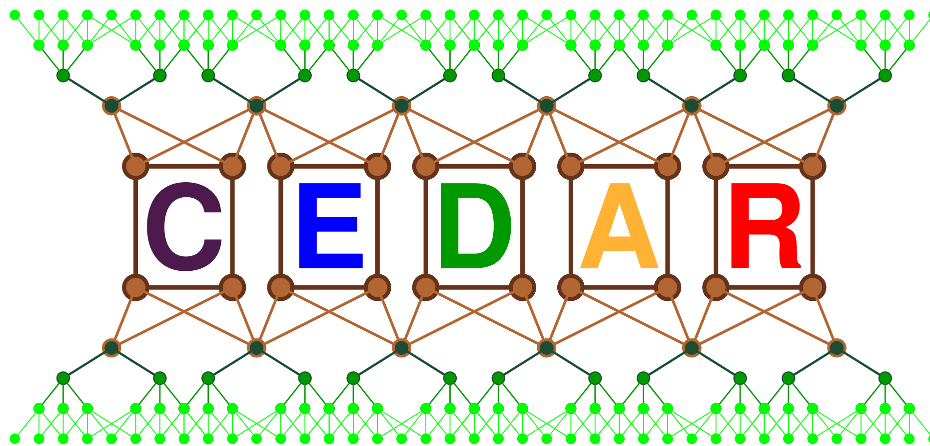


Technical Report Number 7

CedTMart

A Triplestore for Storing and Querying Blinked Data

Minwei Chen, Rafiqul Haque, Mohand-Saïd Hacid

July 2014

Publication Note

This report is based on the work done by Minwei Chen during his internship in the CEDAR Project toward the obtention of his MSc degree at the Université Claude Bernard Lyon 1, on a topic proposed by, and under the joint supervision of, Dr. Rafiqul Haque and Prof. Mohand-Saïd Hacid [10].

Contact information:

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
Email: minwei.chem@insa-lyon.fr
akm-rafiqul.haque@univ-lyon1.fr
mohand-said.hacid@univ-lyon1.fr

CEDAR Project's Web Site: cedar.liris.cnrs.fr

Copyright © 2014 by the CEDAR Project.

This work was carried out as part of the 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). It may not be copied nor reproduced in whole or in part for any commercial purpose. Permission to copy in whole or in part without payment of fee is granted for non-profit educational and research purposes provided that all such whole or partial copies include the following: a notice that such copying is by permission of the UCBL, with an acknowledgement of the authors and individual contributors to the work; and all applicable portions of the copyright notice. Copying, reproducing, or republishing for any other purpose shall require a license with payment of a fee to the UCBL. **All rights reserved.**

CEDAR **Technical Report Number 7**

CedTMart

A Triplestore for Storing and Querying Blinked Data

Minwei Chen, Rafiqul Haque, Mohand-Saïd Hacid

minwei.chem@insa-lyon.fr; {akm-rafiqul.haque; mohand-said.hacid}@univ-lyon1.fr

July 2014

Abstract

In recent years, extensive use of various devices and applications has given rise to Big Data. At the same time, Linked Data technologies are becoming popular because of the increasing trend of using Semantic Web applications. The new term "Blinked Data" combining these two concepts—*viz.*, Big Data and Linked Data—has recently been introduced. Working with Blinked Data poses enormous challenges regarding processing, storing, and querying huge and linked data sets. In addition, complex queries—such as, specifically, SPARQL queries with many joins and variables—can increase significantly the difficulty. There are solutions available, which handle Blinked Data. However, the experiments performed in the $\mathcal{CEDAR}$ project with several triplestores reveal that the current technologies are not adequately efficient. This remains an issue that must be addressed. In this report, we aim to palliate this shortcoming of the state of the art. More specifically, we attempt to overcome the challenges of storing and retrieving RDF data of size that ranges from gigabytes to petabytes. As part of the $\mathcal{CEDAR}$ project, we develop a triplestore called CedTMart which guarantees high-performance in processing complex queries. The triplestore relies on various algorithms and is built on the Hadoop/MapReduce framework to ensure scalability.

Keywords: Blinked Data, Hadoop/MapReduce, Big Data, Linked Data, Triplestore, Optimization, RDF, SPARQL

Table of Contents

1 **Introduction** 1

2 **Motivation** 2

3 **Related Work** 3
- 3.1 Fundamental technologies . . . 3
  - 3.1.1 Linked data . . . 3
  - 3.1.2 MapReduce . . . 3
  - 3.1.3 Hadoop . . . 3
  - 3.1.4 Querying with SPARQL . . . 4
  - 3.1.5 Graph partitioning . . . 5
- 3.2 Triplestores . . . 5

4 **Solution Architecture** 11
- 4.1 Preprocessing module . . . 12
- 4.2 Data distribution Module . . . 13
- 4.3 Query optimization module . . . 13

5 **Development of CedTMart** 13
- 5.1 Data converter . . . 14
- 5.2 Data partitioner and cleaner . . . 15
  - 5.2.1 Predicate Partitioning (PP) . . . 15
  - 5.2.2 Predicate-Object Partitioning (POP) . . . 16
- 5.3 Data compression . . . 18
- 5.4 Comparison . . . 24
- 5.5 Distributor . . . 27
- 5.6 Query processing . . . 30

6 **Evaluation** 30
- 6.1 Parameters . . . 31
- 6.2 Experiment . . . 33
  - 6.2.1 Results of predicate partitioning . . . 34
  - 6.2.2 Results of predicate-object partitioning . . . 34
  - 6.2.3 Results of compression . . . 35
- 6.3 Results of comparison . . . 38

7 **Conclusion and Future Work** 40

1 Introduction

This technical report is about a scalable and high-performance triplestore that enables storing and querying Big Linked Data which has been termed Blinked Data in [3]. It has been defined as *the intersection of two prominent concepts: Big and Linked Data* [18].

Resource Description Framework (RDF) [11] is a widely used technology for modeling Linked Data. Originally designed as a technology for modeling metadata, RDF—a member of the World Wide Consortium (W3C) family—models Linked Data in the form of subject-predicate-object.[1] It provides a mechanism to denote resources and relationships between triples [9]. The RDF data are better known as RDF triples. SPARQL (SPARQL Protocol and RDF Query Language)[2] is a language provided by the W3C to perform queries on RDF triples. It is one of the most widely adopted technology for retrieving and manipulating RDF triples. Together with RDF, SPARQL is used for building the Semantic Web (SW) applications.

In recent years, extensive use of various devices (*e.g.*, sensors, smart phones, and tablets) and applications (*e.g.*, Facebook and Twitter have given the rise to the notion of Big Data [14] which is massive in size and has a wide variety.[3,4] Processing queries efficiently on Big Datasets is a well-known problem. Additionally, data today are not only Big, they are Linked as well. This makes *data processing* and *query processing* far more challenging.

The SPARQL and RDF-related technologies have a plenty to offer to the overlapping worlds of Big Data and NoSQL [17]. A significant number of researchers are focusing on these areas and many works have already been done. However, current technologies more specifically, the triplestores are yet to be sufficiently powerful. As a result, they do not address the challenges we mentioned earlier.

A limited number of triplestores is available for processing and querying Blinked Data. Some of these are built on Hadoop/MapReduce framework[5] which has drawn a huge media attentions in recent years. With the power of Hadoop/MapReduce, some triplestores are capable of handling scalability. Hadoop is a *de facto* technology for building scalable applications. It enables running applications on commodity hardware and thus it is easier to scale-up the computation infrastructures. However, performance of applications is beyond the scope of Hadoop. This project aims to address the shortcomings of existing technologies.

This research project is a fragment conducted within the Constraint Event Driven Automated Reasoning ($\mathcal{CEDAR}$) research framework. In the context of Blinked Data and as part of the $\mathcal{CEDAR}$ project, the objective of this research project is to develop a triplestore called *CedTMart* (stands for *$\mathcal{CEDAR}$ Triple Mart*). The key purpose of this triplestore is to enable storing and querying Blinked Data efficiently by guaranteeing

[1] http://www.w3.org

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

[3] https://www.facebook.com

[4] https://twitter.com

[5] http://hadoop.apache.org

high-performance. The nature of the queries we consider in this project is conjunctive and contains several variables. The CedTMart is designed to deal particularly with huge datasets whose size ranges from gigabytes to petabytes. Additionally, the triple-store is developed to serve as a distributed and scalable storage that can be deployed on a large number of commodity hardware such as, clusters on cloud infrastructure.

The report is organized as follows. In Section 2, the motivation behind this research is presented. Section 3 discusses the state of the art. In Section 4, our solution architecture, CedTMart, is presented. Section 5 describes implementation issues. In Section 6, we report results of experiments we conducted with our design. A conclusion is drawn in Section 7, along with some perpectives of future work.

2 Motivation

The Linked Data concept is developed to make the web easier to manage (accuracy of the information returned by the queries, reasoning, elimination of redundancies, *etc.*.). The advent of various technologies have enabled to develop intelligent applications with ability to reasoning data. Also, the technologies specifically RDF seems promising to solve the well-known *data silo* problem. It can simplify the traditional way of connecting tables in Relational Database Management System (RDBMS). These have drawn a huge attentions recently and thus, many large enterprises (e.g., Facebook, Twitter, and LinkedIn) have adopted Linked Data technologies.[6]

The increasing trend of adopting Linked Data has promoted challenges of managing and querying Blinked Data. The size of current datasets is increasing dramatically. For instance, Facebook generates more than 7 Terabytes (TB) data everyday [14]. Such large-scale organizations are looking for massively scalable, high-performance and robust triplestore technologies. This implies that the state-of-the-art is missing a solution with optimized algorithms which is strongly required to manage jobs such as, *query processing* on such huge size datasets.

The $\mathcal{CEDAR}$ team had tested a few of existing open source triplestores.[7] The tests unfolded some important issues about those triplestores and their performance. The triplestores are built on widely known Hadoop/MapReduce technology. The study concluded that some of these triplestore handle scalability issue successfully, however, their performance remains an optimization problem that has yet to be addressed. The results show that the triplestores are not sufficiently capable to process complex queries on Blinked Datasets within a reasonable time. The study found that of all these triplestores some lack efficient storing techniques which essentially severely affect the performance. These drew our attention to the research towards developing a high-performance and scalable triplestore.

[6] https://www.linkedin.com

[7] These results are published in [3].

3 Related Work

The technologies related to this research revolve around distributed and parallel computing. Also, it involves Linked Data. This section provides comprehensive details of the technologies related to this research.

3.1 Fundamental technologies

This section is a review of the fundamental technologies used so far in Blinked Data applications.

3.1.1 Linked data

Linked Data is simply about using the Web to create typed links between data from different sources [4]. It is a means of publishing "web-native" data using standards like HTTP, URIs and RDF [15]. Bizer *et al.* [8] 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 initiative was made by the W3C to transform the web from unstructured heterogeneous data to a semantic representation, in order to link information from different web pages.

3.1.2 MapReduce

First proposed in 2004, MapReduce [13] is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function which merges all intermediate values associated with the same intermediate key.

A new model called Map-Reduce-Merge [30] introduced three years after the concept of MapReduce added a *Merge* phase. This new phase merges efficiently data already partitioned and sorted/hashed by the map and reduce modules. This was to overcome some limitations on heterogeneous datasets.

Then, in 2012, a new language calls RDFPath [26] was conceived. It follows a syntax similar to XPath for XML that can be used for selecting the values of properties. This query language transforms automatically path queries into MapReduce jobs and makes use of scaling properties of the MapReduce framework.

3.1.3 Hadoop

This Hadoop framework is composed of the several modules. Currently it has Hadoop Common, which is the core component of the framework; HDFS (Hadoop Distributed File System); Hadoop YARN which is a job scheduler and resource manager for clusters; and MapReduce, which is a concrete programming model for large scale data

processing. This Apache project develops an open-source platform for reliable and scalable processing of large data in distributed environments. Hadoop itself has been designed to automatically detect and handle failures at the application layer. Therefore, it can guarantee highly-available services on top of relatively uncertain clusters and networks.

Yahoo Inc. is a primary contributor of Hadoop's core architecture (MapReduce implementation and HDFS [27]).[8] The HDFS is designed to store large data sets reliably and to stream them at high bandwidth to user applications in large clusters. By distributing storage in multiple data nodes and sharing computation across them, the resource can grow with demand. This then becomes a viable solution for storing RDF datasets.

As mentioned in previous section, the combination, RDF Data using MapReduce can provide better scalability, fault-tolerance and compatibility for newer Web-based and mobile applications. The Hadoop/MapReduce has been used in building several large-scale triplestore for processing and querying Linked Data. Some these are open source including SHARD [27], HadoopRDF [21] and others are commercial products.

3.1.4 Querying with SPARQL

The extensive use of semantic web and large RDF datasets pose significant challenges for the efficient storage and retrieval of RDF graphs. In the last few years, several algorithms and frameworks have been proposed or implemented to address these challenges. Splitting or partitioning are often used to fit RDF data into distributed environments. Various graph-based models have been developed to optimize query execution from different aspects. We describe some predominant query processing technologies in this section.

The concept of BitMat, a bit matrix structure for efficient querying over graph databases, was proposed in 2008 [5]. Although previous works such as vertical partitioning have already proposed some database storage and query optimization techniques, there was still a challenge for queries having low-selectivity triple patterns, scalability of the querying method, and optimizations. The main idea of BitMat is a new way to store RDF graphs : to transform RDF datasets into a compact in-memory storage and to use some bitwise operations to obtain a faster processing of queries, comparing to the conventional RDF triple stores. The authors have also published several related works including the detailed implementation and processing mechanisms [5] and a query processing algorithm which is also based on compressed bit-vectors [4].

