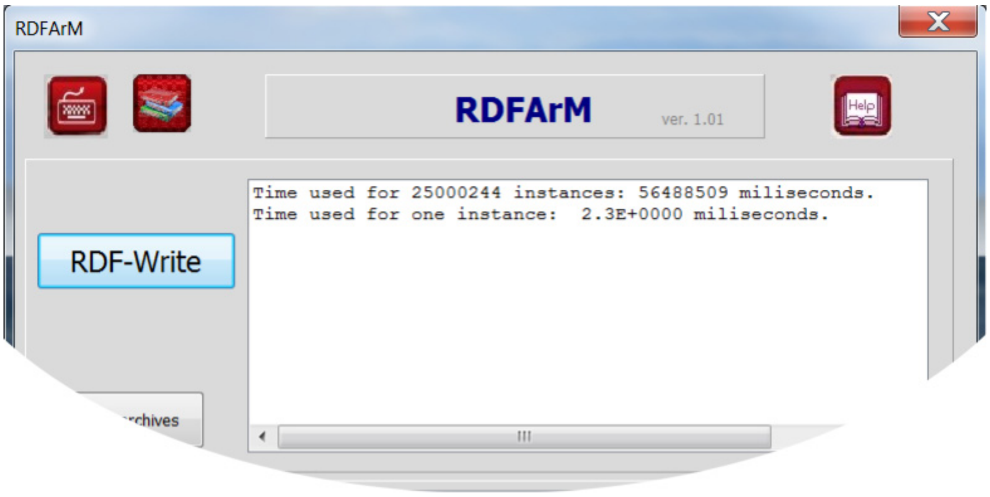
Table 53. Benchmark results for multiprocessor loading of infoboxes-fixed.nt

processor number	triples stored	ms for storing all triples	ms for storing one triple
0	5000000	13394043	2.7
1	5000000	15054986	3.01
2	5000000	14182083	2.8
3	472624	1021416	2.2

As total loading time we assume the largest processor's one, i.e. **15054.986 sec**.

✓ ***Loading of BSBM 25M***

RDFArM has loaded all **25000244** triples from **BSBM 25M** for about **56488** seconds (56488509ms) or average time of **2.3** ms per triple (Figure 66).



```
Time used for 25000244 instances: 56488509 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 66. Screenshot of the report of RDFArM for BSBM 25M

Number of Subjects in this dataset was S=2258132; number of relations R=112; and number of objects O=25000244.

This means that practically we had 112 layers with 2258132 NL-locations (containers) which contain 25000244 objects, i.e. some containers in some layers are empty.

The loading time' results from our experiment and [Bizer & Schultz, 2009; BSBMv3, 2009] are given in Table 54 and Figure 67.

Benchmark configuration used by [Bizer & Schultz, 2009; BSBMv3, 2009] is **Configuration B**. Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$\mathbf{L}_{(\mathbf{RDFArM,B,S2})} = \mathbf{L}_{(\mathbf{RDFArM,K,S2})} * \mathbf{R}_{\mathbf{RDFArM,K,B}}$, and $\mathbf{R}_{\mathbf{RDFArM,K,B}} = \mathbf{0.025729}$.

We compute final loading time as follow: 56488*0.025729= **1453.38** sec.

