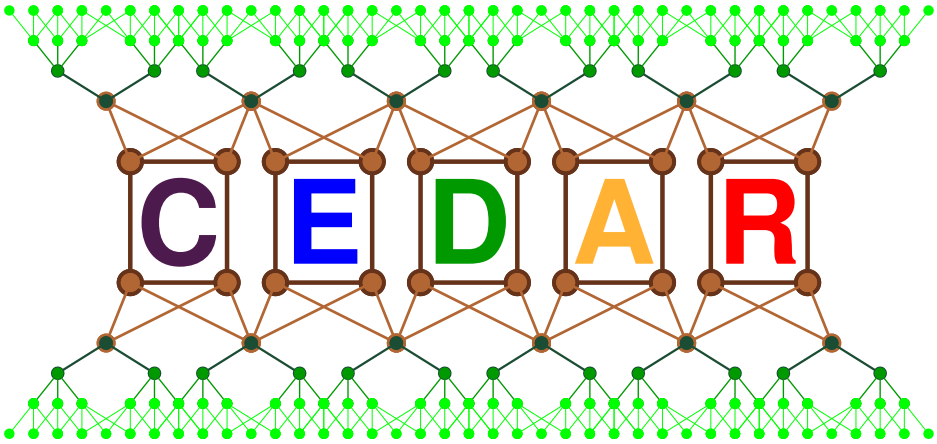


Technical Report Number 5

Experiments with Scalable Triplestores

Hassan Aït-Kaci, Mohand-Saïd Hacid, Rafiqul Haque, Damien Fourure

October 2013

Publication Note

Authors' address:

LIRIS—UFR d'Informatique
Université Claude Bernard Lyon 1
43, boulevard du 11 Novembre 1918
69622 Villeurbanne cedex
France

Phone: +33 (0)4 27 46 57 08

Corresponding author: Rafiqul Haque
akm-rafiqul.haque@univ-lyon1.fr

This work was carried out as part of the $\mathcal{CEDAR}$ Project (Constraint Event-Driven Automated Reasoning) under the *Agence Nationale de la Recherche* (ANR) Chair of Excellence grant N^o **ANR-12-CHEX-0003-01** at the *Université Claude Bernard Lyon 1* (UCBL).

CEDAR **Technical Report Number 5**

Experiments with Scalable Triplestores

Hassan Aït-Kaci, Mohand-Saïd Hacid, Rafiqul Haque, Damien Fourure
cedar@liris.cnrs.fr

October 2013

Abstract

For a triplestore, scalability and high-performance are of the essence. This study examines the scalability and performance of existing triplestores by attempting to reproduce the results reported by their designers. An experiment platform called CedExP was built to test triplestores based on the Hadoop/MapReduce architecture. This initial report focuses on two native triplestores SHARD and HadoopRDF. We ran several experiments both on cloud and non-cloud configurations, increasing the number of nodes (virtual machines), and using various optimization techniques. It was expected that such experiments could reproduce the published results, or even produce better results. Unfortunately, the results could not be reproduced and, in some cases, the results were utterly disappointing. A huge difference was observed between the claimed results and the ones that were produced in this work. The details of all our experiments are presented, analyzed, and discussed. Based on the experience gained in this study and the various observations we could make, we are now planning future work developing our own high-performance triplestore for handling extremely large dataset and improve a few design aspects of Hadoop.

Keywords: Hadoop, MapReduce, RDF, Triplestore, Scalability, Performance

Table of Contents

1 Introduction **1**

2 Experiment Roadmap **2**

3 Preliminaries **3**
3.1 Big Data . . . 3
3.2 Linked Data . . . 3
3.3 Blinked Data . . . 4

4 Tools and Technologies **4**
4.1 Hadoop . . . 4
4.2 MapReduce . . . 5
4.3 Resource Description Framework . . . 6
4.4 SPARQL . . . 6
4.5 The SHARD Triplestore . . . 7
4.6 HadoopRDF . . . 7

5 CedExP—The $\mathcal{CEDAR}$ Experiment Platform **8**
5.1 Overview of CedExP . . . 9
5.2 Configuration of CedExP . . . 10

6 Dataset Generation Experiments **11**
6.1 Analysis . . . 15
6.2 An Open Issue . . . 16

7 N3 Analyzer—An Extension of SHARD Data Generator **17**

8 Experimentation with Triplestores **17**
8.1 Experimentation with the SHARD Triplestore . . . 18
8.1.1 Experimentation with SHARD—Phase I . . . 19
8.1.2 Experimenting with SHARD—Phase II . . . 22
8.1.3 Experimentation with SHARD—Phase III . . . 25
8.1.4 A Comparison of 3-Phase Experiment Results . . . 26
8.1.5 SHARD Reloaded . . . 29
8.1.6 Experiments with SHARD—Phase IV . . . 30
8.2 Experimentation with HadoopRDF . . . 32

9 Expectation *vs.* Reality **33**

10 Conclusion and Future Work **35**

A A Tale of a Safari **36**
A.1 The trip plan . . . 36
A.2 The trek itself . . . 38

B Script for Splitting Dataset **41**

1 Introduction

The Internet has changed the way we see the world and communicate. It has given rise to the notion of "*global village*," which has brought billions of users into the same topological space. Internet users can now store various types of data such as profiles, images, music, business information, and other information in hundreds of thousands of repositories that are hosted in large premises distributed over the Internet, called *data centers*. In addition, the number of Internet users has been increasing exponentially. As a result, people are becoming heavily dependent on various technologies (*e.g.*, smart phone, smart house, *etc.*) which are flooding such repositories with various types of data. Consequently, storage-size requirements in these data centers have been exploding.

In fact, the current state of affairs was predictable, and so is the future. Zikopolous *et al.* [24], for example, estimate that while we are dealing today with terabytes (10^{12} bytes) of data, we are soon to deal with "*brontobytes*" since by 2020 the data size will reach "*zettabytes*," and in the subsequent decade it will reach "*yottabytes*."[1,2] Besides, the statistics speak clearly for themselves: Facebook generates 500 terabytes of data everyday;[3] and Twitter alone generates 7 terabytes of data per day.[4] This clearly indicates that today's predictions will be tomorrow's reality.

The sheer explosion of data size has given rise to the notion of "*Big Data*."[5] All of a sudden, this has become a primary concern for the data-management software service providers. The main challenges these service providers are facing regarding Bid Data are how they should be (1) *managed*, (2) *queryed*, and (3) *analyzed*. To add to the complication, alongside being of enormous size, data are now *linked*.[6]

Under these circumstances, many organizations today are pressed to move from the traditional unconnected relational data silo approach (*e.g.*, isolated relational data storage) to linked-data based storage (*i.e.*, interconnected data over the Internet). With this increasing trend, *Semantic Web* (SW) technologies have gained popularity for handling Linked Data.[7] In particular, the Resource Description Framework (RDF) offers a graph-based model to represent, store, and query SW linked data.[8] As a result, there have been several systems designed specifically for handling large amounts of RDF data in the form of so-called "*triples*."[9] This has then given rise to the notion of "*triplestore*"—a repository containing RDF triples.[10]

This technical report is an initial account of our investigation of existing triplestores. For conducting our experiments, we selected triplestores to be evaluated with respect

[1] The (unofficial) prefixes "*zetta*" and "*yotta*" mean 10^{21} and 10^{24} respectively. Both zettabytes and yottabytes are data sizes in the realm of so-called "*brontobytes*" which range from 10^{15} to 10^{27} bytes.

[2] http://en.wikipedia.org/wiki/Unit_prefix#Unofficial_prefixes

[3] http://gigaom.com/2012/08/22/facebook-is-collecting-your-data-500-terabytes-a-day/

[4] https://blog.twitter.com/2010/measuring-tweets

[5] http://en.wikipedia.org/wiki/Big_data

[6] http://en.wikipedia.org/wiki/Linked_data

[7] http://en.wikipedia.org/wiki/Semantic_Web

[8] http://en.wikipedia.org/wiki/Resource_Description_Framework

[9] http://www.w3.org/TR/rdf-concepts/#section-triples

[10] http://en.wikipedia.org/wiki/Triplestore

to two attributes: (1) *scalability*; and, (2) *performance*—in particular, regarding *query processing time*. The essential motivation for this study has been to try and reproduce the results officially published for triplestores. The experiments reported in this document focus on the SHARD [20] and HadoopRDF triplestores [11]. There are other triplestores that we intend to experiment with as well, such as Jena-Hbase [12], Jena TDB,[11] OpenLink's Virtuoso,[12] and RDF3X,[13] to name a few. But this will be done in a forthcoming study.

This report is organized as follows. In Section 2, we start with an overall summary of the roadmap of our experiments. Then, in Section 3, we give a description of some fundamental concepts. Section 4 describes the tools and technologies that we used in our experiments. In Section 5, we describe CedExp, the experimental platform that we built in order to carry out our study for the $\mathcal{CEDAR}$ project.[14] In Section 6, we present the results of our data generation experiments. In Section 7, we describe *N3 Analyzer*, an extension of the SHARD triplestore, which we built to palliate shortcomings as we experiemented with SHARD. The results of queries are presented and analyzed in Section 8. Section 9 describes the gap between expectations and reality. Finally, a conclusion is drawn and future work is presented in Section 10. We added an appendix where, in Section A, we summarize our experimental trek with a storyline recounting how we proceeded through the details of our study. We thought that this might be instructive for anyone considering investing time and effort in similar pursuits. Also, in Section B, we give the script we used for segmenting our datasets.

2 Experiment Roadmap

The experiment consists of two main parts. In the first part, experiments seek to find a high-performance RDF triplestore that can be used in the second part. The second part of experiments is to try and optimize the performance of the selected triplestore in terms of querying big data over a set of concepts, corresponding clauses, and modifiers, by coupling a T-Box and an A-Box. The second part is our future work and therefore has not been detailed in this report.

The first part of our experiment comprises four phases. In the first three phases the triplestores are tested, the results are structured and analyzed, and compared with the publicly available results produced by these triplestores. The final phase of the experiment is comparing the performance of these triplestores and select the best candidate. The remainder of this report goes through the details of this first part of our experiment.

[11] http://jena.apache.org/documentation/tdb/
[12] http://virtuoso.openlinksw.com/
[13] https://code.google.com/p/rdf3x/
[14] http://cedar.liris.cnrs.fr/

3 Preliminaries

The section describes the basic concepts related to the experiments reported in this paper. The concepts include *Big Data*, *Linked Data*, and *Blinked Data*.[15]

3.1 Big Data

The term "Big Data" applies to information that cannot be processed nor analyzed using traditional database processes or tools [24]. The size of such Big Data is massive, ranging from hundreds of gigabytes to pettabytes and thus storing, searching, sharing, visualizing, and analyzing Big Data is highly challenging. Big Data has four characteristics: *volume*, *velocity*, *variety*, and *veracity* which are proposed in the literature—such as [4], [18], and [24]. The definitions of these characteristics given below are summarized from the cited literature and could be called "*The Four Vs*":

- *Volume*—the size of the dataset.
- *Variety*—different types of data format.
- *Velocity*—speed at which the data is arriving, stored, and retrieved.
- *Veracity*—accuracy, reliability, or certainty. Veracity ratifies that the datasets do not contain inconsistent, ambiguous, duplicate, and deceptive data.

3.2 Linked Data

Linked Data is simply about using the Web to create typed links between data from different sources [3]. It is a means of publishing "web-native" data using standards like HTTP, URIs and RDF [4]. Bizer *et al.* [3] define Linked Data as data published on the Web in such a way that is machine-readable, with meaning explicitly defined, linked to other external data sets, and can in turn be referenced through links from other external datasets. Linked Data uses the RDF format [13].

In order to publish Linked Data on the web, Berners-Lee [1] proposed a set of principles. They are:

- Data must be named with a valid URI.
- A valid URI should associated with HTTP.
- Provide useful information about a thing when its URI dereferenced, leveraging RDF.
- Include RDF statements that link to other URIs so that they can discover related things.

[15] We coined the word "Blinked Data" to designate "Big Linked Data."

3.3 Blinked Data

We introduce a new phrase: *Blinked Data*. The term "Big Data" is generic in that it denotes large volumes of data regardless of their format (relations, triples, semi-structured, *etc.*, ...). Conversely, the term "Linked Data" denotes RDF-triple data whose serialization format can be any of: RDFa (Resource Description Framework attribute), XML-notation RDF, Notation 3 (N3), N-Triples, and Turtle. Linked Data, on the other hand, is not necessarily big. So, in order to refer specifically to "Big RDF-based Data," we combine the term big and linked data and form the new term—"Blinked Data"—to denote big datasets consisting of RDF triples.

4 Tools and Technologies

This section provides a brief description of the technologies include SHARD and HadoopRDF triplestores, Hadoop, MapReduce programming model, SPARQL, and RDF that are used in our experiment.

4.1 Hadoop

The Apache™ Hadoop® [16] is an open source software framework for processing a large dataset that is distributed across a wide range of nodes. Hadoop is an Apache project orginated from Google's MapReduce [14] and the Google File System (GFS) [7]. Hadoop was created by Doug Cutting[16] at the time he was an Yahoo employee and his co-developer Mike Cafarella[17], as an open source project hosted by the Apache Software Foundation [6].