DARQ [22] was proposed in 2008, which is an engine for federated SPARQL queries. It overcomes the large overhead in network traffic. It allows to query one single RDF graph despite the real data being distributed on the web. Using a service description language, the engine can decompose a query into sub-queries. Each of these sub-queries can be answered by an individual service. Also, DARQ uses query rewriting and cost-based query optimizations to speed-up query execution.

Another approach [20] where an iterator-based pipeline has been used to perform par-

[8] https://fr.yahoo.com/?p=us

allelized SPARQL queries. This approach discovers relevant data with URIs resolved over HTTP, with an extension to avoid abnormal long delays during the URI prefetching in each execution. Since a query is represented by a tree structure, a tree of iterators has been used for a number of input graph patterns. Eeach iterator returns solution mappings (solutions for the set of triple patterns assigned to it) to its predecessors.

3.1.5 Graph partitioning

Efficient management of RDF data is an important factor for realizing the vision of SW. Large-scale RDF datasets on a single machine do not scale well. Thus, distributed technologies are used to distribute RDF triples to multiple machines. However, due to inefficient dataset partitioning techniques used by the existing solutions, the performance of distributed triplestores is significantly affected.

The idea of partitioning RDF data vertically was proposed in [1]. The authors proposed to use RDBMS to store the re-structured (split, sorted and indexed) RDF data. In this approach, each table is sorted by subject. Therefore, it is possible to locate them faster. This alternative solution has an extended idea for column-oriented architectures such as MonetDB,[9] and works better with optimized code and data compression.

Another triple indexing scheme was proposed in [30], which considers RDF data as a graph, partitions the graph into multiple sub-graph pieces, and store them individually. Over these partitions, a signature tree is used to index the URIs. When queries arrive, this index can locate all partitions with high speed. This may include the matching of the queries by their constant URIs.

Also, researchers have discovered a promising approach reported in [29]. It employs the nature of graphs to minimize the relations between partitions. It optimizes system design based on the relations to reduce communication cost of query-processing messages, balance size of partitions, and enhance parallelism through independent sub-querying.

3.2 Triplestores

With fast-growing cloud services in the market, more and more enterprises move their data nodes inside virtual infrastructure. Especially, in the production environment where people need reliable, high-performance and scalable solutions which allow thousands of machines working in parallel to perform data queries on demand.

In RDF triplestores, one retrieves stored data that relate objects to others via the SPARQL query language. Today, there is a list of implementations that provide the RDF triplestore functionalities, including Apache Jena,[10] AllegroGraph,[11] Oracle RDF, Sesame,[12] *etc.*, ... Of all open source triplestores, two highly rated were experimented

[9] https://www.monetdb.org
[10] http://jena.apache.org
[11] http://franz.com/agraph/allegrograph
[12] http://www.openrdf.org

within the scope of the *CEDAR* project. This section discusses some notable triple-stores.

SHARD

SHARD[13] 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-based repository which caters for building a distributed and parallel environment for storing and querying RDF data. Since Hadoop allows any number of worker nodes, scaling up the computations should not be problem. In other words, the number of computational node could in principle be increased without degrading performance.

In SHARD, data are persisted in flat files in HDFS clusters. Each line in files represents a triple. Queries are processed in an iterative manner. The iterative query processing has been used to improve conventional MapReduce functions. In particular, it enables incremental query processing to bind variables while satisfying the 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, binds the clause variables, and lists all the variable bindings. Then, in the the subsequent step it reduces the list of matched triples where duplicate $\langle \texttt{key}, \texttt{value} \rangle$ pairs are deleted. The following 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 onto 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.

HadoopRDF

HadoopRDF [21] is another Hadoop-based triplestore. The main focus of HadoopRDF is to optimize queries on Blinked Data. 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 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

[13] http://hadoop.apache.org

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.*, Notation3[14]) 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. These two features play a significant role in compressing the dataset without needing any CoDec.[15] They may be viewed as a particular kind of indexing on triples.

The Predicate-Split (PS) 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 (POS) 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 operation is called Predicate-Object Split of Non Type (POSNT).

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.

AllegroGraph

AllegroGraph[16] is a high-performance RDF graph database system. It is designed for Semantic Web applications to process big RDF datasets with meta-data.

AllegroGraph allows two RDF serialization formats: RDF/XML and N-Triples. It stores graph information of triples having the same context. More concretely, AllegroGraph stores data in the subject, predicate and object triples with added graph fields and more quality. For example, consider a triple where one node S is connected to another node O via the edge P with additional data G.[17] Once data stored in AllegroGraph, it is automatically indexed and then the SPARQL queries are carried out through an API provided by the framework. A hash-based partitioning technology is

[14] http://www.w3.org/TeamSubmission/n3

[15] This stands for Compression and Decompression).

[16] http://franz.com/agraph/allegrograph

[17] One might then argue that it is a *quadruple* rather than a *triple*. However, it is a triple in that $\langle S, P, O \rangle$ has a specific standard meaning, while G is optional and has no specified structure nor semantics other than being "additional information" appropriate to whatever use to be made of it—such as comments, annotations, *etc.*, ...

used for stored triples. Thus, the query processing engine does not need to perform MapReduce operations.

RDFS++ is an ontology engine provided by the AllegroGraph framework to dynamically maintain entailment for reasoning without the materialization phase. The materialization is a pre-processing phase to compute and store inferred triples for a better efficiency of future queries. However, it is difficult to maintain. By doing this, AllegroGraph simplifies the maintenance of stored ontology information and reduces time to perform queries on modified data.

It is important to note that AllegroGraph is more flexible and scalable. It encodes values directly into its triples, so that new predicates and 1-to-N relations can be easily inserted by users without significant changes of existing schema. This could be suitable for Web-based open systems in future. According to a list recorded by W3C,[18] AllegroGraph has announced in August 2011 that it was the first commercial product which could load and query for 1 trillion RDF triples, with the help of a supercomputer infrastructure.

Apache Jena

Apache Jena[19] is an open source framework for Semantic Web and Linked Data applications. It provides various modules to process RDF data. External applications can interact with it directly using provided rich Java API or via HTTP layer.

RDF API is a core component for reading and storing RDF data. There are two primary concepts:

- *Model*—this is the main container API for RDF information in graph form. It has multiple methods. They are: *readers*, *writers* and *iterators* which allow to create RDF-based applications. The other methods are *in-memory* and *secondary storage* data persistence to manage resources.
- *Graph*—this is a simpler abstract interface with low-level RDF stores which is lighter to use and easier to re-engineer.

AR[20] is a SPARQL query processor with a Java application API. *Query* is a main class contains different parser methods and all the details of parsed SPARQL queries. *QueryExecution* calls execution functions then returns query solutions.

Jena provides a component called TDB for storing triples.[21] It can be used for high-performance RDF storage and query on single machines. TDB can be managed by command line scripts or via the Java API. It fetches TDB-graphs or RDF datasets and creates Model or Dataset objects that can directly be queried or used by Jena API. TDB enables concurrency and transactions with an integrated execution optimizer. The SPARQL queries are transformed algebraically before execution and executed

[18] http://www.w3.org/wiki/LargeTripleStores
[19] http://jena.apache.org
[20] http://jena.apache.org
[21] http://jena.apache.org/documentation/tdb

by a dynamically calculated plan. Fusek1[22] is another component of Jena designed as a SPARQL server that uses TDB for persistent storage and provides the SPARQL protocols for queries, normal updates and REST updates over HTTP.

HadoopDB

Hadoop is well-known because it enables building scalable, fault-tolerant and flexible parallel data processing applications. In [2], the authors discuss the feasibility of HadoopDB [2], which comprises DBMS and MapReduce to run on low-end commodity machines or in clouds. Introduced in [2], HadoopDB is an approach for parallel databases on top of MapReduce-based systems. It extends the Hadoop framework and adds a few new components and concepts.

HadoopDB comprises of four components. The *Database Connector* interface between the compute nodes to communicate with each other. The connector supports several DBMS like MySQL and PostgreSQL. Represented by a MapReduce job, a connector connects to JDBC-compliant database instances on nodes, launches individually optimized queries and returns result sets as $\langle \texttt{key}, \texttt{value} \rangle$ pairs. *Catalog* is a of meta-data storage. It provides necessary information to the connector, and contains information about the distribution of datasets in clusters for example the locations of data partitions. The *Data Loader* is responsible for data distribution to nodes, data partitioning on nodes, and bulk-loading the single-node databases with partitioned chunks. The hasher components are used for assigning MapReduce jobs to Hadoop. Finally, the *SQL Planner* enables to process SQL queries. It extends Hive,[23] an open source data warehousing infrastructure on top of Hadoop, to enable MapReduce jobs connect to tables stored as separate data files in HDFS.

Since HadoopDB uses clusters of single database systems instead of HDFS to store datasets, it yields benefits of performance advantages. With enforced query execution approaches published in another paper [6]. Currently, HadoopDB provides better performance and could be a good reference of our future benchmark.

RDF3X

RDF3X [23] is an RDF storage and retrieval system. It is an RDF triplestore with automatic indexing. It comprises a query processor and a query optimizer which optimizes queries based on a cost model.

In this triplestore, triples are stored in a sorted B+Tree to facilitate faster query processing. A dictionary is used principally for long literal expressions and is built as a mapping between concrete *literal data* and unique numerical *ids*. Once the triples are compressed to ids only, the cost of dictionary indexes is considerably reduced, and query execution can be faster simpler with the ids. The numerical results of queries are transformed back into literal expressions and output to users. This is similar to the underlying principle of our CedTMart triplestore.

[22] http://jena.apache.org/documentation/serving_data

[23] https://hive.apache.org

RDF3X generates six indexes: SPO, SOP, OSP, OPS, PSO, POS which are maintained to satisfy all possible permutations of queries. It is worth noting that the size of these indexes are tolerable because they are built on numerical *ids*.

H2RDF

H2RDF+ [25] enables performing distributed queries especially merged and sorted joins over a multiple indexes. It us built upon MapReduce programming model. The indexes in H2RDF++ are built inside HBase.[24] Like RDF3X, six indexes of S, P and O are maintained in H2RDF for operations such as range index scan, but in a different way. As HBase uses a key-value model, indexes store all triples in keys and leave the values empty. Indexes are also id-based. Therefore, HBase is used as dictionary. What is more, an aggregated indexing approach is designed.

In H2RDF+, there are two categories of aggregated indexes, with two bound elements (sp_o, ps_o, po_s, op_s, os_p and so_p) and with one bound element (s_po, p_so, p_os, o_ps, o_sp and s_op). These additional statistics are used to estimate the cost and the size of the result set of a query.

H2RDF+ allows MapReduce-based bulk-import jobs to load and index large RDF datasets, which can be practical to handle existing RDF datasets. It creates multiple *sub-jobs* to create statistics of RDF literal data, assign ids to stored literal values, create mappings between literal data and ids, store literal string values in partitions, and create corresponding indexes into HFiles that can directly be loaded by HBase. Additionally, the queries containing multiple triple patterns are guaranteed by merge join algorithms exploiting pre-ordered indexes.

In H2RDF+, the cost-based query planner decides on the execution order of different join operations to minimize the total execution time. HBase scan performance evaluation, Merge join cost model, Sort-Merge join cost model and Join Planner are four algorithms integrated for executions.

PigSPARQL

PigSPARQL [28] is a SPARQL query processing system and a translation framework based on SPARQL and Pig Latin [24], a new data analysis language developed by Yahoo Research for processing large datasets. Pig Latin for Hadoop is an open source Apache-incubator project which allows users to simplify and customize high-level codes that reside above the MapReduce programs. In PigSPARQL, tasks are executed by MapReduce jobs on a Hadoop cluster. The authors defined a series of translation rules to represent RDF triples in Pig Latin. Also, an optimization strategy is provided for translated SPARQL queries.

[24] http://hbase.apache.org

YARS2

YARS2 [19] is another triplestore which contains a query engine. It enables querying over graph data with a compressed indexing framework optimized for read operations. It provides a data partitioning mechanism to distribute indexes which optimize parallel queries. Apache Lucene[25] is used for keyword indexing for keyword lookups, with *quad indices* for atomic lookups in graphs and *join indices* to search combinations of paths in graphs.

YARS2 demonstrated its ability to process 7 billions of synthetically generated statements. The join-processing algorithm gives a correct reasonable overall performance, but a bottleneck of the throughput for index scan can be noticed.

Virtuoso

Virtuoso [16] is one of the top-rated triplestores. It is a multiple degree data server product allows to manage RDF/XML datasets with dedicated Web service supports (SOAP and REST). Virtuoso has a query engine providing native RDF support, bitmap indexing and SQL query optimization. It also allows to directly map relational data into RDF and perform join between RDF data and relational data. This aspect is important for enterprises to deal with existing data.

OpenRDF Sesame

OpenRDF Sesame is a complete RDF-based Semantic-Web framework.[26] It includes RDF parsers, writers, triple storage, reasoning, and querying. The OpenRDF Java API enables connecting all leading RDF storage solutions. Several plug-ins and extensions are available for different purposes. For example, the connection between Sesame and Virtuoso, the support of PHP programs, the Big Data API, *etc.*, . . .

Oracle RDF

Oracle Spatial and Graph [12] is a feature in the newer versions of Oracle Database. It provides an RDF data management platform with complete API. For example, an application using Jena API can manipulate graph-structured data stored in an Oracle database.

