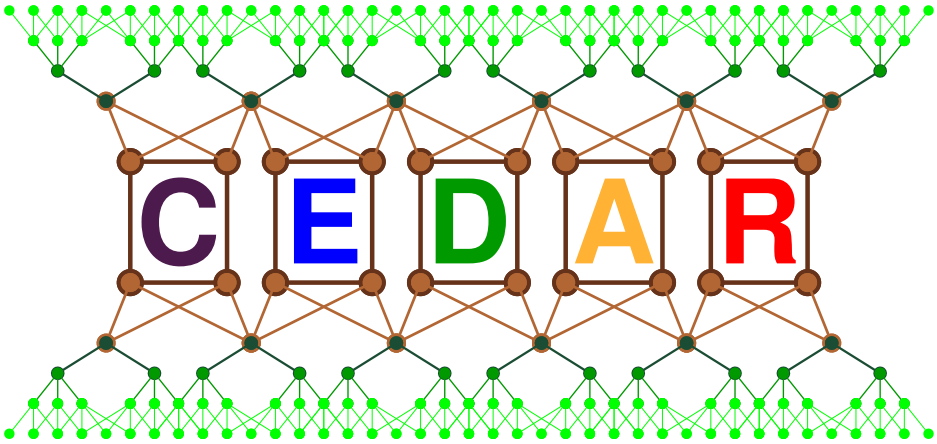


Technical Report Number 15

Development of the CedTMart Query Processor

Minwei Chen, Rafiqul Haque, and Mohand-Saïd Hacid

November 2014

Publication Note

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.chen@univ-lyon1.fr
akm-rafiqul.haque@univ-lyon1.fr
mohand-said.hacid@univ-lyon1.fr

$\mathcal{CEDAR}$ Project's Web Site: cedar.liris.cnrs.fr

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

CEDAR **Technical Report Number 15**

Development of the CedTMart Query Processor

Minwei Chen, Rafiqul Haque, and Mohand-Saïd Hacid

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

November 2014

Abstract

CedTMart [3] is a framework for processing RDF (Resource Description Framework) data and complex multi-join SPARQL queries. It comprises three distinct phases: preprocessing, distribution, and query processing. In [4], we reported the detail of the first two phases. In this report, we discuss the query processing phase. We provide a description of various techniques and methods which we implemented in this triplestore to ensures high-performance in terms of processing queries.

Table of Contents

1 Introduction

In this technical report, we describe the query processing phase of the CedTMart triple-store [4]. It provides a detailed description of various query processing techniques that are developed in this research.

This report is organized as follows. Section 2 provides a brief description of CedT-Mart. A description of SPARQL is provided in Section 3. Section 4 describes the query processor. The experiment results are provided in Section 5. Section 6 draws a conclusion and outlines future work.

2 CedTmart - In a Nutshell

CedTMart is a triplestore for storing and querying large-scale datasets distributed in a cluster of nodes. The goal of this triplestore is to guarantee scalability and high-performance.

It comprises three phases: *pre-processing*, *distribution*, and *query processing*. Different tasks are performed in these phases. Data are cleaned, converted partitioned, and compressed in the pre-processing phase. In the distribution phase, the triplestore performs a comparison between two subsets of data. Also, in this phase, data are partitioned into blocks and distributed to the nodes in a cluster. Queries are processed in the final phase. Figure 1 shows the architecture of CedTMart comprising these phases.

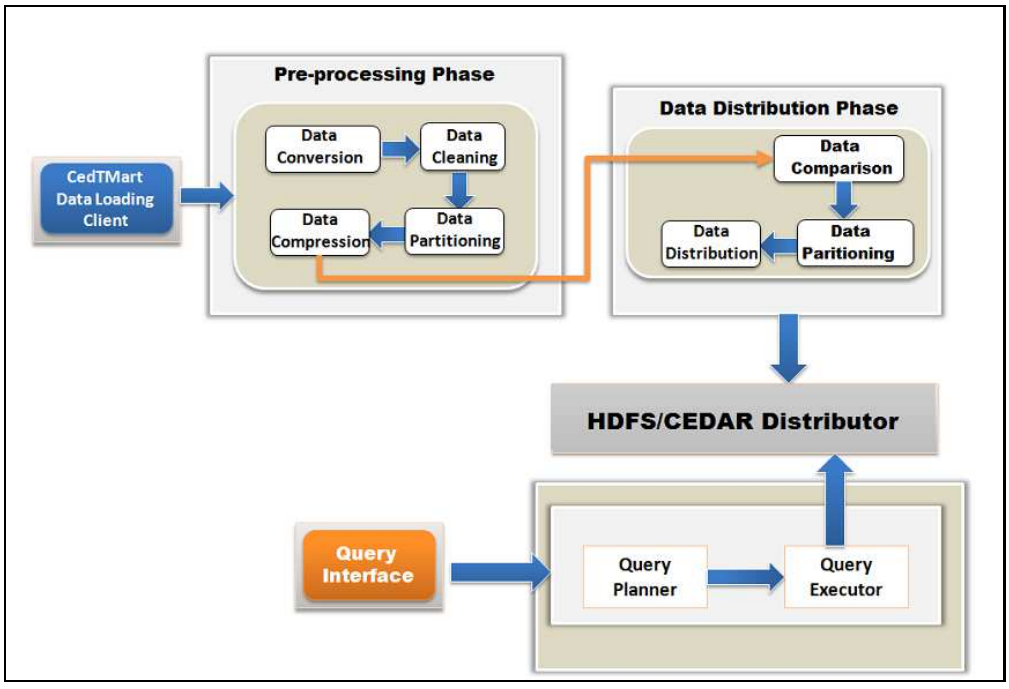


Figure 1: The Architecture of CedTMart (Source [4])

A detailed description of pre-processing and distribution phases are provided in [4].