Hadoop is designed to scale up from single servers to thousands of machines, each offering local computation and storage [9]. It has been accredited the future technology for handling big data. The software framework was developed by decomposing it into three different projects: Hadoop Common, Hadoop Distributed File System, and Hadoop MapReduce. These packages are briefly described below.

- **Hadoop Common**: A set of utilities that supports Hadoop's other modules [6].
- **Hadoop Distributed File System** (HDFS): This is the central component of Hadoop. It is the flagship file system of hadoop which was designed to store large files with streaming data access pattern, running on clusters on commodity hardware [23]. HDFS comprises the followings:
  - *Namenode*: It is the only *masternode* in a cluster,
  - *Datanodes*: These are child nodes controlled by the master node,
  - *HDFS Client*: It allows user applications to access to their files,

[16] http://en.wikipedia.org/wiki/Doug_Cutting
[17] http://web.eecs.umich.edu/~michjc/bio.html

  - *CheckpointNode*: This node is also known as secondary namenode. Not to be confused that the secondary namenode replaces the namenode upon its failure. The main task of CheckpointNode is storing the file system metadata entail *namespace image* of the file system and *journal* which is the *wtite-ahead commit log* for the changes to the file system that must be persistent [21]. The CheckpointNode periodically downloads the metadata and journal from the active namenode, combines them , and returns them to namenode.
  - *BackupNode*: It stores the image of the latest journal. It downloads the latest journal from the namenode and persist them into its own (local) storage directory. Notably, namenode may play a role of either BackupNode or CheckpointNode.

  HDFS stores unstructured data into blocks. In HDFS, A input file is decomposed into blocks of a specific size and then these blocks are stored in the datanodes. The size of each block is 64 MB by default however the size can be customized.

- **Hadoop MapReduce**: A YARN-based system for parallel processing of large datasets [6].

Apache Hadoop is the base technology used in our experiment for processing MapReduce jobs.

4.2 MapReduce

MapReduce [14] is a programming model pioneered by the Google Inc.[18]. It has two distributions: Google's MapReduce provided by the Google Inc. and Hadoop MapReduce by the Apache Software Foundation. There is no known difference between these two versions however one simple distinction is Google's MapReduce is *proprietary* whereas the Apache's distribution is *open source*. Notably, we use Hadoop MapReduce for our experiment.

MapReduce is a simple programming model, yet not too simple to express useful program in [23]. MapReduce programming model is very much similar to the traditional forking and merging technique. Typical functional programming languages such as LISP [15] and Haskell [5] implement the MapReduce programming model. Essentially, the notion of this programming model is rooted long before it was implemented in Haddop framework.

MapReduce works by decomposing the processing into *map* and *reduce* phases that are implemented as `map()` and `reduce()` functions. Hadoop framework can execute Map and Reduce functions written in different languages such as Java [17], Python [22], *etc.*, ... During the map phase, the mapper takes the input data from the user in the form of (`Key, Value`) pairs and produces a set of intermediate `(Key, Value)` pairs. The intermediate pairs are essentially the map outputs. These outputs are then shuffled by the MapReduce framework so that the resulting values associated

[18] https://www.google.com/about/

with the same key are grouped together and another set of (`Key, Value`) pairs are assigned to each group. Up to that point, the process produces partitioned answer sets. Following that, the MapReduce framework passes these sets to its "reducer" to merge them to form the final output.

MapReduce is simple enough for small number of inputs however, the performance might be in question if the framework has to process a large number of files especially if the reducers need to merge a large number intermediate outputs.

4.3 Resource Description Framework

The Resource Description Framework (RDF) [13] is the language recommended by the World Wide Web Consortium (W3C) for modeling information about the resources published in the Web (metadata). The resources are identified using qualified resources which is an URI containing '`#`' ending with an optional fragment identifier. The resource in RDF does not necessarily have to be accessible via the Hypertext Transfer Protocol (HTTP). However, a bare URI (without a '`#`') can be Internet-accessible using the `HTTP` "`GET`" method.

The fundamental element of RDF is the *triple* which is also known as an RDF *statement*. A triple is composed of three parts: *subject*, *object*, and *predicate*. A subject and object can be a resource, a literal, or a blank node. A literal is a concrete value and a blank node represents an anonymous resource (*i.e.*, corresponding to no URI nor literal). A subject can be the object of another triple. Contrary to subjects and objects, predicates cannot blank nodes, except in Notation 3 [2]. A predicate (also called *property*) links a subject ro an object. A set of so-linked RDF triples comprises an RDF *graph*.

There are three different types of concepts in RDF. Fundamental concepts include `RDF:Resource`, `RDF:Property`, and `RDF:Statement`. Schema-definition concepts make up yet another type as defined by RDF *vocabularies*. Common ones are utility concepts that include data structures such as `rdf:Bag`, `rdf:Container`, *etc.*, ... Such utility concepts are optional.

4.4 SPARQL

SPARQL [8] stands for "SPARQL Protocol And RDF Query Language" [8]. It is a language for querying and manipulating RDF datasets. A typical SPARQL query consists of a set of triple patterns meant to match RDF triples. A triple pattern is similar to an RDF triple except that the subject, object, and predicate can be a variable [10]. In that manner, SPARQL retrieves desired RDF data from structured as well as semi-structured datasets. A "join" is expressible as several triple patterns sharing a veriable. Since objects and subjects of triples car be URIs, this enables performing complex queries that may involve joins over distributed RDF databases.

The basic constituents of a SPARQL statement include a *prefix*, a *dataset definition clause*, a *result clause*, a *query pattern*, and a *query modifier*. A prefix declaration is declaring the URIs (*e.g.*`PREFIX <http://example.com/resources/>`). A

dataset definition declares which database should be queried. A result clause declaration specifies the expected outcomes. Query patterns are essentially the triples patterns that must be matched by the data being queried. A query modifier dissects, sequences, and rearranges the outcomes.

SPARQL allows the use of aggregation functions such as `SUM`, `MIN`, `MAX`, *etc.*, and aggregation clauses such as `GROUP BY`, `HAVING`, *etc.*, ... Such aggregators are used in for defining complex queries.

4.5 The SHARD Triplestore

SHARD [20] was developed for storing and retrieving RDF data in a distributed environment. Its main concern is scalability, which is a limitation of centralized database management technologies. It is designed as Hadoop/MapReduce repository which caters for building a distributed and parallel environment for storing and querying RDF data. Since Hadoop/MapReduce allows any number of worknodes, this means that scaling up computation to large amounts of data should not be problem. In other words, the number of computational node could in principle be increased without degrading performance. SHARD being based on Hadoop/MapReduce technology, it purports to leverage it for SPARQL query processing.

In SHARD, queries are processed in an iterative manner. This iterative query processing is meant to improve conventional MapReduce functions. In particular, it enables incremental query processing to bind variables while satisfying query constraints.

Each iteration consists of a MapReduce operation for a single query clause. It first maps triple data from a dataset onto the clause matching triples and binding the clause variables and lists all the variable bindings. Then, the subsequent step is to reduce the list of matched triples where duplicate data are deleted.

Following that is the *intermediate query binding step*, where variables from the current clause are bound to values incrementally. Another MapReduce operation is performed in this intermediate step over both triple data and previously bound variables that were saved to disk.

At a certain stage of this iteration (say, at the ith step), all ith variables are identified. The map operation at this stage binds all the variables (if any) that were not seen in the previous clause. In addition, the map operation rearranges the previous results. The reduce operation applies a join over the intermediate results continuously until all clauses are processed and variables satisfying the clauses are bound.

The final step filters bound variable assignments to satisfy the `SELECT` clause of the given SPARQL query. The filtering is done during the map step and duplicates are removed during the reduce step.

4.6 HadoopRDF

HadoopRDF [11] is the other candidate triplestore for the experimentation conducted in this paper. The main focus of HadoopRDF is to optimize Blinked Data queries. The

triplestore uses Hadoop, and makes use in particular of HDFS, to store the RDF triples.

The scalability issue is not given the main priority here as HadoopRDF relies entirely on HDFS for such issues. As for query-processing performance, on the other hand, since HDFS is not concerned with such issues, HadoopRDF provides its own SPARQL query optimization. So, besides storing big RDF datasets using HDFS, it offers an algorithm which determines the best query plan needed to answer a given SPARQL query based on a cost model .

HadoopRDF optimizes querying using in two phases: *Data Preprocessing* and *Query Processing*. The tasks that are performed at preprocessing step include collecting input from the dataset, converting the data into a format that is compatible with HDFS (*viz.*, Notation 3 [2]), carrying out predicate splitting (PS), and performing predicate-object splitting (POS). In the latter phase, the input is selected based on a given query, then a query plan is generated, and the jobs are executed accordingly.

The most interesting features of HadoopRDF are its predicate and predicate-object splitting. Theses two features play a significant role in compressing the dataset without needing any CoDec.[19] They may be viewed as a particular kind of indexing on triples.

The predicate-split function reads a triple and splits according to its predicate. This means that all the subjects and objects with the same predicate will be stored in one same file. For instance, if `WorksFor` is a predicate of n triples, then a single file (say, `WorksFor-pred`) will contain subject/object pairs of all the triples whose predicate is `WorksFor`. On the other hand, the predicate-object split function discriminates triples according to the `rdf:type` denoting the type of the object. This is called Predicate-Object Split of Type (POST). If the object of a RDF triple is a literal, then the literal remains in the file named by the predicate. This split is called POSNT (Predicate-Object Split of Non Type).

In HadoopRDF, upon launching a query, inputs are selected for the query by an *Input Selector*, a component of the MapReduce framework of HadoopRDF. A cost estimator evaluates the costs by reading the selected inputs against the query launched by a user. The plan generator provides a plan for the *Map* and *Reduce* jobs. Finally, the job executor carries out these jobs on the datasets stored in the data nodes of the Hadoop layer of the triplestore.

5 CedExP—The $\mathcal{CEDAR}$ Experiment Platform

This section describes the CedExP platform that is developed for the experiments conducted in this experimental research. CedExP is an extensible experiment platform. The platform has been built by integrating the technologies that have been described in the previous section. This section provides the detail of the platform with a special focus on how these technologies have been bundled in CedExP ecosystem. Figure 1 shows the CedExP architecture.

[19] CoDec stands for "Compression and Decompression."

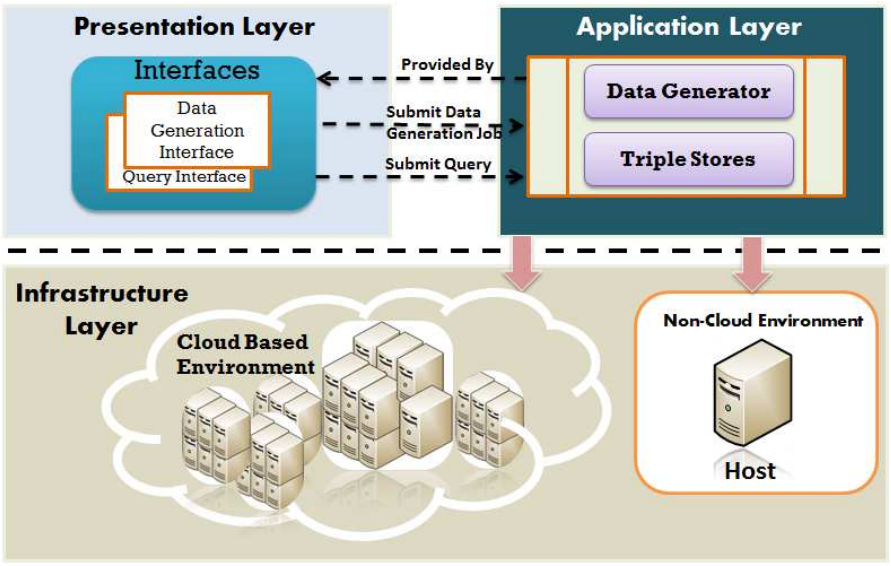


Figure 1: The architecture of the $\mathcal{CEDAR}$ experiment platform

5.1 Overview of CedExP

Since CedExP is a pluggable platform, it facilitates integrating software components whenever it is necessary. It consists of three layers layers: Presentation Layer, Application Layer, and Infrastructure Layer. These layers are briefly described below.

- **Presentation Layer**: This layer provides user interfaces for generating datasets and querying datasets. The interfaces facilitate users to give inputs for dataset generation and querying dataset.
- **Application Layer**: Application layer hosts the applications. It mainly hosts dataset generation applications and query processing applications. CedExP is rather a generic platform which facilitates hosting any generator preferred by the users. Similarly, the users can integrate their preferred query processing applications in CedExP. One important requirement about CedExP platform is application should be built using MapReduce programming model because, the platform is built upon Hadoop technology.
- **Infrastructure Layer**: This is the bottom layer of the CedExP architecture. The CedExP infrastructure can be implemented on *cloud* low cost commodity hardware and *non-cloud* on-premise hardware.

  The non-cloud CedExP infrastructure is relatively more expensive than the cloud infrastructure. Figure 1 shows a non-cloud infrastructure implemented on a single cluster, but that can be increased upon requirement of the users. Notably, the figure shows the current implementation of CedExP architecture.

  The cloud environment was implemented on the LIRIS cloud[20] which is a public

[20]The Infrastructure as a Service (IaaS) provider is the *Laboratoire d'InfoRmatique en Image et Systèmes d'information* (LIRIS).

cloud service provider for the *Université Claude Bernard Lyon 1* (UCBL).[21] We also implemented cloud based environment on PetaSky Cloud which is another public cloud service provider to UCBL projects. CedExP cloud-based infrastructure is a scalable infrastructure by the virtue of Hadoop. The infrastructure can be scaled up-and-down by instantiating and suspending virtual machines (VMs) upon requirement for a job to be processed.