Table 54. Benchmark results for BSBM 25M

system	**loading time in seconds**
Sesame	44225
Jena TDB	1013
Jena SDB	14678
Virtuoso TS	2364
Virtuoso RV	1035
Virtuoso SQL	1035
RDFArM	**1453**

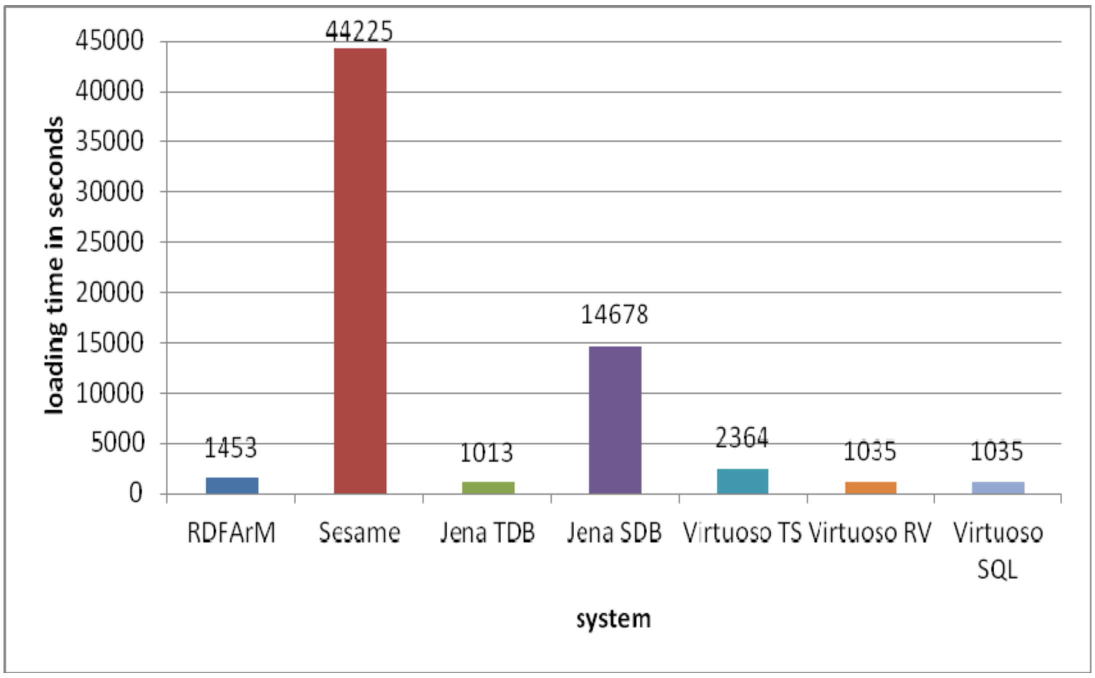


Figure 67. Benchmark results for BSBM 25M

From Table 54 and Figure 67 we may conclude that Jena (with 30%) and Virtuoso (with 29%) are better than RDFArM. RDFArM has 97% better performance than Sesame.

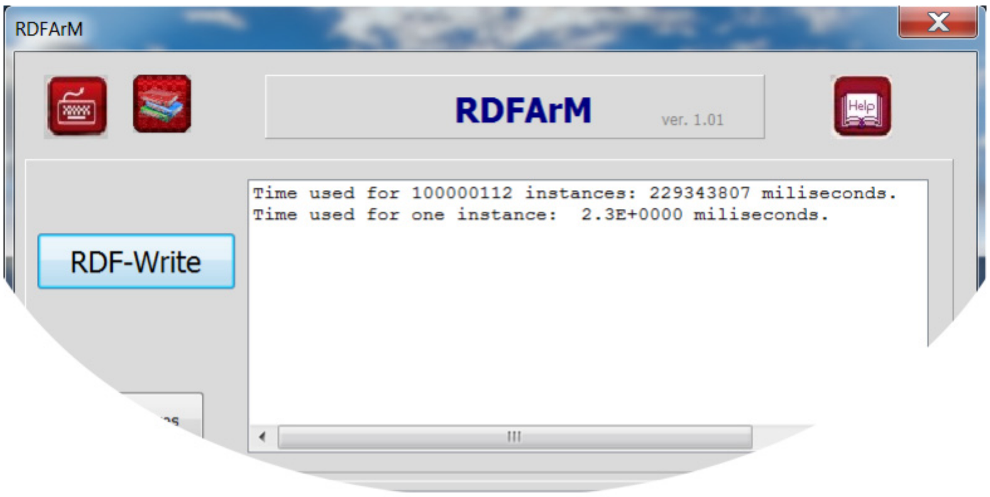
We simulate multi-processors' configuration by separating dataset on portions of 5 million triples and loading them separately. Table 55 presents final times for different processors.

Table 55. Benchmark results for multiprocessors' loading of BSBM 25M

processor number	triples stored	ms for storing all triples	ms for storing one triple
0	5000000	11353862	2.3
1	5000000	11570875	2.3
2	5000000	11529651	2.3
3	5000000	11107771	2.2
4	5000000	10925646	2.2
5	244	704	2.9

✓ ***Loading of BSBM 100M and BSBM 200M***

RDFArM has loaded all **100000112** triples from **BSBM 100M** for about **229344** seconds (229343807 ms) or average time of **2.3** ms per triple (Figure 68).