4 Solution Architecture

This section presents the architecture of our triplestore. CedTMart comprises three modules: *Pre-processing module*, *Data Distribution module*, and *Query Optimization module*. Figure 1 presents a high level architecture of CedTMart. These modules carry

[25] http://lucene.apache.org/core
[26] http://www.openrdf.org

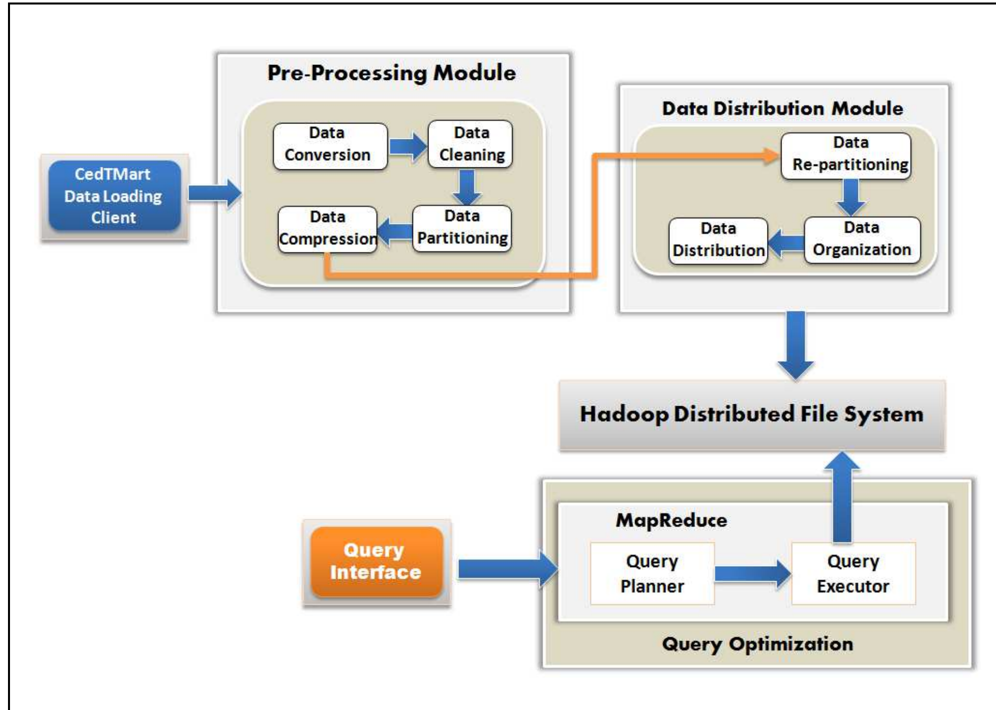


Figure 1: The Architecture of CedTMart

out several functions: a) converting RDF data from any format to Notation3 (N3) format b) cleaning RDF triples, c) compressing datasets, d) distributing data, and e) processing and optimizing SPARQL queries.

4.1 Preprocessing module

This module resides at the top layer of the CedTMart architecture. It carries out five functions: *convert*, *read*, *clean*, *partition*, and *compress*.

- It converts RDF triples to N3 triples [7], which is a shorthand non-XML serialization of RDF data model and more compact and readable than RDF notation.
- It reads triples from converted N3 files.
- It validates triples with respect to the syntax recommended by the W3C and stores invalid tokens in a specified folder.
- It carries out three partition-related functions:
  - *predicate partition (PP)*,
  - *predicate-object partition of type (POPT)*, and
  - *predicate partition of object non type (POPNT)*,

  by storing partitioned data in specified files.
- It compresses data using `D-Gap Compression` method.

We use multi-threading technique to perform these tasks efficiently. It is a suitable technique which enhances the I/O performance.

4.2 Data distribution Module

In pre-processing module, data are partitioned to enable efficient query processing on Blinked Data sets. In this module data are re-repartitioned. The key purpose re-paritioning data is to slice a large files into smallers ones and then distribute them intelligently within clusters that contain hundreds of nodes. The data distribution relies on *edge betweenness* technique. The core of this notion has essentially been borrowed from the technique called the *edge centrality* which was used within the social network analytic domain. Our edge betweenness algorithm measures the closeness between the predicates. It is measured by comparing the common subjects and objects between two predicates.

4.3 Query optimization module

This is a critical module of CedTMart. It comprises two components: *query planner* and *query executor*. The query planner has two main parts. The first part deals with query partitioning and the second part deals with sequencing the order of execution. For query partitioning, we have devised a query plan. The plan is straightforward: decomposing (SPARQL) queries based on the number of variables associated with the triple patterns. Note that, a SPARQL query can be represented a graph. For a given query, the CedTMart reads the triple patterns within WHERE clause and decomposes the query into multiple subgraphs. Then, the order of executing subgraphs is defined. We call this *query flow*. The execution order of subgraphs is determined by number of variables a subgraph contains. For instance, the subgraphs with '0' (or no) variable is executed first followed by the subgraphs with '1' variable and then 'n' variable. The execution of subgraphs is continued until all subgraphs (with n maximum variables) are executed. Additionally, the query planner defines the order of executing queries within the subgraphs. In this case, the order is determined by the weight of the vertices that compose the triple patterns of the given query. The *query executor* realizes the order of query execution and execute them accordingly.

5 Development of CedTMart

CedTMart's components perform various tasks which follow a sequence. Figure 2 shows the flow of tasks that are carried out in Preprocessing and Data Distribution modules. The ongoing Query Optimization module is not shown in the Figure.

There are many technologies that we could use to implement different components of CedTMart. However, in our case, the main constraint to deal with a huge amount of data is the efficiency. The initial idea was to use not only JAVA functions (standard

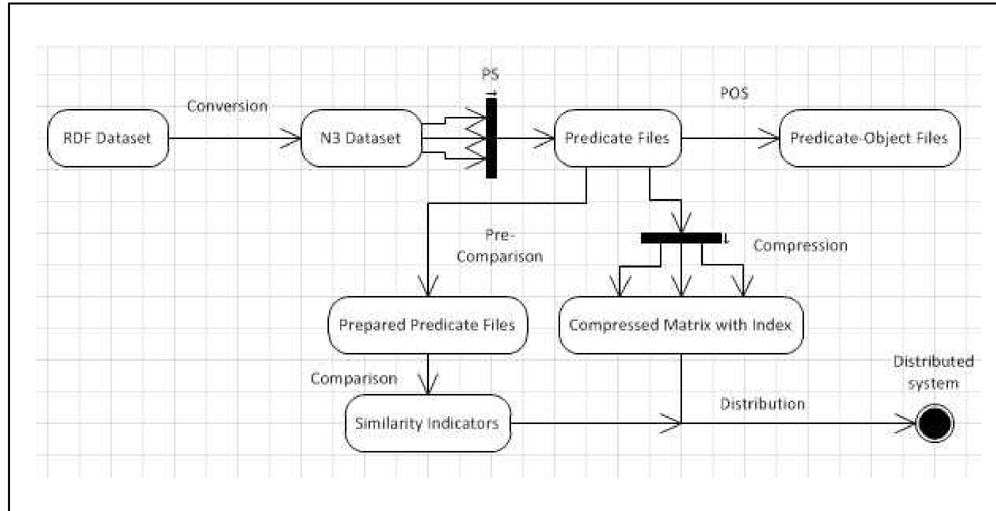


Figure 2: Flow of tasks performed by the triplestore

and external libraries) but also other scripts such as Perl and Shell.[27] In addition, we found some existing database systems which can be used for implementing different functionalities of CedTMart. Using database system would enable to sidestep using main memory. It would enable to use external memory buffer. In this regard, our solution architecture is sufficiently flexible. It allows users to make a choice between all proposed alternatives and can satisfy different constraints. For instance, if a user has systems with huge size memories, he can choose memory-intensive alternative. Conversely, if a user has machines with insufficient memories, he can process his jobs using database technology which would unfortunately be time-consuming. So it is a tradeoff that has to make by the users. A goal of our implementation is to find and use a technique that assists to implement suitable method for each sub-task in different situations, for instance, multi-threading.

As mentioned in last sections that we use the multi-threading technique to enhance the processing performance. We have implemented a task assignment mechanism to put the input source into equal parts for each thread. This is discussed later in this section. We discuss the components of CedTMart in the following sub-sections.

5.1 Data converter

Of all RDF serialization, Notation3 (N3) [7] is not only human-readable but also convenient for semantic text parsing and thus, is more suitable as an input for processing a MapReduce job than other formats (*e.g.*, RDF/XML) [11]). Therefore, we chose N3 format to store data.

There are different formats for presenting RDF data such as RDF/XML, Turtle,[28] N-

[27] http://www.perl.org
[28] http://www.w3.org/TR/turtle

Triple.[29] CedTMart supports several extensions including `.rdf`, `.owl`, `.xml`, `.rdfs` *etc.*, ... It is able to accept input data with these formats yet, since N3 is the native format for storing triples, the data are converted to this format. A converter called *RDF2RDF*[30]—an open source third party library—has been integrated with the CedTMart for performing conversion. However, we have implemented different techniques to optimize data conversion performance such as multi-threading. Since data files can be represented independently by one N3 data file, CedTMart takes the advantage of using multi-threading to parallelise the conversion process. Each thread instantiates a `FormatConverter` class to interface with the environment in which the application is running, then to run a pre-compiled executable jar file of RDF2RDF.

5.2 Data partitioner and cleaner

CedTMart provides a partitioner that performs two types of partitioning namely, *Predicate Partitioning (PP)* and *Predicate-Object Partitioning (POP)*.

5.2.1 Predicate Partitioning (PP)

In this step, N3 data is partitioned into predicate files. Currently, the number of distinct predicates in different ranges from 20 to 30. This indicates partitioning data by predicates can significantly reduce the search space for SPARQL queries which have non-variable predicates such as, `<?x isStudent ?y>`. The CedTMart partitioner splits the files by predicates and it replaces the characters that are illegal for naming files by unique characters [11]. For instance, all triples containing the predicate `p:isStudent` are merged into a single file `p-isStudent`, in a folder called `_ps`.

CedTMart relies on parallelism and thus uses multi-threading. Each thread instantiates a `DataManager` class, where a function called `N3Reader` parses each file (*line by line*) of a set of N3 data files into `CTMTriple`. Then the ⟨`subject`, `object`⟩ pairs of the triples are written into predicate files that are located in a directory (by default it is `_ps` directory).

CedTMart provides a *Data Cleaner*. It is integrated with the `N3Reader` which automatically emits the lines with invalid entries to an output directory (by default it is `_invalidTriple` folder). Figure 3 and Figure 4 show the flowcharts of the operations.

This leads to a join-like operation. Our approach creates a set of writers: each writer results in ⟨`subject`, `object`⟩ pairs of one predicate to the predicate's file. During the development and test phases, we have found a few interesting issues about the technologies we used:

- `java.nio` is a buffer-oriented technology. Therefore, it is used either to *access memory directly* or to *block mediated bulk data transferring*. It does several other tasks. However, those are related to blocking and non-blocking access. As such, if we want to use NIO to grab the data quickly (or in a non-blocking

[29] http://www.w3.org/2001/sw/RDFCore/ntriples
[30] http://www.l3s.de/ minack/rdf2rdf

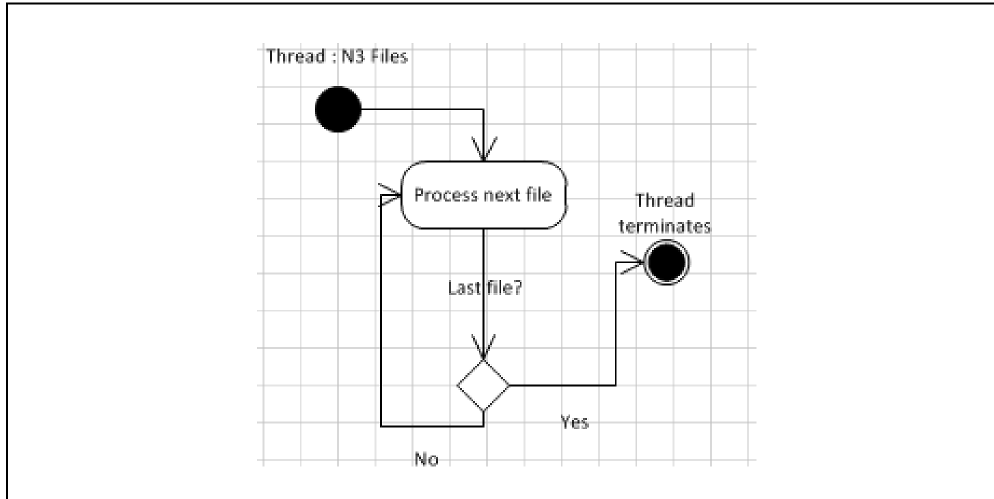


Figure 3: Procedure of PS

manner), it is a feasible choice. However, in our case, we read *line by line* and process at the *same time*. Therefore, it would be better if the lines were detected after NIO had finished reading the available data. In fact, we do not need to put a line reading *facade* over the buffer that NIO just read, the I/O can already provide a nice overall throughput.

- Based on a series of tests, we concluded that `BufferedWriter` is a better choice for writing sequential records.
- The `writer.close()` is computationally expensive method. Its too many close instructions degrades the performance. `BufferedWriter` could be an option to resolve this bottleneck. The key task of `BufferedWriter` in this case is to consolidate a large number of small writes into fewer large writes. Although it is efficient, but painful to implement.

Considering these issues, we choose to use `HashMap` ⟨`String, BufferedWriter`⟩, where a `String` acts as a unique key for each distinct predicate's file name. In this way, each `BufferedWriter` is assigned to one distinct predicate parsed from a thread. Figure 5 shows how each parsed ⟨`subject`, `object`⟩ pair *line* is put to the buffer of a `BufferedWriter` which is assigned to a predicate's `filepath`.