5.2 Configuration of CedExP

This subsection provides the detail of the specifications of technologies that were used in configuring the CedExP environment. Various technologies were used in three layers of CedExP architecture. These technologies are listed below:

- **Application Layer Specification**: CedExP application layer hosts LUBM data generation application for generating datasets. Besides, this layer hosts two triples stores SHARD and HadoopRDF for for processing queries on the datasets. The details of SHARD and HadoopRDF have been provided in Section 4.

  Furthermore, the Eclipse Europa 3.3.2 IDE (Integrated Development Environment) is used for launching the triplestores, modifying the default values of the parameters, and adding new functionality.

- **Presentation Layer Specification**: Both SHARD and HadoopRDF provide interfaces for providing inputs more specifically the *program arguments*. The LUBM data generation application provides interface for providing inputs for generating datasets. It is worth noting that, the LUBM data generator comes with SHARD and HadoopRDF however in CedExP it has been separated since it also can be used in isolation to these applications.

- **Infrastructure Layer Specification**: The infrastructure layer of CedExP is Hadoop based. There is a list of Hadoop vendors distributing open source Hadoop. The list of major vendors includes Cloudera,[22] Hortonworks,[23] and Apache. [24] For the experiments reported in this paper, the open source Apache Hadoop has been used. The Linux operating system was used as the infrastructure.

It is worth noting that the hardware detail is missing here. Since we used different specifications depending on needs, hardware detail is provided in experimentation sections.

While configuring CedExP for SHARD triplestore we encountered a simple configuration error which is explained in the box below along with how it was solved.

[21] http://www.univ-lyon1.fr/
[22] http://www.cloudera.com/content/cloudera/en/home.html
[23] http://hortonworks.com/
[24] http://hadoop.apache.org/

Configuration Error	While configuring the SHARD framework we encountered 28 errors relating the jar files of Cloudera Hadoop. The problem was triggered due to the native Hadoop jar files were missing in the environment. The system threw error messages complaining that `'SHARD' was missing required library` immediately after mounting the framework.
Solution	The problem was solved by adding the required jar files in the framework. This problem was fixed permanently as it was never encountered after the fixation.

6 Dataset Generation Experiments

Data is the central requirement for running queries. In this phase, the goal was to generate a Blinked dataset containing more than a billion of triples serialized in `RDF`. Therefore, the first task performed was generating the university dataset which is large in size. We reached to this number by running the generator several times and by changing the default values of parameters. This section gives the details how the goal was achieved. It is important noting that we used data generation application that comes along as a package with SHARD triplestore application.

We conducted seven tests to reach the target size of dataset. We used machines with different specifications. Table 1 gives the specifications of the machines used in generating the datasets in different sizes. Notably, the data generation tests were performed on a non-cloud environment.

Data generation test 1 The very first test we ran leaving the default values of the parameters (hard coded in the application) unchanged except the value of `UNIV_NUM` parameter which was changed from 1000 to 6000. The program arguments given for the test were `-univ 1` and `-onto [lubm]/univ-bench.owl`, where `-univ 1` denotes the number of the university and `univ-bench.owl` is the script for the ontology.[25] The test generated a tiny dataset 13 MB in size and containing 14 university files with 121,477 RDF triples.

Data generation test 2 The first test was the simplest one. As our goal was generating a big dataset, we changed the parameters again in second test. Additionally, we changed the system specification since we needed a better machine. The following parameters were changed as shown below:

- `UNDER_COURSE_NUM` was 100—changed to 200;
- `GRAD_COURSE_NUM` was 100—changed to 200;

[25] We write `[lubm]` for `http://www.lehigh.edu/7Ezhp2/2004/0401`.

Tests	**System Information**
Test 1	**System Name:** Ced_Exp_Sys_1 Intel(R) Core 2 Duo CPU 2.8 Ghz 32 bit Processor Linux 32 bit 250 GB SATA HDD 2 GB Main Memory
Test 2	**System Name:** Ced_Exp_Sys_2 Intel(R) Core 2 Duo CPU 3 Ghz 32 bit Processor Linux 32 bit 250 GB SATA HDD 4 GB Main Memory
Test 3	**System Name:** Ced_Exp_Sys_2
Test 4	**System Name:** Ced_Exp_Sys_2
Test 5	**System Name:** Ced_Exp_Sys_3 Intel(R) Core I3 CPU 3.20 Ghz 64 bit Processor Linux 64 bit 1 TB SATA HDD 8 GB Main Memory
Test 6	**System Name:** Ced_Exp_Sys_3
Test 7	**System Name:** Ced_Exp_Sys_3

Table 1: Dataset size and system information

- `RESEARCH _NUM` was 30—changed to 60;
- `DEPT_MIN` was 15—changed to 30;
- `DEPT_MAX` was 25—changed to 50;
- `UNIV_NUM` was 6000—changed to 1200.

The test generated second dataset of size 35 MB which contains 41 files containing 339318 RDF triples. Interestingly, although the number of university was reduced in this test, the number of triples were generated more than the previous university.

Data generation test 3 Our aim is to create much larger dataset than it was created in the previous tests. Thus, we continued to carry out the data generation test. Like the previous tests, we changed the values of parameters for this test. The values of parameters were modified as shown below:

- `UNDER_COURSE_NUM` was 200—changed to 1000;
- `GRAD_COURSE_NUM` was 100—changed to 1000;
- `DEPT_MIN` was 30—changed to 150;
- `DEPT_MAX` was 50—changed to 250;
- `RESEARCH _NUM` was 30—changed to 300;
- `UNIV_NUM` was 1200—changed to 60000.

As in the previous test, changing value of university had a very little influence, however the overall performance of the generator was better than the previous test. Almost one and half million triples were generated in this test. The dataset was 181 MB in size containing 216 files.

Data generation test 4 We continued changing parameter until the target size was not produced. The following values of the parameters were used for this test.

- `UNDER_COURSE_NUM` was 1000—changed to 2000;
- `GRAD_COURSE_NUM` was 1000—changed to 2000;
- `DEPT_MIN` was 150—changed to 300;
- `DEPT_MAX` was 250 changed to 500;
- `RESEARCH _NUM` was 300—changed to 600;
- `UNIV_NUM` was 1200—changed to 60000.

We observed that, with these parameters, the size of triples generated in this test is slightly more than double of the previous test. The size of the dataset is 401 MB that contains 337 files containing 3.2 million triples.

Data generation test 5 In this test, the new values were set for the parameters. The new values for this test are given below:

- `UNDER_COURSE_NUM` was 2000—changed to 10000;
- `GRAD_COURSE_NUM` was—changed to 20000;
- `DEPT_MIN` was 150—changed to 1500;
- `DEPT_MAX` was 250—changed to 2500;
- `RESEARCH _NUM` was 600—changed to 3000;
- `UNIV_NUM` was 12000—changed to 60000.

While running this experiment, we encountered a crash due to a "starvation error." The boxes below explains the root cause of the error and how it was solved.

Starvation Error	According to our analysis, this crash happened due to starvation of elements. We experienced it when we increased the value of `DEPT_MIN` and `DEPT_MAX`. We believe SHARD's internal design flaw was the main reason for the element starvation. We diagnosed the problem at the very first step to find the actual cause rather than speculating it. Our first thought was while fetching `tokens` if the system does not find element then SHARD `StringTokenizer` function throws this exception . Nevertheless, we found that each *Process* open `Input` and `Output Stream` (two `OutputStream` (`standard output` and `error output`)and one `InputStream`). However, the `input/input` stream number is limited. Consequently, when the SHARD system generates too many N3 files, the `open` of the *Process* crash.
Solution	We changed SHARD's `waitForEnd()` function (allowing to wait for the end of the all the processes) for closing correctly the `Input/Output` stream open. Then, we caught the exception lunch by a process when it can not open the `Stream` for waiting the end of already lunched process.

The test generated dataset of size 1329 MB containing 1361 files. Unfortunately, the number of triples could not be counted due to another error which was thrown during running the *triple counter*. We tried to resolve this issue however it could not be solved due to unknown reason, the only cause we could assume was the size of data could not be handled by the application. Then, we tried an alternative way to solve this issue. The alternative path was chosen to save the time however it was effective enough to count the triples. The following steps show how the triples was counted:

- **Step 1** : A Linux shell-script was written to count the number of triples contained in a file.
- **Step 2** : The number of triples was multiplied by the number of files. The result produced through this multiplication is the total number of triples generated by the test.

This test generated 12.8 million triples.

Data generation test 6 Since the dataset was far smaller than we aimed, we continued generating triples. We found that the parameters that essentially influences the outcome are: `DEPT_MIN` and `DEPT_MAX`. Therefore, we changed the current values of all other parameters to their default value. The details of the parameter changes are given below:

- `UNDER_COURSE_NUM` was 10000—changed to 100;
- `GRAD_COURSE_NUM` was 20000—changed to 100;
- `DEPT_MIN` was 150—changed to 1500;
- `DEPT_MAX` was 250—changed to 2500;
- `RESEARCH _NUM` was 3000—changed to 30;
- `UNIV_NUM` was 60000—changed to 6000.

The size of the dataset produced by this test is 8260 MB which contains around 100 million triples. The number of triples was calculated the same way it was calculated in test 5.

Data generation test 7 Test 6 generated a fairly large amount of triples however the number was not yet the one we targeted to achieve. Therefore, we investigated data generation application source code to find the reason why SHARD is not producing a big dataset with a billion of triples even though the values of parameters were changed several times. We found that, the change of *values* of parameters in source code is not sufficient. The values of the parameter of program argument should also be changed. Therefore, we changed the parameter *-univ 1* in original argument line *-univ 1 -onto http://www.lehigh.edu/7Ezhp2/2004/0401/univ-bench.owl* to *-univ 6000*. The remaining configuration was same as test 6.

Finally, SHARD produced an impressive result. More than one and half billion triples were generated with the size 220 GB which is labeled *C_UdataSet* . After this number, we understood why the previous tests had failed to generate a Big Data. The dataset was stored on disk for further experiments. Table 2 summarizes the outcomes of all the data generation tests that have been conducted until now.

6.1 Analysis

In this subsection, we analyze the performance of data generation package of SHARD triplestore application. SHARD's data generation package is sustainable for a small

Tests	Size of Dataset (in MB)	No. of Triples
Test 1	13	121477
Test 2	35	339318
Test 3	181	1.4 Million
Test 4	337	3.2 Million
Test 5	1329	12.8 Million (App.)
Test 6	8260	79.6 Million (APP.)
Test 7	220160	1600 Million (App.)

Table 2: Outcomes of the dataset generation tests

and medium scale dataset generation however, we experienced several crashes while trying to generate large scale dataset. The generator was slightly modified to generate the big dataset containing more than a billion triples. We conclude that the updated version of SHARD is relatively more sustainable than the original one for generating large scale datatsets.

The performance of SHARD data generator with respect to *data generation time* was satisfactory yet again for generating small and medium scale dataseta. Nevertheless, for the large-scale dataset, it consumed an excessive amount of time. To be specific, for test 7 it took around 40 hours. The data generation time should be optimized.

6.2 An Open Issue

The data generation package of the SHARD framework was not able to count number of generated triples in dataset without launching a query. In SHARD, the triple counter class is contained in the SHARD triplestore package and this is the reason the triples of a newly generated dataset could not be counted immediately. From our experience, this design of SHARD's data generation system is not always feasible especially if an user wants to know the number of generated triples right after performing the generation operation. We found this fact while we were experimenting SHARD's data generator. If we wanted to know the number of generated triples , we had no option but to launch a query. More importantly, if the dataset is big, it takes several hours to know the amount of triples after launching a query. If the query processing application crashes then the users will never know the number. In our experiment 5, 6, 7, we were experiencing a crash (which we have reported in the previous section) while processing queries due to the large size of the dataset and therefore, we had to choose an alternative way to count the triples.

Based on these observations, we strongly suggest that the triple counter of SHARD should be integrated with its data generation application as well, rather than only bundling it with its triplestore application.

7 N3 Analyzer—An Extension of SHARD Data Generator

In order to address the open issue described in section 6.2, we developed an analyzer called the "N3 Analyzer." This analyzer is integrated with the SHARD data generator. As as result, the data generator can provide a specified number of triples in the generated dataset. The UML class diagram in Figure 2 shows the extension of the SHARD's data-generation application.