```
Time used for 100000112 instances: 229343807 miliseconds.
Time used for one instance:  2.3E+0000 miliseconds.
```

Figure 68. Screenshot of the report of RDFArM for BSBM 100M

Number of Subjects in this dataset was S=9034046; number of relations R=341; and number of objects O=100000112.

This means that practically we had 341 layers with 9034046 NL-locations (containers) which contain 100000112 objects, i.e. some containers in some layers contain more than one object. The loading time' results from our experiment and [Bizer & Schultz, 2009; BSBMv3, 2009] are given in Table 56 and Figure 69.

Benchmark configuration used by [Bizer & Schultz, 2009; BSBMv3, 2009] is **Configuration B**. Our benchmark configuration is **Configuration K**.

The loading times proportionality formula is

$\mathbf{L_{(RDFArM,B,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,B}}$, and $\mathbf{R_{RDFArM,K,B}} = \mathbf{0.025729}$.

We compute final loading time as follow:

229344 * 0.025729 = **5900.79** sec.

Table 56. Benchmark results for BSBM 100M

system	loading time in seconds
Sesame	282455
Jena TDB	5654
Jena SDB	139988
Virtuoso TS	28607
Virtuoso RV	3833
Virtuoso SQL	3833
RDFArM	**5901**

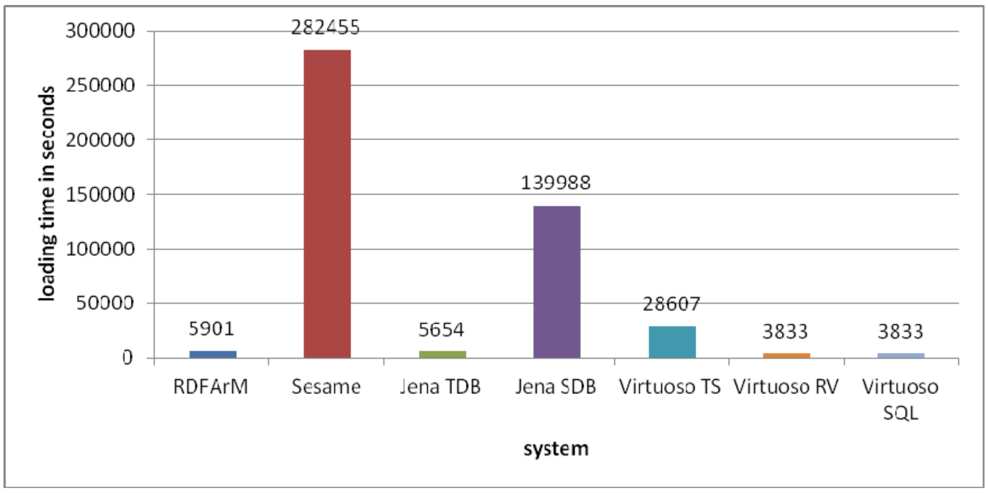


Figure 69. Benchmark results for BSBM 100M

From Table 56 and Figure 69 we may conclude that Virtuoso is 35% better than RDFArM and Jena is 4% better than RDFArM. RDFArM is 98% better than Sesame.

In [BSBMv6, 2011] results received from benchmarks on computer configuration C are published. The N-Triples version of the dataset was used. For Virtuoso, the dataset was split into 100 respectively 200 Turtle files and loaded with the DB.DBA.TTLP function consecutively.

Benchmark configuration used by [BSBMv6, 2011] is **Configuration C**.

Our benchmark configuration is **Configuration K**.

The loading times proportionality formula for **Configuration C** is

$\mathbf{L_{(RDFArM,C,S2)}} = \mathbf{L_{(RDFArM,K,S2)}} * \mathbf{R_{RDFArM,K,C}}$ and $\mathbf{R_{RDFArM,K,C}} = \mathbf{0.068639}$.

We compute RDFArM final loading time for BSBM 100M as follow:

229344 * 0.068639 = **15741.94** sec.

We compute RDFArM final loading time for BSBM 200M as follow:

2 * 229344 * 0.068639 = **31483.88** sec.

The loading time' results from our experiment and [BSBMv6, 2011] are given in Table 57 and Figure 70.