5.2.2 Predicate-Object Partitioning (POP)

CedTMart's partitioner carries out predicate object partitioning. The predicate partitioning results in predicate files. For queries having non-variable predicates, the corresponding predicate files are retrieved first and then queries are performed on the files. However, for those that have variable predicate, all files must be scanned, which will consume a significant amount of time. The CedTMart adopts an efficient approach which was found in [11]. This approach determines the types information of objects

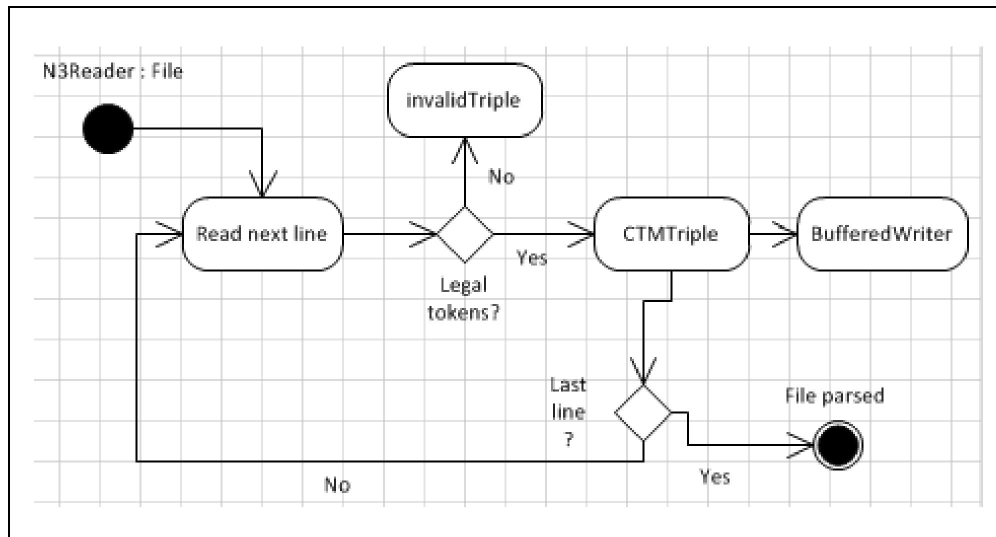


Figure 4: Procedure of `N3Reader` for each file

```
private void writeToBigFile(String filepath, String line)
        throws IOException{
    BufferedWriter writer;
    if(!_predicateWriterList.containsKey(filepath)){
        writer = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream(filepath,true),"UTF-8"));
        _predicateWriterList.put(filepath, writer);
    } else {
        writer = _predicateWriterList.get(filepath);
    }
    writer.write(line);
    writer.newLine();
}
```

Figure 5: Function `writeToBigFile`

and add them in the file names. We called this called *Predicate-Object Partition (POP)*. CedTMart can perform two types of object partitioning:

- *Predicate-Object Partition of Type (POPT)*: In POPT, CedTMart partitions files by *rdf-type* of the predicate `rdf:type` into files `rdf-type_`c_1, `rdf-type_`c_2, ..., `rdf-type_`c_n, where c_1, c_2, ..., c_n are the objects that appear in the files `rdf-type`. Each of these files `rdf-type_`c_1, `rdf-type_`c_2, ..., `rdf-type_`c_n has only one column which stores subjects of triples having the predicate *rdf-type* and the object c_n. The purpose of processing the predicate only is to fetch easily the class hierarchies in ontology applications. It is straightforward because the leaves are directly stored in the file names. This has an important advantage. It frees storage space by combining the repetitive objects into a single entry in file names.
- *Predicate-Object Partition of Non-Type (POPNT)*: For other predicates, instead of splitting them by distinct objects, our triplestore classifies them only according to the type of objects: `Variable`, `Anonymous`, `IRI` and `Literal`. This classification is important for two reasons: it helps to avoid a large amount of possible distinct objects in modern day's datasets and it narrows down the search spaces. A predicate file *pred* splits into maximum four files, `pred_Variable`, `pred_Anonymous`, `pred_IRI` and `pred_Literal`, if it contains all of these four types of objects. The POPNT method essentially promotes a compromise between the number of files and the search space in each file.

Predicate-object partitioning of type operations are simple *read-and-write* operations. The partitioner reads each predicate file line by line, fetch objects by checking its type, then create and write lines into different files. For implementation, we used `writeToBigFile()` function. However, the loop function parses each predicate file using the `SOReader()` in place of the `N3Reader` because each line in a predicate file contains only two entries: *subject* and *object*. Since invalid triples are already eliminated in PP step, the `SOReader()` stores directly parsed lines into `CTMDouble`, which is composed of a `CTMSubject` and a `CTMObject`, to store their *type* information and retrieve them while writing into the destination files. Figures 6 and 7 demonstrate the detailed procedures of predicate-object partitioning and reading the subjects and objects of triples in files.

5.3 Data compression

The predicate files contain lines with two entries: *subject* and *object*. Our goal is to optimize the query processing performance. It is worth noting that, in addition to simple SPARQL queries, there are complex queries such as *conjunctive multi-join queries* which can cause very large result sets, if the data set is huge. Taking this issue into account we invested a significant amount of time to find an efficient approach. We found *BitMat* [5] that can transform datasets into a compact format, load them onto main memory then perform bitwise operations on them. Our idea was to minimize the size of datasets so that data can be load onto memory instead of secondary storage.

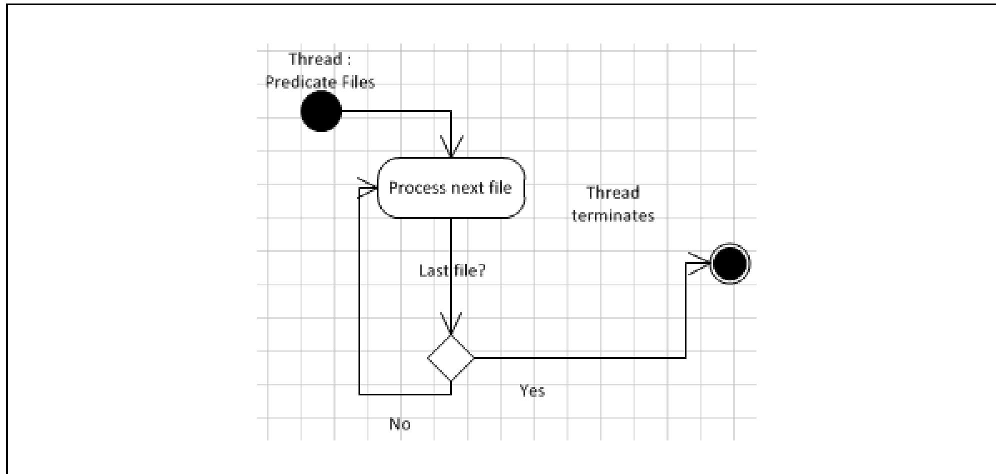


Figure 6: Procedure for Predicate-Object partitioning

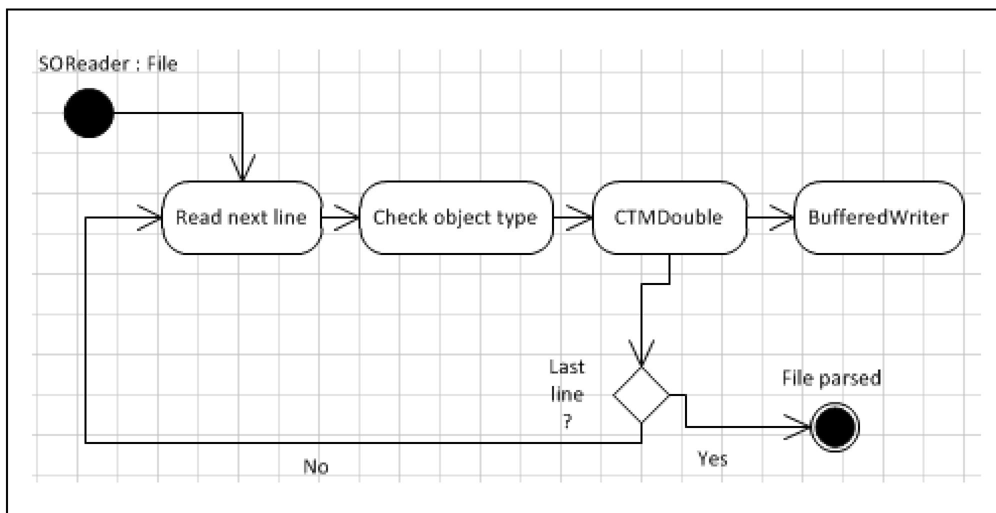


Figure 7: Procedure of Subject-Object `Reader(SOReader)` for each file

BitMat is a compressed inverted index structure for the in-memory storage of RDF/N3 triples. Generally, it builds a three-dimensional (3D) cube of subject, predicate and object. In fact, each predicate is represented by a two-dimensional (2D) matrix of distinct subjects and objects and the line numbers of each dimension corresponds to an *unique literal/concrete* entry stored in an index file (see Figure 8).

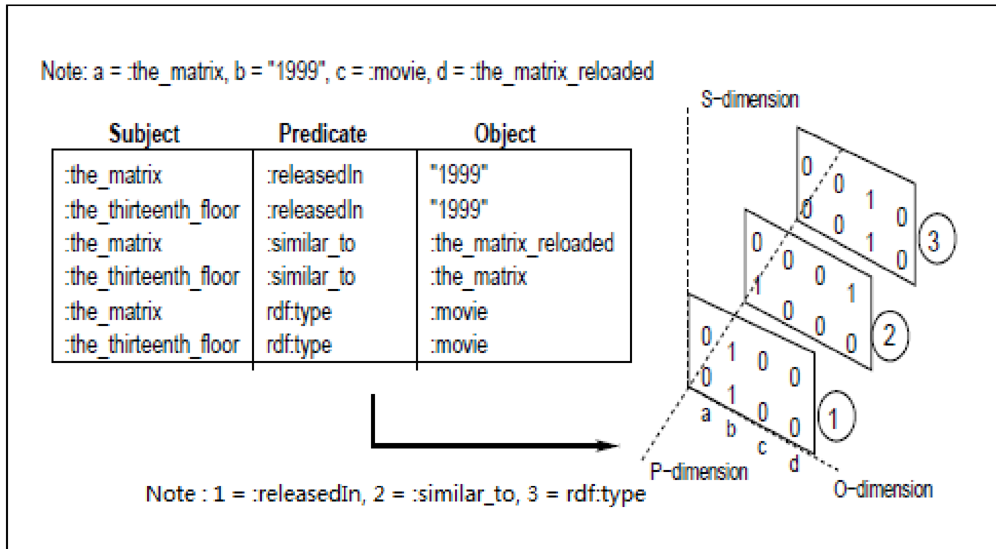


Figure 8: BitMat representation

Knowing that we have queries with variable subjects and variable objects, instead of concatenating a single matrix of ⟨`subject`, `object`⟩ pairs, we also store a transposed matrix if ⟨`object`, `subject`⟩ pairs for each predicate. These two matrices create PSO and POS indexes. Figure 9 shows an example of these two index matrices.

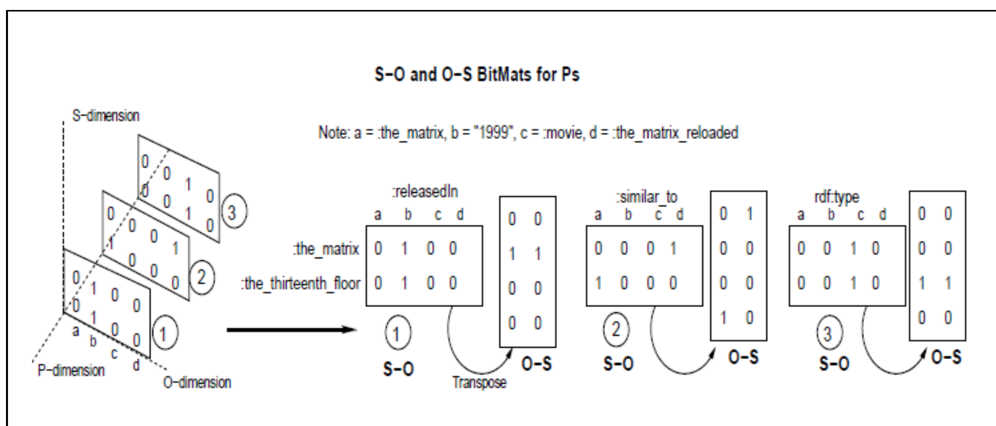


Figure 9: BitMat S-O and O-S matrices (from [5])

Each triple of a given N3 dataset is a point in the 3D cube. A typical dataset covers a very small number of points in such a 3D space. Hence, BitMat matrix tends to be sparse. In order to achieve a compression that can directly be used for query process-

ing, we have used the *D-gap* compression scheme.[31] The idea is to translate a binary array to a structure using integer: The very first integer in the sequence is always $[0]$ or $[1]$. It works as a flag indicating the start bit. The integers following the flag are the lengths of the consecutive blocks of equal bits. The sum of all elements of the sequence without the starting flag value provides the total length of the blocks. For example, the output for the binary sequence (`000 1 000 111 00 1111`) is $([0], 3, 1, 3, 3, 2, 4)$. This means, it begins with '`0`' with length 3, then '`1`' with length 1, '`0`' with length 3, then '`1`' with length 3, then '`0`' with length 2, and then '`1`' with length 4. The total length can be easily calculated as the sum of the lengths of the alternating subsequences of `0`'s and `1`'s; in this case, $3 + 1 + 3 + 3 + 2 + 4 = 16$.