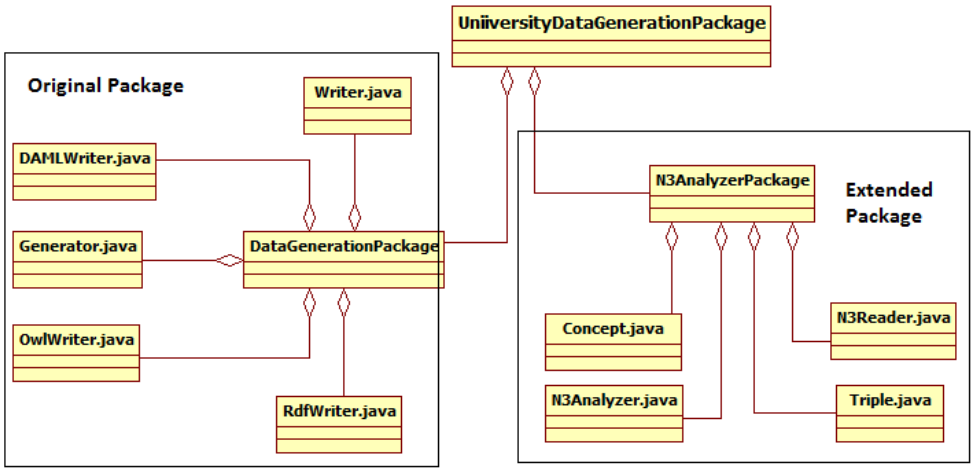


Figure 2: Class diagram for the extended SHARD data generation application

The class diagram shows that the university's data-generation package is composed of the original data-generation package, and our N3-Analyzer package extending it. The package contain the `Triple.java` class, which contains the method for counting the triples. In addition to designing our N3 Analyzer to count triples, we added another functionality (the "analyzer" *per se*), which can generate specifically scripted RDF graphs from the datasets. The `N3Reader.java` class specifies how to generate a script by reading the graphs contained in the dataset. The key purpose of this capability is to generate RDF graphs conforming to the specific nature of an ontology. Figure 2 shows the classes of N3 Analyzer package.

8 Experimentation with Triplestores

In this section, we provide experiment results. The experiments were performed both in non-cloud and cloud physical environment. We used fourteen queries provided by LUBM.[26] The queries were studied to identify the number of joins, variables, and triple patterns exist in these queries. Table 3 shows the constituents of the queries.

[26] http://swat.cse.lehigh.edu/projects/lubm/queries-sparql.txt

Queries	No. of joins	No. of variables	Triple pattern
Query 1	1	1	2
Query 2	5	3	6
Query 3	1	1	2
Query 4	4	4	5
Query 5	1	1	2
Query 6	0	1	1
Query 7	3	2	4
Query 8	4	3	5
Query 9	5	3	6
Query 10	1	1	2
Query 11	1	1	2
Query 12	3	2	4
Query 13	1	1	2
Query 14	0	1	1

Table 3: List of joins and variables in LUBM Queries

8.1 Experimentation with the SHARD Triplestore

The SHARD triplestore was experimented on both cloud and non-cloud infrastructures. The non-cloud infrastructure provides local mode fore processing jobs whereas the cloud environment distributed mode for running jobs. These two modes are briefly explained below:

- **Local Mode**: It provides a non-distributed mode of experiment where data is stored into and read from the stand-alone machine. Typically, local mode is provided by a single machine cluster.
- **Distributed Mode**: This mode provides a distributed runtime environment for processing queries. In this mode, data are distributed across several nodes on clusters.

The SHARD triplestore processes queries in three steps. They are as follows:

- **Data Cashing**: The query processing starts with copying the data from the local source (HDD) to the triplestore. In distributed mode, data is cached in the namenode. From the cache, the data is distributed to the cashes of child nodes. Child nodes are also called compute node. The distributed caching of data is managed by the HDFS system.
- **Storing Intermediate Result**: During query processing SHARD produces intermediate result (See section 3 for more detail). The intermediate results persist onto disk.
- **Storing Result**: The final outcome is produced in final step and persist onto disk.

Since the SHARD triplestore copies data from a local source to HDFS for each new query, the machine where HDFS is installed should have enough space. It is worth noting that in SHARD the triplestore removes the cached data after completing the query processing.

8.1.1 Experimentation with SHARD—Phase I

In this subsection, we present and analyze the results of the experiments that were carried out in this phase. All experiments were performed on a non-cloud infrastructure. The queries were processed in local mode. The key purpose of experimenting SHARD on a non-cloud infrastructure was to investigate the performance of SHARD in a standalone machine.

The Hadoop cluster in our non-cloud infrastructure contains only one node which in our case is 'Ced_Exp_Sys_3.' This node plays multiple roles include: *namenode*, *datanode*, *secondary node*, and *BackupNode*.

Query processing tests In the first test, we ran the LUBM queries on `C_UdataSet` containing 1600 million triples. Unfortunately, the SHARD system failed to process the queries. It crashed with a Java heap space error. We discuss the reason for this error, and how we corrected it.

Starvation Error	Heap space errors are common in Java based components. It happens when a component is used on a dataset too large to. This is what happened when we launched our query. The Java heap space is the memory of the Java Virtual Machine (JVM). The JVM gets this memory from the main memory. While processing any job, if the JVM lacks memory it requires, then it throws a Java Heap-Space Error. In our case, the system we used for testing the query had only 8 GB Random Access Memory (RAM). Thus, the system obviously could not share memory with the JVM as required for processing the query. Therefore, the JVM threw a `java.lang.OutOfMemoryError` exception. This error reveals a fact that SHARD heavily relies on main memory.
Solution	Since the exception was thrown due to shortage of memory required by the JVM to process the queries, we increased the heap size for the JVM. Additionally, we sliced the `C_UdataSet` into smaller sizes. Table 4 shows the different sizes of the datasets. We wrote a Linux shell-script for dissecting the various `C_UdataSet` datasets. This script is provided in Appendix Section B.

Datasets	Size in GB
C_UdataSet_S1	16
C_UdataSet_S2	20
C_UdataSet_S3	30
C_UdataSet_S4	40
C_UdataSet_S5	60
C_UdataSet_S6	75
C_UdataSet_S7	80
C_UdataSet_S8	97
C_UdataSet_S9	116
C_UdataSet_S10	120
C_UdataSet_S11	140
C_UdataSet_S12	160
C_UdataSet_S13	180
C_UdataSet_S14	200
C_UdataSet_S15	220

Table 4: Size of the CEDAR datasets

From the experience we gained in the first test, we decided to run next experiments on smaller datasets. In this test, the queries were launched on C_UdataSet_S2 which is 20 GB in size containing 153.61 million triples. The number of triples increases after reasoning. The SHARD triplestore carries out reasoning over subClassOf and subPropertyOf properties. The triplestore application performs reasoning before processing the queries. The dataset C_UdataSet_S2 became larger than its original size after reasoning. The number of triples grew to 179.1 million. The application processes queries on this newly created dataset. The query response time of this test is provided in the results and analysis discussion below.

The third test we conducted on the C_UdataSet_S4 dataset containing 306.9 million triples which increased to 358 million triples after reasoning the dataset. The fourth, fifth, and sixth test was carried out on C_UdataSet_S5, C_UdataSet_S7, and C_UdataSet_S8 which contain respectively 460.4, 613.9, and 767 million triples (before reasoning), and 536.9, 715.9, and 894.8 million triples after reasoning. We conducted another experiment on C_UdataSet_S9. However, we experienced a fatal error which could not be solved until later.

Query results and analysis Table 5 presents the query response times (*in millisecond*) of the queries that were carried out on different datasets in Phase I.

To provide a comprehensive visualization of the response times, the results are presented in a histogram. Figure 3 presents the histogram. This figure shows that response time of queries 14 and 6 are almost the same and the lowest of all. Query 2 takes the maximum time to be processed. The response times of Queries 4, 7, 8, and 9 are less than that of Query 2. Nevertheless, these queries consume a significant amount of time

Queries	C_Udata-Set_S2	C_Udata-Set_S4	C_Udata-Set_S5	C_Udata-Set_S7	C_Udata-Set_S8
Query 1	349031	664987	986942	1332158	1676712
Query 2	1489547	2846984	4319285	5916312	7529019
Query 3	351541	675960	996063	1338456	1680696
Query 4	1300913	2535974	3843819	5197707	6607025
Query 5	352580	665793	995289	1337403	1683091
Query 6	144257	274298	410428	547540	687070
Query 7	1134418	2172838	3366242	4576603	5845316
Query 8	1284526	2496910	3829507	5202318	6594931
Query 9	1173616	2244575	3376164	4518199	5683806
Query 10	354639	671969	996184	1347573	1679463
Query 11	350556	670777	1008015	1355364	1684884
Query 12	777031	1480790	2201811	2996355	3718997
Query 13	301501	568546	846867	1150081	1420353
Query 14	148243	277254	408406	551538	684580

Table 5: Response times for the queries processed in the non-cloud infrastructure

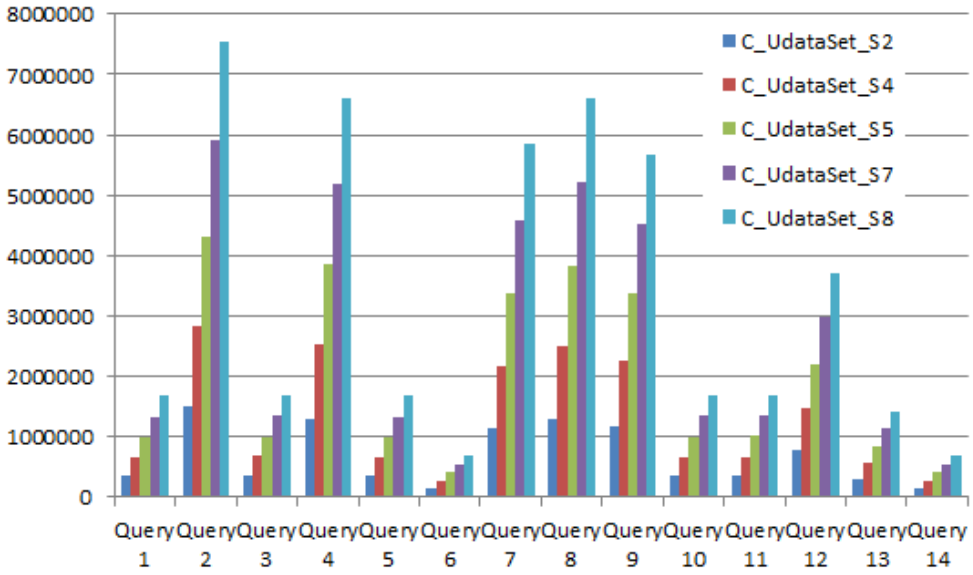


Figure 3: Comparison of response time for queries in the non-cloud infrastructure

for processing. The response time of Queries 1, 3, 5, 10, 11, and 13 is noticeably lower than those of other queries, except those of Query 6 and Query 14, which it exceeds.

Our analysis reveals a few interesting aspects of these results. For example, the response time depends heavily on the nature of the queries. Taking Query 2 for example: it has 5 joins, 6 triples patterns, and 3 parameters—which indicates the complexity level of this query is high. On the other hand, Queries 6 and 14 have no join. The response time of Query 2 is four times those of Queris 4 and 16. However, Query 9 has

the same number of triple patterns, variables, and join as Query 2, but the response time of this query is slightly lower than that of Query 2. Furthermore, Query 4, 7, and 8 have almost the same specification; hence, the response times of these queries are nearly the same with minimal difference. The remaining queries including Queries 1, 3, 5, 10, 11, and 13, have the same specification; therefore, their response times are nearly the same.

The fact that the nature of a query determines the response time is an observable reality. During experiments, we have observed that the patterns of response time of queries for all datasets are the same. To clarify more, the pattern of response time for Query 2 is same for all datasets. The response time of this query is always the highest because it is the most complex query of all .Conversely, the response times of Query 4 and 6 are the lowest for all experiments. The reason is evident: there is no join in these queries.

Another important observation is the positive correlation between the size of the dataset and the complexity level of the queries. The response times of complex Queries 2, 4, 7, and 8 increase significantly with the size of the dataset. The response time of Query 2 for `C_UdataSet_S4` was significantly more than the response time of the same query for `C_UdataSet_S2`. The response time of simple queries increase with the increment of the size of the dataset however, the increment rate is minimal. Figure 8 shows that the response times of Queries 1, 3, 5, 6, 10, 11, 13, and 14 increase moderately with the size of the dataset.

8.1.2 Experimenting with SHARD—Phase II

This section gives the results of the experiments conducted in the PetaSky cloud infrastructure.

Two experiments were launched on `C_UdataSet_S9` which led to the same fatal error encountered before on that same dataset. We concluded that the non-cloud single machine cluster host cannot handle this large dataset. There were two options we could choose: (i) assembling a new host with a richer specification; or, (ii) a scalable infrastructure that could handle a large scale dataset. Since nowadays a scalable infrastructure is cost-effective, we preferred a cloud-based scalable infrastructure over the other option. Another reason for selecting a cloud-based infrastructure was performance. Since it facilitates invoking instances on demand, it can optimize the performance of the SHARD triplestore.

In this phase, the experiment was moved to the cloud-based infrastructure provided by PetaSky—a project of handling extremely large dataset [19]. The infrastructure provides distributed environment for experiments. The PetaSky cloud infrastructure comprises one cluster consisting of three instances `PETASKY-1`, `PETASKY-2`, and `PETASKY-3`, all with the same specification, except concerning the storage. PetaSky uses Serial Advanced Technology Attachment (SATA) storage technology.[27] The SATA storage of `PETASKY-1` is 1 TB; the storage of `PETASKY-2` and `PETASKY-3` is 300 GB. The other details of these instances is given below.

[27] http://en.wikipedia.org/wiki/Serial_ATA

- **Processor**: Intel (R) XEON (R) CPU 1.87 Ghz.
- **Processing Speed**: 64 bit Processor
- **Memory**: RAM 18.542640 GB
- **Operating System**: 64 bit Debian 3.2.35.2

Each instance of the cluster contains four virtual processors. The cluster in the PetaSky cloud infrastructure was configured by assigning `PETASKY-1` as namenode, and assigning `PETASKY-2` and `PETASKY-3` as data nodes. Notably, the namenode plays the role of data node as well. It is also worth noting that these instances are shared by other users as well because, PetaSky is a public cloud.