Table 57. Benchmark results for BSBM 100M and 200M on Configuration C

system	**loading time in seconds**	
	100M	**200M**
Jena TDB	4488	9913
Virtuoso	6566	14378
RDFArM	**15742**	**31484**

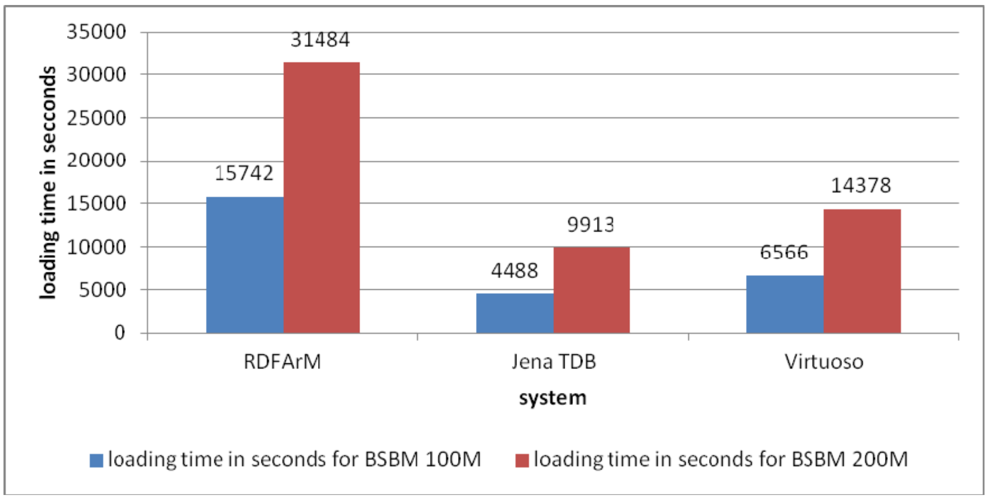


Figure 70. Benchmark results for BSBM 100M and 200M on Configuration C

We have no benchmarks for Sesame. Because of this experiment will not be used for the analysis. From Table 57 and Figure 70 we may conclude that RDFArM has to be improved for big datasets to be comparable to Virtuoso and Jena. This is done in its multi-processors' version RDFArM-MP.

We simulate multi-processors' configuration by separating dataset on portions of 5 million triples and loading them separately. Table 58 presents final times for different processors.

Table 58. Benchmark results for multiprocessors' loading of BSBM 100M

processor number	triples stored	ms for storing all triples	ms for storing one triple
0	5000000	11271727	2.3
1	5000000	11251369	2.3
2	5000000	11514715	2.3
3	5000000	11318091	2.3
4	5000000	11484496	2.3
5	5000000	11571904	2.3
6	5000000	11524854	2.3
7	5000000	11541593	2.3
8	5000000	11547395	2.3
9	5000000	11582902	2.3
10	5000000	11508093	2.3
11	5000000	11461596	2.3

12	5000000	11588535	2.3
13	5000000	11597551	2.3
14	5000000	11565899	2.3
15	5000000	11367450	2.3
16	5000000	11379821	2.3
17	5000000	11337077	2.3
18	5000000	11420647	2.3
19	5000000	11507826	2.3
20	112	266	2.4

As total loading time of multi-processors' configuration we assume the largest processor's time, i.e. 11597.551 sec.

We compute final loading time as follow: 11597.551 * 0.068639 = **796.04** sec.

➢ *Conclusion of chapter 6*

We have presented results from series of experiments which were needed to estimate the storing time of NL-addressing for middle-size and very large RDF-datasets.

We described the experimental storing models and special algorithm for NL-storing RDF instances. Estimation of experimental systems was provided to make different configurations comparable. Special proportionality constants for hardware and software were proposed. Using proportionality constants, experiments with middle-size and large datasets become comparable.

Experiments were provided with both real and artificial datasets. Experimental results were systematized in corresponded tables. For easy reading visualization by histograms was given.

Experimental results will be analyzed in the next chapter.

The goal experiments for NL-storing of middle-size and large RDF-datasets were to estimate possible further development of NL-ArM. We assumed that its "software growth" will be done in the same grade as one of the known systems like Virtuoso, Jena, and Sesame. In the next chapter we will analyze what will be the place of NL-ArM in this environment but already we may see that NL-addressing have good performance and NL-ArM has similar results as Jena and Sesame.