The `DgapCompressor` class generates three files for each predicate: a matrix S-O with the extension `.matrixSO`, a matrix O-S with the extension `.matrixOS`, and an index `Id-Literal` with the extension `.index`. Instead of storing binary huge lines, we store each line in D-Gap compressed format. Also, we add metadata for the purpose of statistical analysis. Figures 10, 11, and 12 are examples of the predicate `:name`, produced by our preprocessor client.

```
1:[0]1,1,34363

1:[0]3,1,34361

1:[0]5,1,34359

1:[0]7,1,34357

1:[0]9,1,34355

1:[0]11,1,34353
```

Figure 10: An example of compressed predicate: some lines in the S-O matrix

```
1:[1]1,34364

31:[0]2,1,1763,1,1284,1,1171,1,1042,1,1171,1,1088,1,888,1,1008,1,1265,1,1248,1,1042,1,1169,1,1038
,1,1123,1,1030,1,1037,1,867,1,931,1,1039,1,1126,1,984,1,906,1,1071,1,1040,1,1012,1,1066,1,1151,1,
1096,1,1288,1,1226,1,1162

1084:[0]4,1,21,1,9,1,11,1,11,1,7,1,10,1,6,1,9,1,27,1,15,1,15,1,11,1,16,1,14,1,17,1,15,1,18,1,11,1
,17,1,123,1,18,1,16,1,21,1,19,1,17,1,16,1,22,1,16,1,346,1,3,1,4,1,4,1,5,1,4,1,834,1,7,1,10,1,8,1,
5,1,9,1,5,1,9,1,6,1,8,1,17,1,16,1,14,1,18,1,17,1,12,1,16,1,10,1,16,1,13,1,14,1,76,1,19,1,18,1,19,
1,19,1,20,1,19,1,18,1,18,1,19,1,250,1,2,1,1,1,6,1,1,1,1,1,512,1,5,1,6,1,5,1,5,1,9,1,10,1,9,1,9,1,
6,1,7,1,6,1,18,1,16,1,18,1,18,1,14,1,15,1,16,1,16,1,17,1,11,1,16,1,10,1,82,1,16,1,19,1,20,1,20,1,
19,1,15,1,17,1,16,1,18,1,210,1,4,1,2,1,3,1,2,1,408,1,8,1,7,1,10,1,6,1,5,1,5,1,9,1,10,1,5,1,9,1,11
```

Figure 11: An example of compressed predicate: some lines in the O-S matrix

This compressed data structure and a proposed query processing algorithm [5] enable loading larger datasets into *main memory* and directly perform queries on compressed data with an improved overall performance. In order to create an unique index of

[31] http://bmagic.sourceforge.net/dGap.html

```
0 <http://www.Department2.University1.edu>
11836 <http://www.Department3.University1.edu/UndergraduateStudent45>
18359 <http://www.Department35.University1.edu/UndergraduateStudent78>
7733 <http://www.Department27.University1.edu/AssociateProfessor8/Publication0>
2420 <http://www.Department22.University1.edu/GraduateStudent139>
14256 <http://www.Department32.University1.edu/AssociateProfessor1/Publication3>
8943 <http://www.Department28.University1.edu/GraduateStudent17>
4840 <http://www.Department24.University1.edu/Lecturer2>
16676 <http://www.Department34.University1.edu/FullProfessor4/Publication3>
11363 <http://www.Department3.University1.edu/GraduateStudent48>
```

Figure 12: An example of compressed predicate: some lines in the index

all tokens or nodes (subjects and objects), the compression can be the most memory-intensive part in the client program. This step performs the following operations:

- `Integer insertNode(String node)`—inserts a `String` token and an unique integer `Id` is returned. This is a composed "*insert if not exists then select*" operation.
- `Integer fetchIdByNode(String node)`—fetches the unique integer `Id` of a given `String`.
- `Integer fetchNodeById(Integer id)`—fetches the unique `String` of a given `Id`.
- `void addSO(SOIntegerPair so)`—adds an integer pair in a list containing all nodes in the `Subject` × `Object` space where all nodes are ⟨`subject`, `object`⟩ entry pairs represented by their unique integer `Id`.
- `List`⟨`SOIntegerPair`⟩ `fetchSOList()`—returns the entire list of points.

The database technologies are used for these operations. We have developed several methods extending the `DBUtils` class, using relational or non relational database or in-memory approaches. For existing database solutions, we have implemented replaceable modules giving support to the following:

- *MonetDB* is an open source column store product under Mozilla open source License.[32] It aims to use the maximum of available memory and cores, which is practically possible to be deployed in parallel processing of queries. Also, it tries to avoid storing data onto disc. It has a SPARQL-based component for working with linked data and allowing users to build simple triplestores.
- *MongoDB* is an open source NoSQL database under Apache License.[33] Its data storage model is designed for semi-structured data. It stores document-like data into sets of key-value pairs.

[32] https://www.monetdb.org/Home
[33] https://code.google.com/p/guava-libraries

- *MySQL* and *Oracle* are two relational database management systems owned by Oracle.[34] MySQL is a free, fast, robust and open source database under GNU (General Public License). Oracle is more versatile and supports many unique features, however, it has a large number of tuning options that takes a lot of time to investigate the right combinations.
- *PostgreSQL* is an object-oriented relational database system.[35] It is similar to a relational database but with a different model where objects, classes and inheritance are directly supported in data schemas and therefore in the query language. It is under PostgreSQL License, a permissive free software license like the BSD licenses.
- *Redis* is an open source and BSD licensed advanced key-value store.[36] It is often referred to a data structure server since the keys can contain strings, hashes, lists, sets and sorted sets.

As mentioned earlier, CedTMart provides several alternatives including main-memory-based database which we have implemented with:

- a bidirectional `HashMap` using Google's Guava library:[37]
  - `BiMap⟨Integer, String⟩` which allows two-ways key-value operations `fetchIdByNode(String node);` and,
  - `fetchNodeById(Integer id);`
- an `ArrayList` of integer pairs: `ArrayList⟨SOIntegerPair⟩` describing nodes in the `Subject` × `Object` space.

CedTMart performs compression in two phases: *Loading* and *Computing & Writing*. In the loading phase, like POS step, a thread called *loop function* is assigned to process each file of a set of files. In the file processing function, an `SOReader` reads predicate files line by line but without detecting the object type. Figures 13, 14, and 15 depict the compression procedure.

Once the entire file is loaded into the temporary database, the function:

```
void writeCompressedFormat ( String  inFileName
                           , String  outputPath
                           , DBUtils dbu
                           , String  comparePath
                           )
```

in class `DgapCompressor` is called. At first, it sorts the list of ⟨`subject`, `object`⟩ nodes according to subject elements and produces the outcome *S-O matrix*. Then

[34] http://www.oracle.com/index.html
[35] http://www.postgresql.org
[36] http://redis.io
[37] https://www.monetdb.org/Home

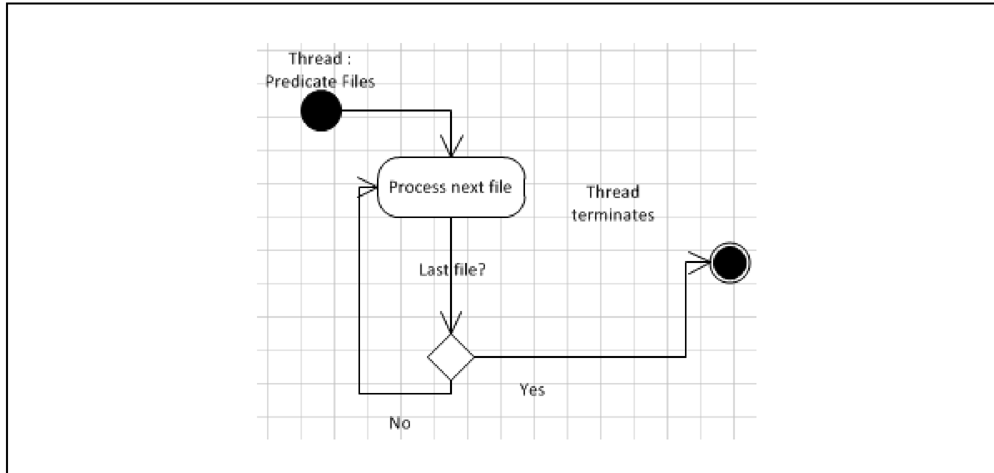


Figure 13: Compression procedure

it sorts the list according to object elements and outputs the O-S matrix, finally it iterates the bidirectional `HashMap` and writes the index file. In this step, as an option, sorted subjects and objects can be represented as arrays to a compare folder (by default `_compare`) for performing comparison at later step. Figure 16 gives a clearer view of this core function.

During this implementation, we learned how to avoid very long strings by *flushing* the buffer of its writer periodically. The JVM can be extremely inefficient even nearly frozen while processing long strings (for example longer than 128K). In our tests we have observed that a function without splitting too long strings could never finish (See Section 6). Also, in some extreme cases (for example predicate *a*), this method does not compress at all because our initial assumption become invalid in such situations. If ⟨`subject`, `object`⟩ nodes of predicates are not sparse at all, the D-Gap compression will not provide satisfying results.

5.4 Comparison

The purpose of the *Data Distributor* module is to distribute compressed data intelligently across the nodes in Hadoop clusters. In order to do so, CedTMart measures *betweenness* among predicates. It calculates the *similarity* between two predicates, which refers to the sum of common distinct subjects and common distinct objects which belong to two predicates. Since it is difficult to perform this computation on compressed data, we reuse predicate files produced during the predicate-partition step.

The comparison is done in two steps: 1. *Pre-Comparison* and 2. *Comparison*. In the Pre-Comparison step, predicate files are split into two individual *arrays of subjects and objects*. Then, they are stored in files:

- `predicate_file_name.S`, and
- `predicate_file_name.O`,

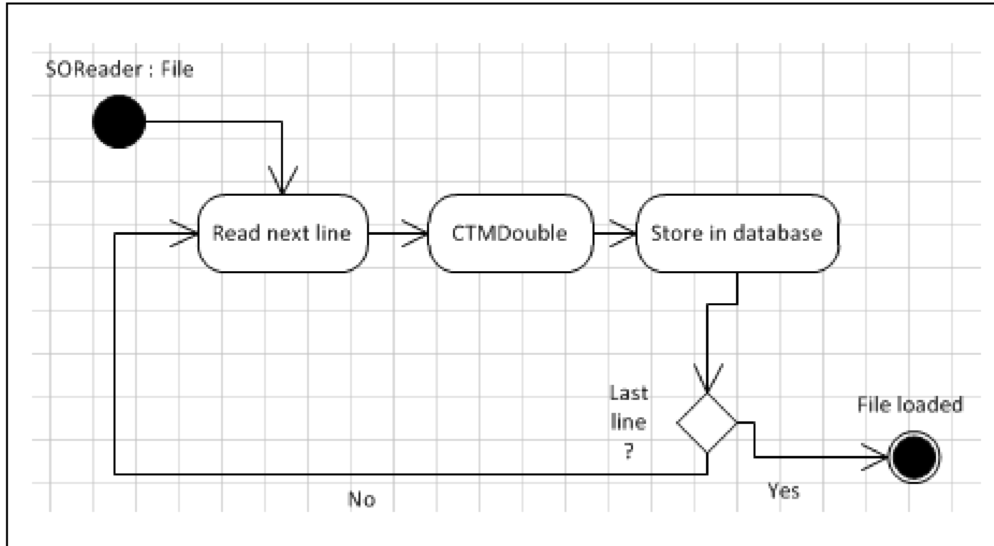


Figure 14: Loading phase procedure for each file

which are located by default in the `_compare` folder. CedTMart exports the sorted *subjects* and *objects* to the *Compression* step. Figure 17 shows the Pre-Comparison step.

In addition to this approach, the CedTMart provides two other alternatives for splitting predicate files:

- a Java parser-writer using `SOReader` and `writeToBigFile()` function.
- a Perl parser-writer using a perl script. It consumes less memory (only 5M in contrast to almost 200M per thread running the above Java function)

For comparison, several comparing methods have been implemented in CedTMart using Java subsystems and external scripts. They can be grouped into *in-memory* and *disk buffered* methods. For all in-memory and disk buffered methods, in the loop function of the *Comparison* step, a thread always catches a list of `FilePair` containing four array files of two different predicates, say A and B. Then, it compares A and B's common subjects and objects; and finally, returns a sum storing in a specified indicator file (by default the `_indicator` folder). Figure 18 shows the process of comparing subjects and objects in files.

The in-memory compression method `InRamUtils` can be used if sufficient physical resource in particular, RAM (Random Access Memory) is available to load each predicate file containing two columns (*Subject* and *Object*). The core function of in-memory is `compareTwoPredicates (File f1, File f2)`. This function creates two `Future⟨HashSet⟨String⟩⟩` instances and submits to each a value-returning task for execution. It returns a `Future` instance that represents the pending results of the task. A `Future`'s `get()` method returns task results upon successful completion of two `HashSet⟨String⟩`. The function `SmallerSet.retainAll(BiggerSet)` is then executed by the core function, which finally retains the elements in a set. (Note that

```
public void insertOrIgnorePredicateNodes(DBUtils dbu, CTMDouble so)
        throws SQLException {
    String S = so.getSubject().toString();
    String O = so.getObject().toString();
    Integer iS = dbu.fetchIdByNode(S);
    Integer iO = dbu.fetchIdByNode(O);
    if(iS == null){
        iS = dbu.insertNode(S);
    } else {
        iS = dbu.fetchIdByNode(S);
    }
    if(iO == null){
        iO = dbu.insertNode(O);
    } else {
        iO = dbu.fetchIdByNode(O);
    }
    dbu.addSO(new SOIntegerPair(iS,iO));
}
```

Figure 15: Loading predicate file for comparison

these elements are still contained in another set.) Finally, it obtains a collection of common subjects or objects. An advantage of the in-memory comparator is that it is not necessary to sort subjects or objects arrays because `HashSet` is already hashed into a unique set. Figure 19 depicts the implementation of comparing ⟨`subject`, `object`⟩ arrays in main memory.