The SHARD framework was installed on the PetaSky cloud. The datasets were copied from the non-cloud host machine to the PetaSky instance `PETASKY-1`. The query file `queries_sparql.txt` was also copied to the same instance.

Four experiments were performed on the PetaSky cloud infrastructure. The queries were performed on the following datasets:

- `C_UdataSet_S2`
- `C_UdataSet_S4`
- `C_UdataSet_S5`
- `C_UdataSet_S7`

The results of these experiments were recorded and analyzed.

Query results and analysis Table 6 presents the results of the queries performed on the datasets. Since scalability was not the issue, the experiment was shifted to cloud based infrastructure to optimize the response time. However, the response time was not optimized as desired although this infrastructure provides more processing capability than the non-cloud one. The performance of SHARD was rather degraded in distributed environment. The response times of queries are more than the response times of queries performed in non-cloud based single machine cluster infrastructure. This is the main reason no experiment was conducted after finishing the experiment with dataset `C_UdataSet_S7`. Figure 9 shows the response times of queries on different size datasets.

Interestingly, the pattern of response time is found the same as the previous experiments. To explain more, like the previous experiment, response time of Query 2 is the highest of all whereas response times of queries 4 and 16 are the lowest of all. Like the experiments in Phase - I, the correlation between size of the dataset and response time is positive.

While running experiments on the PetaSky cloud, we observed that one or more of the compute nodes failed. Hadoop is a fault tolerant system, the namenode can assign the job of the failed compute node to another node which is active. In hadoop, each compute node sends a signal called *heartbeat* to the namenode at a regular interval. The

Queries	C_Udata-Set_S2	C_Udata-Set_S4	C_Udata-Set_S5	C_Udata-Set_S7
Query 1	437762	1270535	1848299	2675383
Query 2	1431660	4098426	5967641	8868094
Query 3	398984	1199660	1813116	2640609
Query 4	1107177	3317469	4941861	7301488
Query 5	408639	1222865	1786475	2626656
Query 6	174279	573898	836381	1243090
Query 7	910899	2740167	3951826	5909064
Query 8	1109634	3399223	4921301	7178025
Query 9	1168131	3785146	5494386	8016436
Query 10	399340	1248625	1795034	2595177
Query 11	401399	1243834	1780964	2589490
Query 12	799216	2539781	3660651	5326362
Query 13	381591	1190937	1697386	2492465
Query 14	168119	563958	832149	1211306

Table 6: Response times for the queries on the PetaSky cloud infrastructure

default value for interval is three seconds. If the namenode does not receive heartbeat from a compute node, then it considers the compute node a failed node and assigns corresponding job to another active computer node.

The compute nodes PETASKY-2 and PETASKY-3 were failed to send heartbeat to the namenode PETASKY-1 within the specified time and therefore were considered the failed nodes. We observed that one compute node failed more than once while running the queries. Table 7 shows the number of times the PETASKY-n instances

Datasets	Occurrence of Failure in PETASKY-2	Occurrence of Failures in PETASKY-3
C_UdataSet -1	3	2
C_UdataSet -2	5	7
C_UdataSet -3	6	8
C_UdataSet -4	9	11

Table 7: Occurrences of failure during experiments

had failed.

Furthermore, we observed that the failure rate of compute nodes increased with increment of datasize. The failure rate of both compute nodes of PETASKY-2 and PETASKY-3 was found more while processing queries on C_UdataSet_S4 than C_UdataSet_S2 dataset. Notably, the namenode PETASKY-1 is a compute node as well which never failed. According to our understanding, the high failure rate of compute nodes was the reason which increased the query response times substantially.

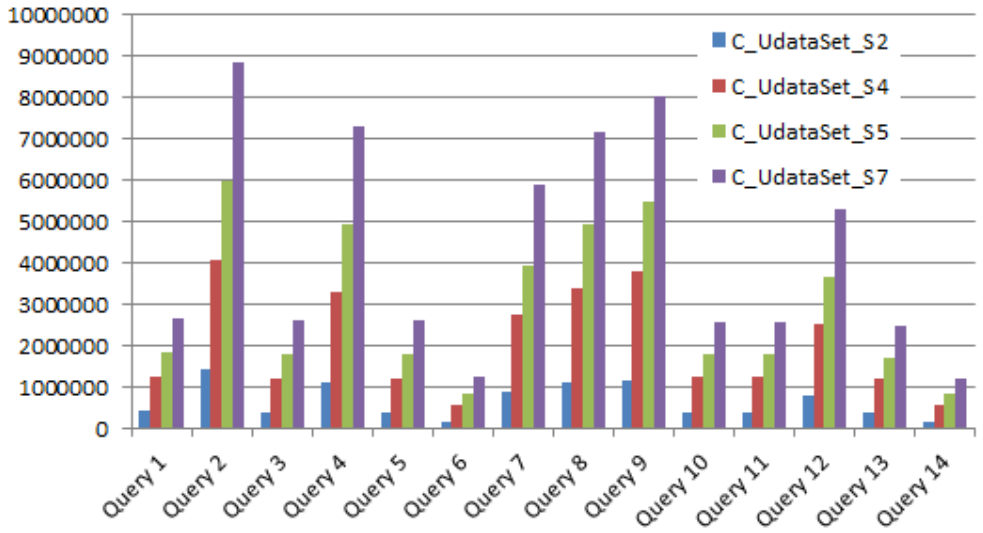


Figure 4: Comparison of response times for queries on the PetaSky cloud infrastructure

8.1.3 Experimentation with SHARD—Phase III

SHARD so far was scalable as the triplestore could handle gigabytes of data containing millions of triples. Nonetheless, the performance was never satisfactory. However, the creators of SHARD did claim that the repository is scalable and efficient [20]. They supported their claim based on experiments they conducted with SHARD. The results of their experiments showed that the response times of Query 9 is 740 seconds, which essentially indicates that SHARD is able to process the queries with much higher speed than what we observed in our experiments. Their queries were performed on a dataset containing 800 million triples which is equivalent to our `C_UdataSet_S8` GB dataset. Since Kurt *et al.* used 19 XL nodes of the Amazon's Elastic Cloud,[28] we assumed that the performance discrepancy we observed for our dataset was perhaps due to the number of instances required to process big data. Therefore, we built our own LIRIS cloud infrastructure to contain nineteen instances, each of which has the same specification except for the storage of the namenode. The specification of the LIRIS cloud instances is as follows:

- **Processor**: Intel Core Duo CPU 2.2 Ghz. Each instance has 2 processors.
- **Processing Speed**: 64 bit Processor
- **Memory**: RAM 8 GB
- **Operating System**: 64 bit Ubuntu 12.04

Considering the storage, the namenode of the LIRIS cloud has 610 GB of storage, whereas the remaining instances have 110 GB each. Like PetaSky, the LIRIS cloud is public; thus, the instances are shared by other users. It was configured by assigning

[28] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html

Queries	C_UdataSet_S2	C_UdataSet_S4
Query 1	940206	2901310
Query 2	3664658	8948040
Query 3	1251061	2937791
Query 4	3085118	7449991
Query 5	1217698	2913151
Query 6	597691	1452316
Query 7	2466565	5882546
Query 8	3099742	7213797
Query 9	3612315	8421878
Query 10	1225850	2859148
Query 11	1231084	2862419
Query 12	2421422	5754733
Query 13	1214504	2926672
Query 14	593124	1449000

Table 8: Response times for queries processed on the PetaSky cloud infrastructure

one namenode and eighteen compute nodes, with the namenode playing the role of a compute node as well.

The SHARD framework was installed on the LIRIS cloud environment. The datasets and the query file were copied to LIRIS Cloud from the non-cloud host. Only two experiments were performed on the LIRIS Cloud infrastructure. The queries were carried out on the C_UdataSet_S2 and C_UdataSet_S4 datasets.

Results and analysis The results are shown in table 8. Figure 5 shows the comparison of response time of experiments carried out on the C_UdataSet_S2 and C_UdataSet_S4 datasets. The results of these experiments are disappointing. The response times of queries increased dramatically. Furthermore, we observed that the response times of the queries of the second experiment are significantly more than the response times of queries carried out in the first experiment. Such a huge difference between the response times of the queries was not seen in any of the previous experiments.

After observing the results of queries shown in Table 8, we decided to stop experiment.

8.1.4 A Comparison of 3-Phase Experiment Results

In this section, we provide a graphical representation of the results produced by the experiments in three phases. We restrict the comparison to C_UdataSet_S2 and C_UdataSet_S4 datasets because only these two experiments were carried out successfully on the LIRIS Cloud infrastructure.

The analysis reveals that increasing the number of instances only cannot optimize

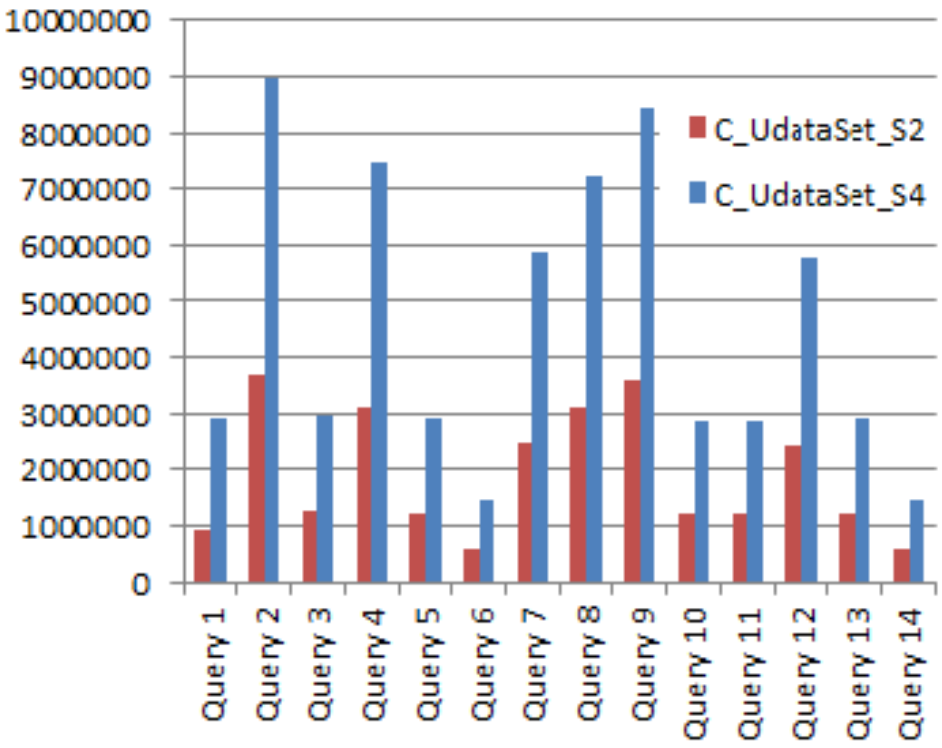


Figure 5: Comparison of response times of queries on the LIRIS cloud infrastructure

the performance of SHARD although the theory says otherwise. We observed this fact in our experiments. To be fair, in a few cases such as for Queries 2, 4, 7, and 8 on `C_UdataSet_S2`, SHARD on the PetaSky cloud did performe better than the single non-cloud machine. However, we observed that it was not the same for `C_UdataSet_S4` dataset. Figure 7shows the comparisons of the response time of queries performed on this dataset in three phases. For this dataset, the response times produced by the non-cloud host are the best of all. This implies that, with the increment of datasets, the performance of SHARD degraded in distributed environment provided by the PetaSky and LIRIS cloud infrastructures.

The results produced by SHARD on the distributed environment provided by the LIRIS cloud infrastructure raised an important question: *Why could SHARD not perform better on a larger cloud-based cluster than on a single non-cloud machine*? The results led to another important question: *Why could SHARD not reproduce the results reported in [20]*?

From our experience, we strongly disagree that only scaling up the infrastructure optimizes SHARD's performance. There are more factors that influence the performance of SHARD. In order to identify these factors, we stopped experiments temporarily and started a rigorous investigation on experiment environment and technologies we employed in our experiments. The focal point of the investigation was Hadoop.

After the investigation, we designed a new plan to conduct experiments. The next

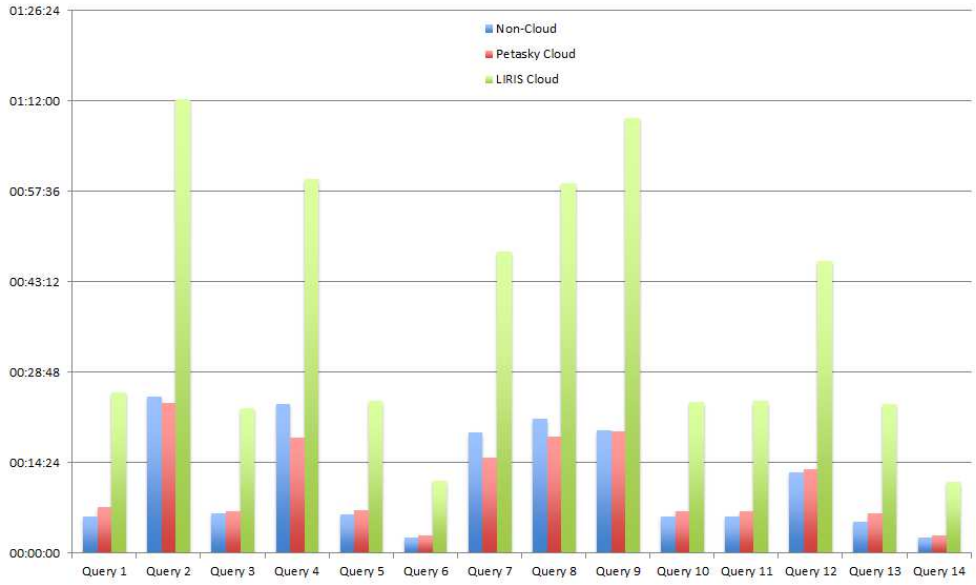


Figure 6: Comparison of response times of three experiments with the `C_UdataSet_S2` dataset in the three phases

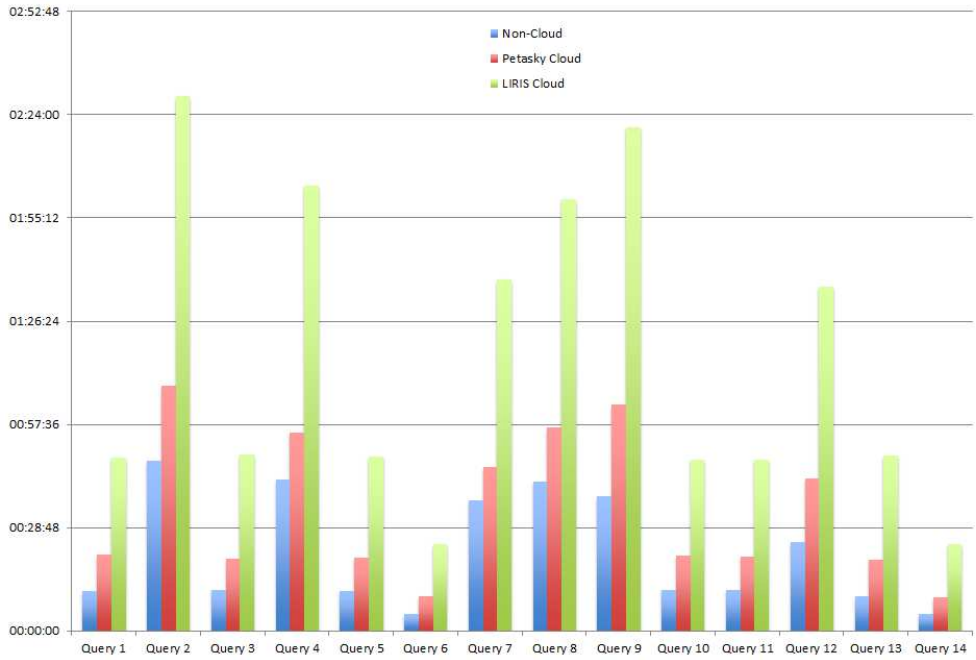


Figure 7: Comparison of response times in three experiments with the `C_UdataSet_S4` dataset in three phases

subsections describe the experiments.

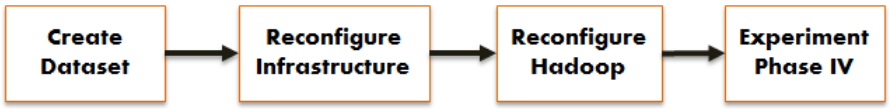


Figure 8: New experiment plan

8.1.5 SHARD Reloaded

In this experimental research, our goal was to reproduce the results of SHARD recorded in [20]. In order to achieve this goal, we followed a twofold approach: first, we concentrated on how to get better performance than that obtained on the non-cloud host. Then, we applied this knowledge to try and reproduce the published result. Our assumption was if we knew how to achieve the first target, we would be able to reach the other. Figure 8 shows the experiment plan.

- **Dataset Creation**: A new smaller dataset was created. The essential reason for creating a smaller dataset was to reduce the total processing time of the experiment. The new dataset `C_UdataSet_S16` is 8 GB in size.
- **Infrastructure Reconfiguration**: The LIRIS cloud infrastructure was reconfigured. We suspended all the machines from the LIRIS cloud infrastructure. The plan was to compare the one-to-one performance between non-cloud and cloud hosts with one instance. The purpose was to check the performance of SHARD on two different infrastructure having same number of machine with the same specification.

  We configured a new instance for the LIRIS cloud infrastructure. The specification of the instance is given below:

  - **Processor**: Intel Core Duo 2.20 Ghz 64 bit
  - **Memory**: 8GB
  - **HDD**: SATA 30 GB

  The instance has 8 virtual CPUs (VCPUs). The reason for installing 8 VCPUs was to equalize the processing power of the non-cloud host. We were aware of the fact that the processing speed influences the performance of Hadoop. Notably, 1 Core CPU = 2 VCPUs. Since the non-cloud host had 4 core CPUs, this was equivalent to 8 VCPUs. Nonetheless, there were still differences between these machines. There were two differences: (i) the processing power of the cloud machine was shared whereas the non-cloud based host was not; and, (ii) the processors of LIRIS cloud instance are less powerful than the processors of the non-cloud host.
- **Hadoop Configuration**: We found out that Hadoop has more than 190 parameters. We also realized that many of these parameters are critical in determining the performance of Hadoop, although without any clear correlation. We configured Hadoop by changing the values of these parameters following a "guess-and-test" approach.

The SHARD framework was installed on the LIRIS cloud machine. We ran several experiments in this phase. The next subsection describes the experiments.

8.1.6 Experiments with SHARD—Phase IV

This section presents the analyses the results of the experiments we conducted during phase IV. During these experiments, different strategies were applied. Experiments were performed at *OffPeak* period, the default values of the Hadoop parameters were changed, and the datasets were compressed. The term 'OffPeak' refers to a period when the instance is not shared by many users. Notably, these strategies were applied to the LIRIS cloud infrastructure only.

We used the `C_UdataSet_S16` dataset for the first experiment on the non-cloud infrastructure. The subsequent experiment was carried out on the LIRIS cloud infrastructure on the same dataset at 'Peak' period with default values. The experiment was conducted in local mode. The subsequent experiments were performed on the LIRIS cloud at 'OffPeak' period with default values of the parameters.

Since neither of these experiments produced a better outcome than the results produced by the non-cloud host, we proceeded changing the default values of the Hadoop parameters hoping to enhance performance. Five experiments were conducted at 'OffPeak' period with the changed values of the parameters. The best results with respect to the response time produced by these experiments were close to the response time produced by the non-cloud host.

For the next experiments, we compressed the dataset using `Bzip2` CoDec. Five experiments were conducted on the compressed dataset with changed default values of the parameters at 'OffPeak' period. The best result among these five was very close to the result produced by the machine hosted by the non-cloud infrastructure.

For the next experiment, we added a new instance in the LIRIS cloud infrastructure and local mode for processing jobs was changed to distributed mode. Then, we launched the experiment on the `C_UdataSet_S16` dataset at 'OffPeak' period. The results of this experiment were better than the results of all experiments conducted up to that point on both the cloud and non-cloud infrastructures. The next section discusses the results.

Results and analysis We now compare the outcomes of the experiments. We selected the best results from the group experiments. In particular, we selected results from the following experiments:

- `Exp_OffPeak_1_DV`,
- `Exp_OffPeak_1_CV`,
- `Exp_OffPeak_6_CV_Compressed`.

We also selected `Exp_Peak_1_DV` and `Exp_NC_1_DV` for the comparison. Table 9 shows the response times of the queries performed in these experiments.

Queries	Exp_Off-Peak_1_CV	Exp_Off-Peak_1_DV	Exp_Off-Peak_6_CV	Exp_Off-Peak_1_DV-_Compressed	Exp_NC Peak_1_DV
Query 1	145688	151854	158595	2955467	132724
Query 2	776775	1315630	408698	1249933	538170
Query 3	147519	145686	128140	232571	135273
Query 4	931979	903893	367268	1076594	466737
Query 5	186404	163412	119686	181673	134293
Query 6	60898	53216	48709	67732	54174
Query 7	1167413	1007699	379748	1172036	392557
Query 8	1154007	938264	311338	955160	455734
Query 9	550115	522442	335699	584148	450885
Query 10	139327	153408	120980	167346	134271
Query 11	139458	138333	123515	270536	134306
Query 12	322609	308843	232111	602061	292719
Query 13	118369	120454	117018	156577	113274
Query 14	55144	56109	48971	114361	54140

Table 9: response times for the Queries on a 8 GB dataset

Figure 9 depicts the comparison of the results produced in these experiments.

To simplify our discussion, we rank the experiments according to response time. Table 10 shows the ranking.

Experiments	Rank
Exp_OffPeak_1_DV	3
Exp_OffPeak_1_CV	3
Exp_OffPeak_6_CV_Compressed	1
Exp_Peak_1_DV	4
Exp_NC_1_DV	2

Table 10: Ranking of experiment response times

This analysis shows that the queries of the experiment Exp_Peak_1_DV are the most expensive as the response time of each query are longer than the response time of all other experiments except for Query 2. The experiment Exp_Peak_1_DV is ranked at the 4th place. By contrast, this also shows that the response time of the queries of the experiment Exp_OffPeak_6_CV_Compressed is the lowest except for the response time of Query 1. For the first time, the performance of SHARD in a distributed environment provided by the LIRIS cloud infrastructure improved on its performance on the non-cloud host. The experiment *Exp_OffPeak_6_CV_Compressed* is ranked 1st. The experiments *Exp_OffPeak_1_DV* and Exp_OffPeak_1_CV are ranked 3rd since the response response time of these experiments are equal. The experiment *Exp_NV_1_DV* is ranked 2nd. The response time of this experiment is better than other experiments except for Exp_OffPeak_6_CV_Compressed, which is the best of all.

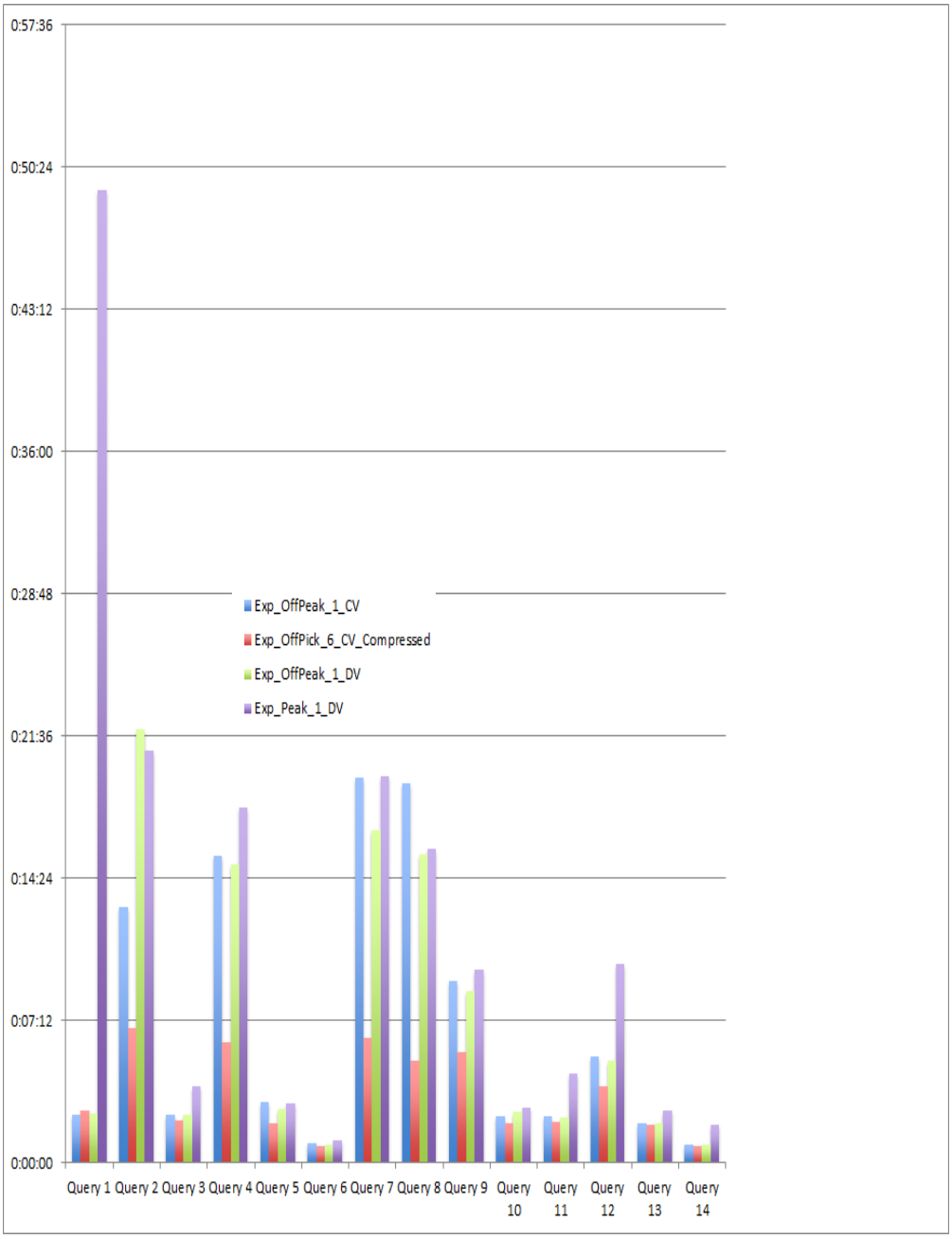


Figure 9: Comparison of response time performances on an 8 GB dataset

8.2 Experimentation with HadoopRDF

In this section we describe our experiments with HadoopRDF. HadoopRDF was installed on the non-cloud host 'Ced_Exp_Sys_3' where Hadoop was already configured. The datasets and query file were already stored in the host machine. The only experiment conducted using this triplestore was on the dataset '`C_UdataSet_S2`.' The

framework works in two different phases: (i) *data preprocessing phase*, and (ii) *Query Processing Phase*. The three steps in the data preprocessing phase were: (i) *splitting triples by predicates* (PS), (ii) *splitting triples by predicate Object Type*(POST), and (iii) *splitting triples by predicate Object of Non Type* (POSNT). While running the first phase, the experiment could not be completed successfully. The system threw an exception that is explained below.