The CedTMart preprocessor client provides multiple combinations of various solutions. These are summarized as follows.

- The *in-memory comparator* (class `InRamComparator`). It does not need to sort the input arrays of ⟨`subject`, `object`⟩ pairs. However, it is the most memory hungry approach.
- The *Java comparator* (class `JavaComparator`), which contains the function `compareTwoPredicates(File f1, File f2)`. It compares strings *line by line* and thus is available for sorted files only.
- The Perl comparator (class `PerlComparator`), which calls a `ProcessBuilder` instance to manage a collection of process attributes and to create a new *Process* instance with those attributes in order to run an external Perl script using the `start()` method.

Since two disc buffered methods (Java and Perl) can only take sorted arrays as inputs to calculate distinct common subjects and objects, in the current version of CedTMart, we use GNU's *sort* executable to sort text files.[38]

[38] http://www.gnu.org

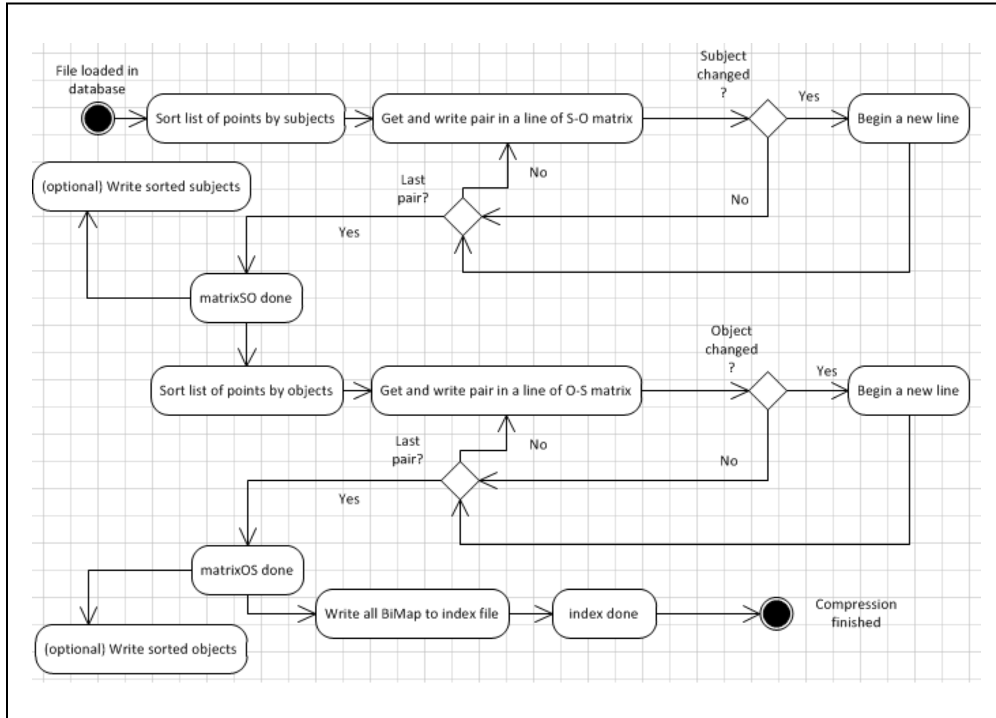


Figure 16: Procedure for computing and writing phase for each file

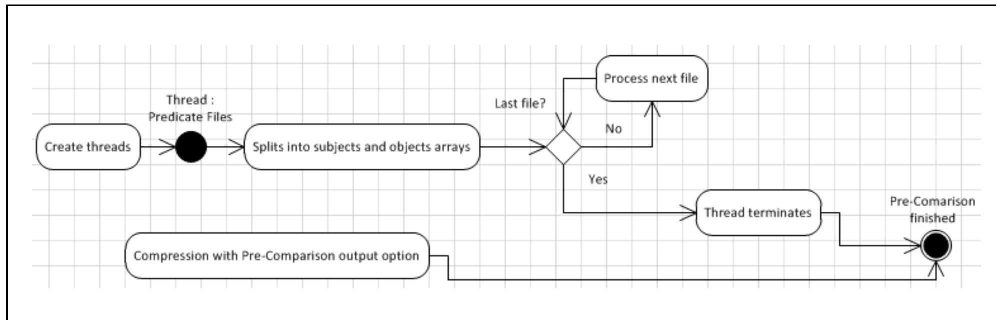


Figure 17: Flowchart of the Pre-Comparison step

5.5 Distributor

The purpose of this component is to distribute compressed data with index to clusters of nodes in particular, Hadoop clusters. The *Data Distributor* contains a data distribution client. Also, it relies on a set of protocols which we have implemented in this project. It transfer data over socket. The protocols are explained below. It is worth noting that the distributor has been implemented considering Hadoop's master-slave architecture. Figure 20 shows data distribution.

- The client connects with a node (this should be the namenode) on a given address and a specified port (predefined by the user) and announces the amount of

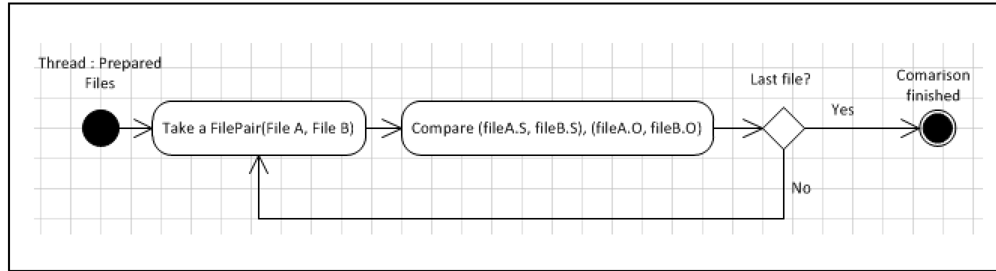


Figure 18: Comparison of files

```
public static int compareTwoPredicates(File f1, File f2)
        throws IOException, ParseException,
        InterruptedException, ExecutionException {
    IOUtils.logLog("Start loading");
    ExecutorService executor = Executors.newFixedThreadPool(2);
    Future<HashSet<String>> fut1 = executor.submit(
            new FileLoader(f1.getAbsolutePath()));
    Future<HashSet<String>> fut2 = executor.submit(
            new FileLoader(f2.getAbsolutePath()));
    HashSet<String> set1 = fut1.get();
    HashSet<String> set2 = fut2.get();
    executor.shutdown();
    return compareTwoHashSets(set1, set2);
}

private static int compareTwoHashSets(HashSet<String> s1,
        HashSet<String> s2){
    //small.retainAll(large);
    if(s1.size()<s2.size()){
        int bigsize = s2.size();
        s1.retainAll(s2);
        return bigsize - s1.size();
    } else {
        int bigsize = s1.size();
        s2.retainAll(s1);
        return bigsize - s2.size();
    }
}
```

Figure 19: Comparing two ⟨`subject`, `object`⟩ arrays in main memory

compressed data that will be sent to other nodes (these are compute nodes). This is a string message with message's length and a header of 8 bytes.

- The namenode checks the free spaces available on computing node(CN) and sends a formatted JSON string back to the client (see Figure 21). The String contains amount of free space, IP address, and port of each available compute node, and maximum size of data blocks to be sent (default value is 512MB). The IP and port information are used to establish the connection between client and each CN. Then, the client sends data to CNs according to *free_space* (in MB) and *ratio*.
- The client disconnects from the namenode once it receives a valid response.

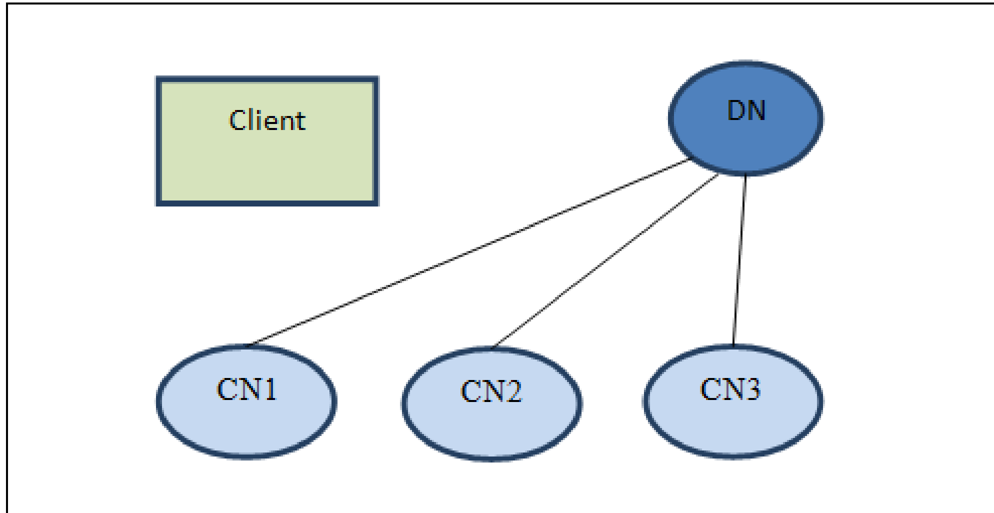


Figure 20: Illustration of data distribution across nodes


```
[{"ip":"192.168.0.1","port":"8888","free_space":"20480","ratio":"0.5"},
{"ip":"192.168.0.2","port":"8686","free_space":"200000","ratio":"0.3"}]
```


Figure 21: JSON response for DN

- CedTMart client distributes data in two ways: *strategically* or *randomly*. For strategic distribution, it uses the outcome of comparison operation. In the function `distribute()` of the main thread (`CTMServer`), the client looks for valid indicators in the `_indicator` folder. Then it assigns compressed predicate files to a number of groups according to the betweenness indicator. The client choose random distribution if there is no indicator found. In this case, the client distributes data randomly based on the information of available compute nodes returned by the namenode.
- The client creates a predefined number of threads. Each opens a connection to a list of different nodes and sends chunks of data with the predefined size these nodes. To explain more, for each compressed predicate file, each client thread opens the socket to the node that it wants to contact and then begins sending data. If the data file is smaller than the maximum block size, the entire data chunk is sent with the header: `DATA:N`⟨`filename`⟩ ||-||. An acknowledgement `DATA:ACK;` is sent in response. It indicates the successful transfer of data blocks. If the data chunk is bigger, it is split into chunks and sent with the header `PATA:N`⟨`filename`⟩`, P`⟨`size_of_chunk`⟩ ||-||. Figure 22 presents the data transfer protocol.

Note that, currently, in CedTMart, data can be transferred as simple string and are not be encrypted. We plan to implement a data security mechanism in future.

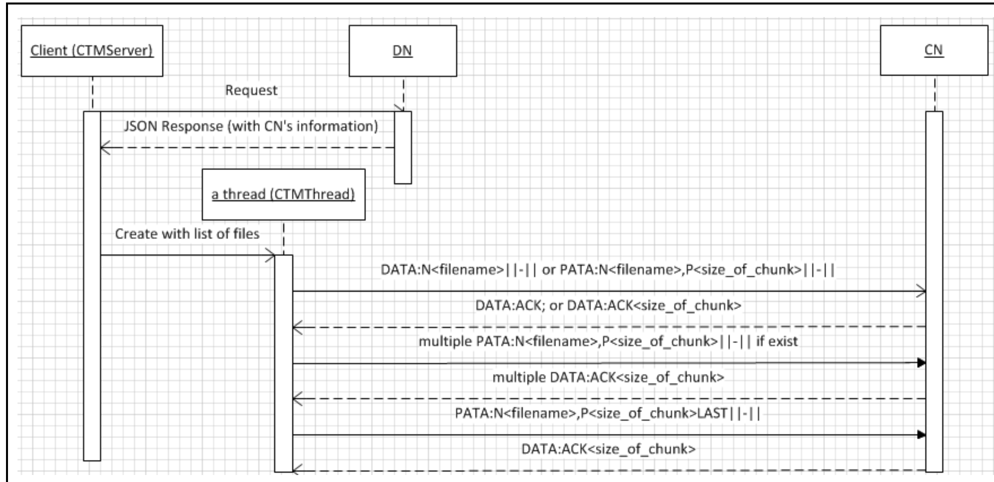


Figure 22: Connection sequence between client, DN, and a CN

We have implemented a component to enable the client using HDFS [29] for data distribution (see Figure 23). HDFS is designed to store large files typically, in the range of gigabytes to petabytes across multiple machines, which is the relevant size of our generated data. It ensures reliability by replicating the data across multiple node machines. For example, with the default replication value 3, it stores the data on three nodes within which two on the same rack and the other on a different one. Compute nodes communicate with each other to maintain the load balancing, to move copies and to keep the data replication.

In this alternative, the distributor deploys compressed predicate files to Hadoop clusters. This will use Hadoop Java API which solves automatically low level protocols: the HDFS uses the TCP/IP layer for communication and clients use Remote procedure call (RPC) to communicate between each other.