PS Error	During the preprocessing phase, HadoopRDF uses its `Parser` to parse the `PREFIX`es that exist in a dataset. The error was thrown by the system due to failure of this `PREFIX` parsing phase. As we analyzed the parser carefully, we found that the problem was due to a misconfiguration of the default input and output paths. The parser could not find the input path to read the files and the output path to store the data after parsing.
Solution	We created two different folders: an *input folder* for storing the data read by the parser and an *output folder* for writing the triples after the split.

The PS step was completed successfully after fixing the error. The next steps were POST and POSNT. We observed that the POSNT step completed successfully *if and only if* the POST step completed successfully. However, HadoopRDF failed to complete the POST step on our datasets. While running the experiment, the system threw an exception. Unfortunately this error could not be fixed, and therefore the triplestore could not be tested any further.

9 Expectation *vs.* Reality

Before starting the experiments, we expected SHARD to reproduce the result reported in [20]. However, from what we could observe in our experiments, SHARD failed to reproduce the result even though the exact same infrastructure was provided. The reproducibility of SHARD's behavior was always in question throughout the experiments. The results were not always reproducible in our experiments as well. Figure 10 illustrates this fact, showing the outcomes of the experiments conducted on the '`C_UdataSet_S16`' dataset.

We conducted six experiments using the same environment and infrastructure on this dataset. Figure 10 shows that the outcomes for most of the queries were not consistent. In some cases, the response times of the same query performed in different experiments vary significantly—for example, the response times of Query 2, 4, 7, and 8.

According to our analysis, SHARD's performance heavily relies on Hadoop's configuration. Our observation revealed that SHARD plays virtually no role in optimizing the response time. The triplestore does not provide any mechanism such as *indexing* that could improve its performance.

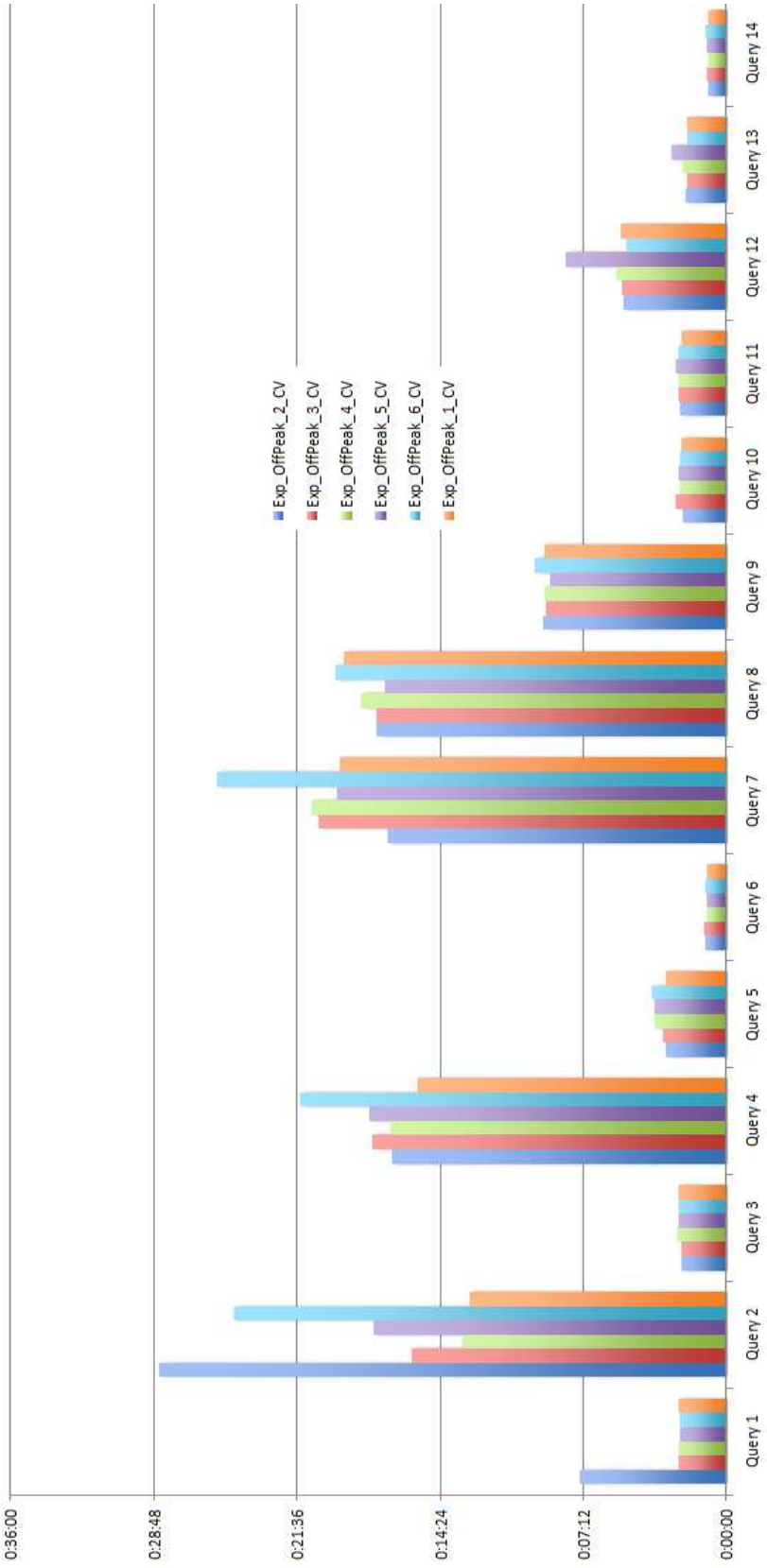


Figure 10: Response-time pattern

One might expect that *increasing the instances in the environment will increase the speed of processing map and reduce jobs.* However, based on our experience, we can only conclude that this is not true. Essentially, several parameters influence the processing time of map and reduce jobs. Our expectation was that changing the values of theses parameters would improve the performance of Hadoop. While some minor improvement was observed in experiments with changed values of the parameters, that was still far from the desired level. The performance of Hadoop reached optimum level when the number of instances was increased, the values of parameters were changed, and compression and decompression technique was used.

10 Conclusion and Future Work

In this technical report, we presented the results of experiments performed on SHARD and HadoopRDF triples stores. The report provides the detail of what has been experienced and attained from the experiments.

An experiment platform called CedExP was introduced in this report. CedExP is an extensible platform which facilitates experimenting scalable triplestores. The platform facilitates scaling up the computing node to any number upon requirement. This platform was used in conducting the experiments.

The experiment was started with SHARD and then HadoopRDF was experimented. While experimenting SHARD, it was found that SHARD triplestore cannot count the newly generated data until a query is launched on the dataset. Besides, it was necessary to understand the nature of the graphs. A graph analyzer called "N3 Analyzer" has been developed to count the number of triples contained in a newly generated dataset.

The main purpose of this experimental research was to evaluate the performance of the triplestores. Reproducing the outcomes reported in the literature was the *Litmus Test* for the triplestores HadoopRDF and SHARD. The performance reported in the literatures essentially demonstrates the high-level of efficiency of these triplestores. Thus, before running experiments with HadoopRDF and SHARD, our expectation was very high. However, they produced disappointing outcomes although CedExP was configured following the specifications provided in the literature. The results were not even comparable with the results published in the literatures.

Seeing the unexpected unsatisfactory results, we applied several techniques (such as CoDec) to improve them. Some improved the performance significantly, however not as much as we were expecting. These techniques have been explained in this report.

During our experiments, several errors were encountered. Solutions for most of these errors were provided. Nevertheless, a few errors were insolvable. For example, a fatal error in the JVM forced us to discontinue the experiment on the non-cloud host. Also, the experiment on HadoopRDF was cancelled due to a problem that is deeply rooted in the framework itself for which we had no access.

The essential epiphany that this work made us reach is that the standard Hadoop technology is far from being a magic wand. Still, it does provide a scalable, though low-level, processing infrastructure for processing Big Data. According to our experience,

in order to achieve efficient query processing, specifically for Big Linked Data (or "Blinked Data," as we call that), it must be adapted to the specificity of RDF triple-based data. This is because the HDFS, which is the central component of Hadoop, is a monolithic black-box design. Our experiments have unveiled a few shortcomings due to this non-transparency aspect of Hadoop. Nonetheless, we think that conducting our experiments has given us hints on how to modify Hadoop to enhance its ability to process queries on Blinked Data efficiently.

As for future experiments, testing other existing triplestores is coming next: Jena-Hbase [12], Jena TDB,[29] OpenLink's Virtuoso,[30] and RDF3X,[31] to name a few. We will also continue experimenting with SHARD until we can reproduce the published results.

A high-performance scalable triplestore is *sine qua non* for the $\mathcal{CEDAR}$ Project. The objective of this in-depth hands-on experimental work with the state of the art is to build our own triplestore for processing Blinked Data as needed by the $\mathcal{CEDAR}$ Project.

Appendix

A A Tale of a Safari

A.1 The trip plan

At the outset, our journey's essential plan started out with trying to obtain an answer to this question: *What candidate triplestores can fit the interests of the $\mathcal{CEDAR}$ project?* In order to answer this question, we scanned the web using different search strings such as "*big RDF repositories*," "*big RDF triplestores*," and "*Large-scale triplestore*." In this way, we could identify several candidate triplestores. Table 11 provides the list of repositories that were found on the web. Then, we filtered this list based on the following criteria:

- Is the architectural type of the RDF repository *centralized* or *distributed*?
- Is the type of the underlying technology *conventional* or *MapReduce-based*?
- Is the license type of the triplestore *proprietary* or *open source*?
- Is the scale of data that the triplestore can handle *large*, *medium*, or *small*?

Regarding the last point, the range of the scale is defined as follows. A dataset that is less than 20 GB is considered small scale; 21–50 GB is medium scale; above 50 GB is considered large. The scale is decided based upon the number of triples contained in a dataset of a particular size. For example, a dataset of more than 80 GB contains more than 500 million triples.

[29] http://jena.apache.org/documentation/tdb/
[30] http://virtuoso.openlinksw.com/
[31] https://code.google.com/p/rdf3x/

3Store	**4Store**	**5Store**	**BigData**
AllegroGraph	ARC	BigOWLIM	SHARD
BrightstarDB	Dydra	IBM DB2	SerQL
Apache Jena	Mulgara	OntoBroker	StrixDB
OpenAnzo	OpenLink	Oracle	OWLLIM
Meronymy	SPARQL DBS	Parliament	RAP
Pointrel System	Profium Sense	HadoopRDF	Stardog
RDF:Core	RDF:Trine	RDF-3X	Kowaqri
RDFBroker	Redland	RedStore	RDF gateway
Saffron Memory Base	Semantics Platform	Sesame	Virtuso Open Link
Jena HBASE			

Table 11: List of RDF triple repositories

Since distributed repositories that are built on *MapReduce* technology are our prime interest, the Hadoop/MapReduce technology was given higher priority. Therefore, those repositories relying on this technology were flagged as potential candidates to experiment with. Our essential motivation for selecting this technology rely on the following points:

- Hadoop/MapReduce-based triplestore can be easily scalable [20];
- Hadoop/MapReduce is a cost-effective technology as it can be deployed on conventional hardware [23]; and,
- Hadoop's Distributed File System (HDFS), which is the core strength of the Hadoop/MapReduce technology, reduces the complexity of handling many tasks, bearing the onus on itself.

Nevertheless, we do plan to test repositories that are built on other distributed technology at some time in the future, but not in the priority list. In fact, in the process of this initial investigation, we realized that most RDF repositories rely on traditional technologies rather than on MapReduce technology.

The license type of the repositories is another criterion for finding the potential candidate repositories. Since re-engineering open source applications does not have legal obligations, our first preference has focused on open source repositories.

We also paid attention to results of experiments that had already been carried out using such triplestores. This brought us to several immediate potential candidate triplestores. This set is just an initial set, or course. But we had to start somewhere, so we focused on this study on SHARD [20] and HadoopRDF [11]. Experimenting with other triplestores will follow in further work.

A.2 The trek itself

Painting the experiment landscape We designed and implemented an experiment platform that comprises different technologies. We downloaded these technologies Apache Hadoop, SHARD, HadoopRDF from the web. We also downloaded the experiment queries from the Lehigh University Benchmark (LUBM) portal and then stored them into a file.[32] LUBM has defined fourteen queries written in SPARQL.