5.6 Query processing

Lately, we have finished the implementation of query optimization module that comprises *Query Planner* and *Query Executor*. These components rely on two algorithms that we have designed in this research project. Since are still debugging the implementation, we decided to detail the implementation in the next report.

6 Evaluation

We have conducted several experiments with the current version of CedTMart prototype. In this section, we present the results of the experiments. We evaluate the performance of preprocessing and distribution modules of CedTMart and compare our framework with others. Before going into details, we provide a description of the parameters that are needed to operate the CedTMart prototype.

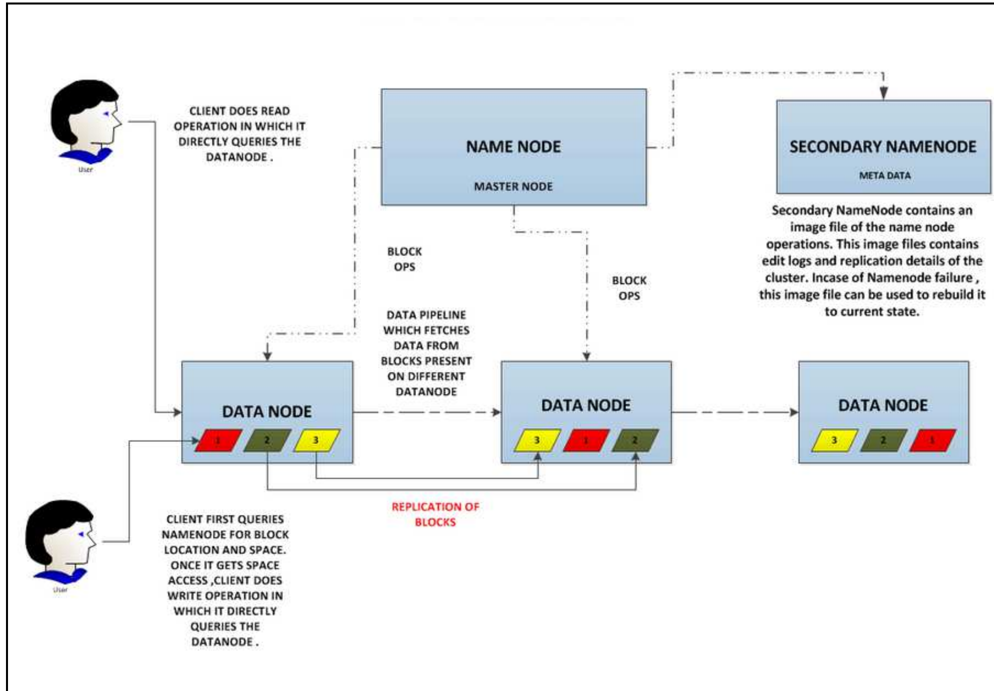


Figure 23: A simple illustration of HDFS

6.1 Parameters

CedTMart is designed to run in command line interface. The users can directly run the executable `.jar` file. Once it is executed, the preprocessor program prepares itself using all default values. If users want to load some modules with external scripts or external executables; these required elements which should appear in:

- the `exec` folder besides the main jar executable, for all executable programs and jar executables;
- the `script` folder besides the main jar executable, for all external Perl and Shell scripts.

Note that, the input datasets are always prerequisite. The default value for RDF datasets is `__rdf`, and for N3 datasets is `__n3`. If no arguments detected by the main function, the client will show a menu which asks to choose an operation. Figure 24 shows the menu provided by the CedTMart.

The default number of worker threads is 4. The menu enables users to define different value of number of threads for each operation. The jobs will be assigned to the number of threads (see Figure 25).

If an error value is detected, the CedTMart will run using its default values. Figure 26 shows the default values.

It is worth noting that users can specify easily all these parameters by modifying values in the configuration section. The options are listed below:

```
Type number to execute :
        Clean all existing processed data - 10000
        RDF to N3 converter - 101
        N3 Reader/Partitionner(PS) - 102
        Predicate Reader/Splitter(POS) - 103
        Compressor for PS files(with optional Pre-Comparator for PS files) - 104
        Pre-Comparator for PS files - 105
        Comparator for S/O arrays - 106
        Distributor of compressed PS files - 107
        Exit - -1
#
```

Figure 24: A demo without argument: choose a task

```
102
How many threads (>0) ?
8

Partitionning N3 :
Input : D:\DEV\Eclipse_WP_EE\CtmRdf\..\CtmDataSet\__n3(26M)
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_ps
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_ns
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_invalidTriple
Adding 4
Adding 4
Adding 4
Adding 4
Adding 4
Adding 4
Adding 4
Adding 3
Sub task 0 with 4file(s)
Sub task 1 with 4file(s)
Sub task 2 with 4file(s)
Sub task 3 with 4file(s)
Sub task 4 with 4file(s)
Sub task 5 with 4file(s)
Sub task 6 with 4file(s)
Sub task 7 with 3file(s)
Sub tasks assigned
```

Figure 25: A demo without argument: number of threads

```
#
102
How many threads (>0) ?
IN
Input error, using 4 threads...

Partitionning N3 :
Input : D:\DEV\Eclipse_WP_EE\CtmRdf\..\CtmDataSet\__n3(26M)
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_ps
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_ns
Output : D:\DEV\Eclipse_WP_EE\CtmRdf\_invalidTriple
```

Figure 26: A demo without argument: number of threads

- `setLogging(boolean, boolean)`: the first boolean informs the logger to output log messages to the console or not, and the second informs the logger to output to an automatically created log file or not;
- `nbThreads`: number of worker threads;
- `compressMode`: it defines a mode for the compressor, perhaps needs external DB support (see Section 5.3);
- `writeprecompare`: it informs the comparator to write sorted subject/object files of each predicate or not (see Section 5.3);
- `precompareMode`: it defines a mode for the pre-comparator, perhaps needs Perl executable in PATH environment variable;
- `compareMode`: it defines a mode for the comparator, perhaps needs Perl or GNU executable in PATH environment variable;
- `ctlParams`: it stores input and output paths for RDF to N3 conversion, PP, POP, compression, pre-comparison, comparison and only input source for distribution (because the program receives destination information by requesting a manager node in a distributed environment)

Once an user performs a specified operation, all concerned parameters are displayed on the screen. This ensures that the source to destination has been processed correctly.

6.2 Experiment

In our experiment, we use two physical machines with the following specification:

- *Machine 1*
  - I7 960 quad core processor with Hyper-Threading Technology,
  - 16GB main memory
  - 1TB Seagate 7200.12 hard disk (HD)

```
----------------------------------------
|--------          CTM          --------|
----------------------------------------
Using parameter : nsPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_ns
Using parameter : invalidPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_invalidTriple
Using parameter : compressedPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_compressed
Using parameter : psPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_ps
Using parameter : posPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_pos
Using parameter : rdfPath - D:\DEV\Eclipse_WP_EE\CtmRdf\..\CtmDataSet\__rdf
Using parameter : indicatorPath - D:\DEV\Eclipse_WP_EE\CtmRdf\_indicator
Using parameter : comparePath - D:\DEV\Eclipse_WP_EE\CtmRdf\_compare
Using parameter : n3Path - D:\DEV\Eclipse_WP_EE\CtmRdf\..\CtmDataSet\__n3(26M)
```

Figure 27: Display all paths

- *Machine 2*,
  - I5 3320M dual core with Hyper-Threading Technology,
  - 16GB main memory,
  - 180GB Intel 530 solid state drive (SSD)

In the following subsections, we show individual results for our experiments of different tasks. Two smaller datasets, 26MB and 2GB are used during implementation phase to ensure the correctness of our data processing, and a bigger dataset of 20GB containing 23000 N3 files is used for experimental results.

For each result, we performed five experiments. Then repeatedly removed the pair consisting of the highest and lowest values and got the average of three left scores.

6.2.1 Results of predicate partitioning

In this experiment, the 20GB dataset is merged into 17 predicate files which are equal to 15.6 GB. We record the elapsed time for the program running with 1, 2 and 4 threads on machine 1 and machine 2 with the optional merging operation which combines output files of each worker thread in 17 distinct predicate files. The test results range from 188 seconds to less than 2000 seconds based on different configurations. Table 1 shows the performance comparison for our test cases respectively.

In this I/O intense phase, the size of main memory do not influence much. The memory usage stay around 50-100 MB per thread. Also, we tested using 8 GB main memory in Machine 2 but we did not notice any difference. Limited by the bottleneck of local I/O performance, the solid state drive is more efficient. It takes only 1/10 of elapsed time than hard disk.

6.2.2 Results of predicate-object partitioning

In this experiment, the 15.6 GB of 17 predicate files is converted to 30 files and 15.1 GB. The results of different settings range from 260 seconds to 1050 seconds. Table 2

Table 1: PP using various settings

Machine 1		
Operation	Number of threads	Elapsed time in second
Partitioning	1	1501
Partitioning	2	1925
Partitioning	4	1933
Merging	1	320
Machine 2		
Operation	Number of threads	Elapsed time in second
Partitioning	1	392
Partitioning	2	248
Partitioning	4	188
Merging	1	141

shows the results for different settings respectively.

Again, limited by the bottleneck of local I/O performance, we observed significant improvement of processing time when the number of threads increased from 2 to 4. The processing time of Machine 2 (which has SSD for storing data) is always inversely proportional to the number of worker threads. It is worth noting that fragments in hard disks affect the benchmark for 5 to 30 percent, for example, 4 threads can take over than 1350 seconds without defragmenting periodically the hard disk in Machine 1.

6.2.3 Results of compression

We performed compression operation in this experiment. We used both local file system (FS) with main-memory and the database technologies. Our triplestore provides both of these modes to perform compression. The outcomes of in-memory compression test is presented in Table 3.

Table 4 shows the the outcomes of the experiments with database technologies using Machine 1.

Due to huge memory consumption per thread, we performed tests with single thread only. We found that the compression rate varies between 10% and 95% depending on predicate files. It is worth noting that we found data compression a CPU bound job. In the beginning we often encountered `java.lang.OutOfMemoryError` error with a message called *GC overhead limit exceeded*. Figure 28 shows the JVM memory structure.

The JVM memory structure is divided into three parts. The *Heap Memory* is also called the *dynamic memory*, where programs store new objects or variables. Two other

Table 2: POP using various settings

Machine 1		
Operation	Number of threads	Elapsed time in second
Splitting	1	1050
Splitting	2	979
Splitting	4	1108
Machine 2		
Operation	Number of threads	Elapsed time in second
Splitting	1	424
Splitting	2	327
Splitting	4	261

Table 3: In memory compression using various settings

Operation	Number of threads	Elapsed time in second
Machine 1		
In memory compression	1	1053
Machine 2		
In memory compression	1	1120

Table 4: Compression using database technologies

Databases	Time Elapsed (in second)
Redis	1042
MonetDB	1043
PostgresDB	1044
Oracle	1045
MySQL	1046
MongoDB	1047

Figure 28: Java VM memory structure

parts represent the *Stack Memory* where only the values are stored within the scope of the function they are created in. We observed 99% memory consumption for storing temporary objects in complex data structures for example, large HashSet of strings. The `-Xmx⟨size⟩` was set the maximum Java heap size and the `-Xms⟨size⟩` was set for the initial Java heap size. If the free heap size would be smaller than 40%, JVM would increase the size to `Xmx` threshold. If free heap size would be larger than 70%, JVM would decrease the size to `Xms` threshold. In some circumstances like many Java WEB servers, when we know clearly how much main memory our programs use, we can set a same value to the initial and maximum size purposely, in order to avoid pauses caused by resizing the heap zone.

Another important issue is the GC (garbage collector) which is a daemon thread invokes `finalize()` method of eligible objects (basically those objects linked with any other object) and removes them. While experimenting our triplestore with Oracle we found that if 98% of the total time was spent in garbage collection and less than 2% of the heap is recovered, an OutOfMemoryError was thrown. Thus, during the tests, we used `-Xms12288m -Xmx12288m -XX:+UseConcMarkSweepGc` arguments for the Java VM to maximize the heap size. Additionally, the JVM was instructed to disable the error check altogether then we did it ourselves.

In the experiment using local FS with in-memory compression mode, the worst case is, a predicate file of 4GB can use 9GB main memory. In some extreme cases, there may have very long string in generated matrix files which should be avoided. We discussed of such cases in the earlier section. Taking these cases into account, in addition to in-memory compression, CedTMart enables using existing relational or non relational database systems. It is worth noting that queries such as *insert, ignore, return* should be performed in large tables with two unique columns: one integer and the other is string. For the relational database systems, generally, MySQL with MyISAM engine provides better performance than with InnoDB and PostgreSQL. Additionally, Oracle with *parallel query execution* can provide significantly better performance than MySQL with MyISAM. However, it has a large number of tuning options which pose enormous challenge especially in finding the right tuning combinations. MonetDB is a special one using column storage. To load a large table of over than 1GB, MonetDB can take 150% more time than MySQL, but its query time has substantially better than MySQL or even Oracle. Among tested NoSQL databases, Redis key-value store performs much better than MongoDB. We used the Jedis library with two modes:

Table 5: Redis `Insert` and `Get` results

Operation	Quantity	Elapsed time in second
Standard		
insert random string value of 64 bytes	1000000	255
get value using random integer keys	1000000	>1000
Pipeline		
insert random string value of 64 bytes	1000000	6
get value using random integer keys	1000000	107
insert random string value of 64 bytes	32000000	97
get value using random integer keys	32000000	>1000

standard and *pipeline*.[39] The outcomes of compression using Redis is presented in Table 5.

In Redis, most of the time was spent on fetching values. Note that, Redis is also an in memory database. According to its official documentation, Redis loads everything into memory. The data written to disk as well however would only be read for special purposes such as restarting the server or making a backup. In our tests, 32 Million lines occupy 6GB of main memory. We found that a bidirectional Mapping is needed to improve the performance of compression operation using database technology. Redis is an ideal option for that.

Our observation concludes that the local FS with in-memory mode is better than the traditional database solution for performing high performance compression operations.

6.3 Results of comparison

In this experiment, 17 predicate files are split in 17*2 files of separate subjects and objects, then a comparator counts common subjects and objects between each pair of two distinct predicates. The comparison outcomes are presented in 6.

Like compression, this is a I/O intensive task. From the table it is clear that the Java splitter performs better than the Perl splitter yet it consumes more memory. In addition, we observed that the memory usage of Java BufferedWriter and Perl's writer increase if the faster storage such as SSD is used.

In order to optimize the performance, we used GNU's *sort* for sorting files of subjects and objects. The total time consumed for completing this step is 75 minutes using single thread. We then increased the threads. Using 8 threads the comparison of 20GB dataset was completed in 1000 seconds.

The comparison module of CedTMart has three classes of solutions. Each of them use less than 100 MB of memory (per thread). The first solution comprises two Java

[39] https://github.com/xetorthio/jedis

Table 6: Pre-Comparison split using various settings and alternatives

Machine 1			
Operation	Number of threads	Elapsed time in second	Memory usage
Java split	1	1083	198MB
Java split	2	1016	403MB
Java split	4	977	767MB
Perl split	1	1002	4MB
Perl split	2	1054	9MB
Perl split	4	1249	17MB
Machine 2			
Operation	Number of threads	Elapsed time in second	Memory usage
Java split	1	205	254MB
Java split	2	151	509MB
Java split	4	120	1027MB
Perl split	1	356	5MB
Perl split	2	252	11MB
Perl split	4	209	22MB

Table 7: Comparison using Java and GNU alternatives

Operation	Number of threads	Elapsed time in second
Java comparison	1	11317
Java comparison	2	6216
Java comparison	4	3789
GNU comparison	1	39712
GNU comparison	2	21988
GNU comparison	4	13465

methods: the in-memory HashSet-based algorithm and file comparator for sorted files. Second, the GNU's *comm* technique which compares two sorted files line by line. The third solution is the Perl script which essentially spends more time than the GNU's comm. Table 7 shows the results.

We found that, in the current version of CedTMart, comparison consumes most of the time. Also, optimizing the performance of comparison operations was more challenging than others. We believe that there is a scope of improving this component, which we will be doing in future.

7 Conclusion and Future Work

In this research project we developed a high-performance triplestore for storing and querying Blinked Data. We implemented components: a data cleaner, a data partitioner, a data compressor of the preprocessing module. We implemented data a re-partitioner, a data organizer, and a data distributor of distribution module. Additionally, we recently finished the implementation of a query planner and a query executor of query execution module. We designed several high-performance algorithms.

We conducted several experiments. The experiments show that significant improvement of CedTMart's performance in some cases. However, there are bottlenecks that have to be addressed to enhance the current performance. For instance, the main-memory based module of compression step is the most memory-intensive one, and other solutions using existing database systems cannot provide a same performance. The jobs in the database systems generate additional costs like indexing, redundancy, network traffic, *etc.*, ... Column based and some key-value data stores seems more suitable for the compression. However, many of them are implemented as an in-memory architecture and takes almost the same size of the Java-based solution. Furthermore, the comparison with sort operation is more CPU intensive. We are working on to reduce consumption of computing resource and to enhance performance.

In the future, we plan to experiment with the query processing module we recently implemented. Also, we plan to work on improving the algorithms to enhance the current performance of each module of the CedTMart independently of one another.

We will continue this agile development protocol until we reach the performance we seek.

References

[1] Daniel J. Abadi, Adam Marcus, Samuel R. Madden, and Kate Hollenbach. Scalable semantic web data management using vertical partitioning. In Christoph Koch, Johannes Gehrke, Minos N. Garofalakis, Divesh Srivastava, Karl Aberer, Anand Deshpande, Daniela Florescu, Chee Yong Chan, Venkatesh Ganti, Carl-Christian Kanne, Wolfgang Klas, and Erich J. Neuhold, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'07)*, pages 411–422, Vienna, Austria, September 23–27, 2007. VLDB Endowment, ACM. [Available online[40]].

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. HadoopDB: An architectural hybrid of MapReduce and DBMS technologies for analytical workloads. In Serge Abiteboul, Volker Markl, Tova Milo, and Jignesh Patel, editors, *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB'09)*, pages 922–933, Lyon, France, August 2009. VLDB Endowment, ACM. [Available online[41]].

[3] Hassan Aït-Kaci, Mohand-Saïd Hacid, Rafiqul Haque, and Damien Fourure. Experiments with triplestores. CEDAR Technical Report Number 5, Université Claude Bernard Lyon 1, Lyon, France, October 2013. [Available online[42]].

[4] Medha Atre, Vineet Chaoji, Mohammed J. Zaki, and James A. Hendler. Bitmat - scalable indexing and querying of large RDF graphs. Technical report, Rensselaer Polytechnic Institute and Yahoo Research, USA and India, 2011. [Available online[43]].

[5] Medha Atre, Jagannathan Srinivasan, and James A. Hendler. Bitmat: A main-memory bit matrix of RDF triples for conjunctive triple pattern queries. In *Proceedings of the Posters and Demo Track of the 7th International Semantic Web Conference*, Karlsruhe, Germany, 2008. [Available online[44]].

[6] Kamil Bajda-Pawlikowski, Daniel J. Abadi, Avi Silberschatz, and Erik Paulson. Efficient processing of data warehousing queries in a split execution environment. In *Proceedings of the ACM SIGMOD Conference*, pages 1165–1176, 2011. [Available online[45]].

[7] Tim Berners-Lee and Dan Connolly. Notation3 (N3): A readable RDF syntax, March 2011. [Available online[46]].

[8] 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[47]].

[9] Pierre-Antoine Champin. RDF tutorial. [Available online[48]].

[40] http://dl.acm.org/citation.cfm?id=1325851.1325900
[41] http://db.cs.yale.edu/hadoopdb/hadoopdb.pdf
[42] http://www.cedar-liris.fr/documents/ctr4.pdf
[43] http://www.cis.upenn.edu/˜atrem/papers/bitmat_jreport.pdf
[44] http://ceur-ws.org/Vol-401/iswc2008pd_submission_16.pdf
[45] http://cs-www.cs.yale.edu/homes/dna/papers/split-execution-hadoopdb.pdf
[46] http://www.w3.org/TeamSubmission/n3/
[47] http://tomheath.com/papers/bizer-heath-berners-lee-ijswis-linked-data.pdf
[48] http://liris.cnrs.fr/amille/enseignements/Ecole_Centrale/rdf.pdf

[10] Minwei Chen. CedTMart—a triplestore for storing and querying blinked data. Master's thesis, Université Claude Bernard Lyon 1, Computer Science Department, Villeurbanne, France, July 2014.

[11] The World Wide Web Consortium. Rdf primer w3c recommendation, February 2004. [Available online[49]].

[12] Oracle Corporation. Oracle spatial and graph: Advanced data management. White Paper, June 2013. [Available online[50]].

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communication of the ACM*, 51(1):107–113, January 2008. [Available online[51]].

[14] Dirk deRoos, Chris Eaton, George Lapis, Paul Zikopoulos, and Tom Deutsch. *Understanding Big Data: Analytics for Enterprise Class Hadoop and Streaming Data*. McGraw-Hill Osborne Media, 2011. [Available online[52]].

[15] Leigh Dodds and Ian Davis. *Linked data patterns - a patter n catalogue for modeling, publishing, and consuming Linked Data*. Creative Commons Attribution 2.0 UK, England & Wales, May 2012. [Available online[53]].

[16] Orri Erling. Advances in Virtuoso RDF triple storage (bitmap indexing). Technical report, OpenLink Software, USA, 2006. [Available online[54]].

[17] Katarina Grolinger, Michael Hayes, Wilson A. Higashino, Alexandra L'Heureux, and David S. Allison. Challenges for MapReduce in Big Data. In *Proceedings of the 10th IEEE World Congress on Services (SERVICES'14)*. IEEE Computer Society, June–July 2014. [Available online[55]].

[18] Rafiqul Haque and Mohand-Saïd Hacid. Blinked data: Concept, characteristics, and challenges. In *Proceedings of Service Congress 2014 (to appear)*, 2014.

[19] Andreas Harth, Jürgen Umbrich, Aidan Hogan, and Stefan Decker. YARS2: A federated repository for querying graph structured data from the web. In *Proceedings of the 6th International Semantic Web Conference and 2nd Asian Conference on Asian Semantic Web Conference (ISWC'07/ASWC'07)*, pages 211–224, Busan, Korea, 2007. Springer-Verlag. [Available online[56]].

[20] Olaf Hartig, Christian Bizer, and Johann-Christoph Freytag. Executing SPARQL queries over the web of Linked Data. In *Proceedings of the 8th International Semantic Web Conference (ISWC'09)*, pages 293–309, Chantilly, VA (USA), 2009. Springer-Verlag. [Available online[57]].

[21] 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[58]].

[49] http://www.w3.org/wiki/LargeTripleStores
[50] http://www.oracle.com/.../spatial-and-graph-wp-12c-1896143.pdf
[51] http://static.googleusercontent.com/media/.../archive/mapreduce-osdi04.pdf
[52] http://public.dhe.ibm.com/common/ssi/ecm/en/iml14296usen/IML14296USEN.PDF
[53] http://patterns.dataincubator.org/book/linked-data-patterns.pdf
[54] http://virtuoso.openlinksw.com/wiki/main/Main/VOSBitmapIndexing
[55] http://ir.lib.uwo.ca/cgi/viewcontent.cgi?article=1095&context=electricalpub
[56] http://aidanhogan.com/docs/iswc2007.pdf
[57] https://www2.informatik.hu-berlin.de/.../HartigEtAl_QueryTheWeb_ISWC09_Preprint.pdf
[58] http://adsabs.harvard.edu/abs/2009LNCS.5931..680F

[22] Shridevika Maharajan. Performance of native SPARQL query processors. Master's thesis, Uppsala university, Department of Information Technology, Uppsala, Sweden, May 2012. [Available online[59]].

[23] Thomas Neumann and Gerhard Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1):91–113, February 2010.

[24] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD'08)*, pages 1099–1110, Vancouver, BC (Canada), 2008. ACM. [Available online[60]].

[25] Nikolaos Papailiou, Ioannis Konstantinou, Dimitrios Tsoumakos, Panagiotis Karras, and Nectarios Koziris. H2RDF+: High-performance distributed joins over large-scale RDF graphs. In Tsau Young Lin, Vijay Raghavan, and Benjamin Wah, editors, *Proceedings of the IEEE International Conference on Big Data*, pages 255–263, Santa Clara, CA (USA), October 2013. IEEE Computer Society. [Available online[61]].

[26] Bastian Quilitz and Ulf Leser. Querying distributed RDF data sources with SPARQL. In *Proceedings of the 5th European Semantic Web Conference on The Semantic Web: Research and Applications (ESWC'08)*, pages 524–538, Tenerife, Canary Islands (Spain), 2008. Springer-Verlag. [Available online[62]].

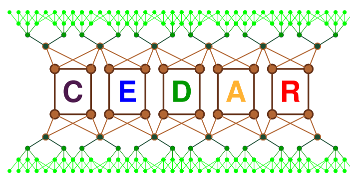
[27] Kurt Rohloff and Richard E. Schantz. High-performance, massively scalable distributed systems using the MapReduce software framework: The SHARD triple-store. In *Programming Support Innovations for Emerging Distributed Applications*, New York, NY, USA, October 2010. ACM. [Available online[63]].

[28] Alexander Schätzle, Martin Przyjaciel-Zablocki, and Georg Lausen. PigSPARQL: Mapping SPARQL to Pig Latin. In Roberto De Virgilio, Fausto Giunchiglia, and Letizia Tanca, editors, *Proceedings of the International Workshop on Semantic Web Information Management (SWIM'11)*, pages 4:1–4:8, Athens, Greece, June 2011. ACM. [Available online[64]].

[29] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Proceedings of the 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST'10)*, pages 1–10, Washington, DC (USA), 2010. IEEE Computer Society. [Available online[65]].

[30] Ying Yan, Chen Wang, Aoying Zhou, Weining Qian, Li Ma, and Yue Pan. Efficient indices using graph partitioning in RDF triple stores. In *Proceedings of the 25th IEEE International Conference on Data Engineering (ICDE'09)*, pages 1263–1266, Washington, DC, USA, March 2009. IEEE Computer Society.

[59] http://www.it.uu.se/research/group/udbl/Theses/Shridevika.MaharajanMSc.pdf
[60] http://infolab.stanford.edu/~usriv/papers/pig-latin.pdf
[61] http://www.cslab.ece.ntua.gr/~ikons/h2rdf+bigdata.pdf
[62] https://www.sar.informatik.hu-berlin.de/wbi/research/publications/2008/DARQ-FINAL.pdf
[63] http://www.avometric.com/papers/2010/Rohloff_Schantz_PsiEta_2010.pdf
[64] http://www2.informatik.uni-freiburg.de/~schaetzl/papers/PigSPARQL_SWIM2011.pdf
[65] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.178.989



Technical Report Number 7

CedTMart

Minwei Chen, Rafiqul Haque, Mohand-Saïd Hacid

July 2014