The stepping stone First and foremost, we needed a dataset to be large. Therefore, the first step was to create a big RDF dataset. In order to create the dataset, we used LUBM data generator that is integrated in the SHARD triplestore as a component for generating datasets of universities. We customized the data generation parameters and created different datasets. We stored the dataset in appropriate locations.

Cedar meets the first challenge While generating the datasets, the application was crashed due to low specification of the host machine in particular, *low memory size*. Moreover, the host machine had low processing speed and storage. We replaced the host machine by a new machine with richer specification. Then, the data generator was hosted and configured in the new machine.

A simple test with SHARD The first triplestore we tested was SHARD itself. We started with a simple test on a very small dataset of size 13 MB. This test was essentially to check whether our experimental platform could be configured properly.[33]

The simple test failed and was fixed However, this simple test failed due to invalid SPARQL expressions found in the LUBM queries. The application threw an exception as it failed to read the syntax. In order to resolve this problem, we refined the queries. Then, we relaunched the experiment. Then, the test completed successfully.

The experiment begins After successful completion of the simple test, we concluded that CedExp had been configured properly. The experiments were conducted in four phases on non-cloud and cloud infrastructures. Additionally, the queries were launched as job runs both in local and distributed modes. In the first phase, the experiments were carried out on the non-cloud infrastructure and the mode of running jobs was local. We submitted the fourteen LUBM queries with SHARD on a 220 GB dataset.

Crashed and managed However, we experienced a crash immediately after submitting the queries. Upon observation, it turned out that the crash was due to the large size of the dataset. So we considered two possible workarounds: (i) replacing the host machine with a more powerful host; or, (ii) splitting the dataset into different sizes such

[32] http://swat.cse.lehigh.edu/projects/lubm/

[33] CedExp—see Section 5.

as 20, 40, and 60 GB. We chose the latter because we wanted to assess what maximum size of the dataset could be processed by the host.

The queries resubmitted After slicing the datasets, we relaunched our experiments on several datasets of size 20 GB each. We continued with such experiments using datasets of inceasing sizes until SHARD failed.

Encountering an unsolvable problem The last successful experiment was carried out on the dataset of 97 GB. After that, we ran into an exception while querying a 116 GB dataset. Unfortunately, we could not find any way to solve this problem. This compelled us to change the experiment environment.

Migrating to the PetaSky cloud-based infrastructure The second phase of the experiment begins here. In this phase, the experiment was migrated to the *PetaSky* cloud-based infrastructure hosted at the LIRIS.[34] Each of the LIRIS PetaSky machines is more powerful in terms of main memory than the machine we used for the non-cloud infrastructure. The datasets and query file were copied to the master machine of the PetaSky cluster. On these machines, we ran the queries with SHARD on datatsets of 20, 40, 60, and 80 GB, and all were successful.

The performance was mediocre We had expected an improvement of performance of the SHARD triplestore on the PetaSky cloud infrastructure than what we had observed on the non-cloud infrastructure. Unfortunately, that was not the case: the results produced by the experiments conducted on the PetaSky cloud environment were not satisfactory. In fact, the performance of SHARD in this three-machine cluster was *worse*!

Can larger be better? Since the targeted query-processing time was not achieved, a larger cluster consisting of twenty machines was built. This experiment infrastructure was named "LIRIS Cloud-Based Infrastructure." The SHARD triplestore was configured for the new infrastructure. Additionally, datasets and files were copied to this new infrastructure. Then, we ran the LUBM queries.

Our expectation regarding this cluster was very high. However, yet again, we were disappointed: the performance of these experiments degraded even more significantly.

What can change the world? The experiment results for the cloud-based infrastructure prompted a question: *What are the factors that influence the performance of an application that runs on a Hadoop-based environment?* Before starting our experiments, we had assumed that increasing the number of node instances would improve the performance of SHARD. However, the outcome was in fact the opposite. Therefore, we dedicated the fourth phase of our experiment to identifying exactly what factors do

[34] http://com.isima.fr/Petasky

indeed affect the performance of job processing using Hadoop and MapReduce. We configured the experiment platform on both the cloud and non-cloud environments. A small dataset of size 8 GB was created for the new experiments. The purpose of reducing the data size was to reduce the total time consumed by an experiment.

In search of the Holy Grail In our investigation, we found that Hadoop has several configuration parameters. We studied these parameters in order to identify those that influence the performance of Hadoop's job processing. We could narrow down this set of parameters to some that, as we could observe, did affect the performance of Hadoop.

Then, we restarted the experiments on both the cloud and non-cloud infrastructure. Some experiments were performed in local mode with default values of configuration parameters. A difference in terms of query processing time was observed in the results produced on the cloud and non-cloud infrastructures. The non-cloud machine host performed the best. This result triggered a question, *How can the cloud-based host be made to have a better the performance that that of the non-cloud host?* Answering this question was critical because the non-cloud host has a much lower capacity for dataset sizes. Indeed, for Big Data, the cloud-based host is clearly a mandatory platform.

At this stage, our aim was to answer this question. From this stage onward, we launched experiments on a host on the LIRIS cloud infrastructure only. The key idea was that, if it could be possible to elucidate how to ameliorate the performance of the non-cloud host, then it would be possible to reach the target performance set at the beginning of our study. To this end, we changed the values of the configuration parameters of Hadoop, and assigned different values in each experiment until the target performance was reached.

Triumph over feeble performance The experiments with changed values of the parameters produced significant results: the performance time did improved.

In our study, we found that CoDec can be applied to improve the performance of Hadoop. The dataset was compressed using BZIP2 CoDec and the queries were launched on compressed dataset. Interestingly, the result produced through this experiment was very close to the results produced by the non-cloud host. Then, we instantiated a new virtual machine on the LIRIS cloud infrastructure and we relaunch the queries. Finally, the cloud-based host outperformed the non-cloud host. The two-machine cluster with data compression technique and changed values of configuration parameters gave the best performance of SHARD.

HadoopRDF loaded Alongside SHARD, we tested HadoofRDF. Unfortunately, it was not possible to conduct a successful test with this triplestore due to anomalies in its behavior. Since we could not find any documentation to help us debug it, we were compelled to abort the test.

What is the current location on the experiment space? At this stage, we conducted experiments with SHARD on larger datasets. We increased the number of instances on the LIRIS cloud infrastructure. The datasets were compressed. The plan was plan to submit queries on each of these datasets. In addition, we implemented our own triplestore within the $\mathcal{CEDAR}$ project to overcome all shortcomings encountered in the triplestores we experimented with.

B Script for Splitting Dataset

Listing 1: Script for Segmenting the Dataset.

```
if [ $# == 3 ]
then
    inputFolder=$1
    outputFolder=$2
    nbFilesPerFolder=$3
    nbFiles=13140
    folder=7
    echo "mkdir -p $outputFolder/data$folder"
    mkdir -p $outputFolder/data$folder
    #copying each file in the inputFolder
    for f in $inputFolder/*
    do
#if $nbFiles == $nbFilesPerFolder means the folder is full.
#So I incremente the folder, create the new one
#and reset the number files.
        if [ $nbFiles == $nbFilesPerFolder ]
        then
            folder=`expr $folder + 1`
            mkdir -p $outputFolder/data$folder
            echo "mkdir -p $outputFolder/data$folder"
            nbFiles=1
        else
            nbFiles=`expr $nbFiles + 1`
        fi
        #moving the file into the right folder
        mv $inputFolder/$(basename $f)
        $outputFolder/data$folder/$(basename $f)
        echo "mv $1/$(basename $f)
        $outputFolder/data$folder/$(basename $f)"
    done
else
    echo "Usage: $0 inputFolder outputFoler nbFileParFolder"
fi
```

References

[1] Tim Berners-Lee, Yuhsin Chen, Lydia Chilton, Dan Connolly, Ruth Dhanaraj, James Hollenbach, Adam Lerer, , and David Sheets. Tabulator: Exploring and analyzing linked data on the semantic web. In *Procedings of the 3rd International Semantic Web User Interaction Workshop (SWUI06)*, 2006.

[2] Tom Berners-Lee. Notation 3, August 2005. [Available online[35]].

[3] Christian Bizer, Tom Heath, and Tim Berners-Lee. Linked data—the story so far. *International Journal on Semantic Web and Information Systems*, 2009. [Available online[36]].

[4] Leigh Dodds and Ian Davis. Linked data patterns—a pattern catalogue for modelling, publishing, and consuming Linked Data, 2012. [Available online[37]].

[5] Simon Marlow (editor). Haskell 2010 language report, 2010. [Available online[38]].

[6] Apache Software Foundation. [Available online[39]].

[7] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *Proceedings of the 19th ACM Symposium on Operating systems principles*, pages 29–43, Lake George, NY, USA, October 2003. ACM. [Available online[40]].

[8] SPARQL Working Group. SPARQL query language for RDF. [Available online[41]].

[9] Apache Hadoop. [Available online[42]].

[10] Jiewen Huang, Daniel J. Abadi, and Kun Ren. Scalable SPARQL querying of large RDF graphs. In Nick Koudas, Wolfgang Lehner, and Sunita Sarawagi, editors, *Proceedings of the 37th International Conference on Very Large Data Bases*, Seattle, WA, USA, August-September 2011. VLDB Endowment. [Available online[43]].

[11] Mohammad Farhan Husain, Pankil Doshi, Latifur Khan, and Bhavani Thuraisingham. Storage and retrieval of large RDF graphs using Hadoop and MapReduce. In Martin Gilje Jaatun, Gansen Zhao, and Chunming Rong, editors, *Proceedings of the 1st International Conference on Cloud Computing*, pages 680–686, Beijing, China, December 2009. LNCS 5931, Springer-Verlag. [Available online[44]].

[12] Vaibhav Khadilkar, Murat Kantarcioglu, Bhavani Thuraisingham, and Paolo Castagna. Jena-HBase: A distributed, scalable and efficient RDF triple store. Technical report, The University of Texas at Dallas, Dallas, TX, USA, August 2012. [Available online[45]].

[13] Graham Klyne and Jeremy J. Carroll. Resource description framework (rdf): Concepts and abstract syntax, February 1999. [Available online[46]].

[35] http://www.w3.org/DesignIssues/Notation3
[36] http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf
[37] http://patterns.dataincubator.org/book/linked-data-patterns.pdf
[38] http://www.haskell.org/definition/haskell2010.pdf
[39] http://www.apache.org/
[40] http://research.google.com/archive/gfs.html
[41] http://www.w3.org/TR/rdf-sparql-query/
[42] http://hadoop.apache.org/
[43] http://www.cs.yale.edu/homes/dna/papers/sw-graph-scale.pdf
[44] http://adsabs.harvard.edu/abs/2009LNCS.5931..680F
[45] http://www.utdallas.edu/~vvk072000/Research/Jena-HBase-Ext/tech-report.pdf
[46] http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[14] Ralf Lämmel. Googles MapReduce programming model—revisited. *Science of Computer Programming*, 70(1):1–30, January 2008. [Available online[47]].

[15] John McCarthy. History of lisp, February 1979. [Available online[48]].

[16] Apache Hadoop Project Members. Hadoop. [Available online[49]].

[17] Oracle. The Java programming language. [Available online[50]].

[18] Oracle. Oracle: Big data for the enterprise, June 2013. [Available online[51]].

[19] PetaSky Project. [Available online[52]].

[20] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, New York, NY, USA, October 2010. ACM. [Available online[53]].

[21] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In Michael Factor and Xubin He, editors, *Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies*, Lake Tahoe, USA, May 2010. IEEE Computer Society. [Available online[54]].

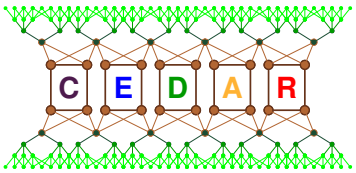
[22] Guido van Rossum. The Python programming language. [Available online[55]].

[23] Tom White. *Hadoop: The Definitive Guide*. OReilly, 2011. [Available online[56]].

[24] Paul C. Zikopoulos, Dirk deRoos, Krishnan Parasuraman, Thomas Deutsch, David Corrigan, and James Giles. *Harness the Power of Big Data—The IBM Big Data Platform*. McGraw-Hill, 2013. [Available online[57]].

Acknowledgements: The authors thank all the $\mathcal{CEDAR}$ Project's team for their feedbacks.

[47] http://www.sciencedirect.com/science/article/pii/S0167642307001281
[48] http://www-formal.stanford.edu/jmc/history/lisp/lisp.html
[49] http://hadoop.apache.org
[50] http://www.java.com/
[51] http://www.oracle.com/us/products/database/big-data-for-enterprise-519135.pdf
[52] http://com.isima.fr/Petasky/PetaSky-Mastodons.pdf
[53] http://www.avometric.com/papers/2010/Rohloff_Schantz_PsiEta_2010.pdf
[54] http://storageconference.org/2010/Papers/MSST/Shvachko.pdf
[55] http://www.python.org/
[56] http://ce.sysu.edu.cn/hope/UploadFiles/Education/2011/10/201110221516245419.pdf
[57] ftp://public.dhe.ibm.com/software/pdf/at/SWP10/Harness_the_Power_of_Big_Data.pdf



Technical Report Number 5

Experiments with Scalable Triplestores

Hassan Aït-Kaci, Mohand-Saïd Hacid, Rafiqul Haque, Damien Fourure

October 2013