We describe the query processing phase in this report.

3 Preliminary

Several RDF query languages are available such as, DQL [13], N3QL [7], Versa [10], RDFQ [6], RDQL [9], and SPARQL. Among these, SPARQL is the most widely used language recommended by the World Wide Web Consortium (W3C). [1] Since CedTMart process SPARQL query, we provide a brief description of SPARQL in this section.

3.1 SPARQL

SPARQL [cite] stands for SPARQL Protocol and RDF Query Language. It is a widely used language for expressing queries. A SPARQL query expression contains clauses (*e.g.*, `SELECT`, `WHERE`, `FROM` *etc.*.) and triple patterns. It may contain OPTIONAL pattern and modifiers (*e.g.*, `GROUP BY`). A triple pattern is essentially a triple which consists of *subject*, *predicate*, and *object*.

SPARQL offers four different types of query forms, `SELECT`, `CONSTRUCT`, `ASK`, and `DESCRIBE` for expressing different types of queries. The `SELECT` query returns RDF graphs, `CONSTRUCT` returns an RDF graph, `ASK` returns a `Boolean` value which denotes whether or not a triple pattern matches, and `DESCRIBE` returns a RDF graph that describes a resource [cite].

A SPAQRL query can be represented as a SPARQL graph (this is why we use the terms SPARQL query and SPARQL graph interchangeably in this report). The triple patterns contained in a query compose SPARQL graph. The subjects and objects represent the nodes and the labelled edges are predicates. Processing a SPARQL query is essentially a process of matching a SPARQL graph against RDF graphs. Figure 2 shows an example of a SPARQL graph. Note that, a SPARQL graph may contain a

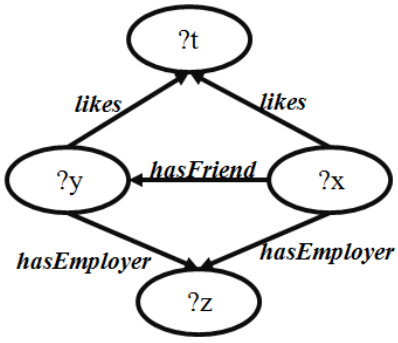


Figure 2: An Example of SPARQL Query

blank node which is a node without any URI (Uniform Resource Indicator) or literal. Blank nodes are also called *anonymous nodes*.

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

4 Query Processing with CedTMart

The CedTMart query processor comprises a *query analyser*, a *query planner*, and a *query executor*. We describe these components in this section. It is worth noting that in the previous version of our triplestore the query analyzer was integrated in the query planner. We decoupled it from the planner to enable the system to carry out analysis rigorously.

4.1 Query Analyser

A query must be analysed carefully to devise an efficient query plan. We developed the query analyzer to perform analysis efficiently. It carries out two types analysis: *statistical analysis* and *analysis of the characteristics* of queries. The former concerns with statistical information of queries such as, *number of variables* in a triple pattern. The latter concerns with the nature of queries such as, *form of queries*, which influences significantly the planning of query execution. The analysis results in a set of meta-data stored in a file.

Consider the example of SPARQL query shown below is submitted to CedTMart.

```
SELECT ?x WHERE {
        ?x hasFriend ?y.
        ?x hasEmployer ?z.
        ?y hasEmployer ?z.
        ?x likes ?t.
        ?y likes ?t.
        ?t hasAuthor Martin.
        Promod hasFriend Martin
}
```

The triplestore parses the query and then performs analysis. During statistical analysis, the analyser counts number of variables (prefixed with `?`. In this report, we call this type of variable *result variables*) associating with the result clause `SELECT`. For instance, in the above example, the analyzer finds one result variable that is, `?x`. Then the analyser count the number of triple patterns inside the graph pattern (inside the `WHERE clause`). Also, it counts the number of variables each triple pattern is containing. For instance, the query in the above example has seven triple patterns. Each of the first five triple patterns contains two variables. The sixth pattern <"`?t hasAuthor Martin`"> contains only one variable and the last one has no variable.

Considering the analysis of the nature of queries, the key purpose is to determine the *level of complexity*. We use qualitative metrics, *low*, *medium*, *high*, *X-high* (denotes eXtremely high) to measure the complexity. During this analysis, the analyser investigates the query forms first. According to our observation, the simplest form of a SPARQL query is `ASK`. The other forms include `SELECT`, `CONSTRUCT`, and `DESCRIBE`, are relatively more complex than this. However, an `ASK` query may promote complexity depending on the number of triple patterns it should match. The complexity of a

SPARQL query can be *high* if it contains more than one query form. For instance, if a SPARQL expression contains `SELECT` and `DESCRIBE` query forms, it will be more complex than a query with `SELECT` form only.

Then, the query analyser analyses the result variable. However, note that, it can be a '*' instead of a result variable such as `?x`. The complexity of *star queries* can be lower than the queries with results variable(s) since in this type of queries, it is easier to find the target RDF graphs from a distributed dataset. Additionally, typically, the star queries contains dataset definition clause that is, FROM clause, which essentially simplifies the queries significantly. Conversely, in order to retrieve a RDF graph corresponding to a result variable, the query processing engine may need to scan all the lines/rows of the complete dataset.

The queries with dataset definition are *simple*. The dataset definition enables targeting specific data location, which is critical to reduce query processing time in a distributed environment.

Furthermore, the analyser analyses the number of triple patterns inside the graph pattern of queries. The analyzer counts the number during statistical analysis and uses it to calculate the number of JOINs in queries. The number of JOINS has a positive correlation with the complexity of queries. To sum up, the level of complexity of a query relies on the following attributes:

- number of query form contained in the query,
- type of query form,
- the type of result variable (star or regular variable),
- number of result variable contained in the query
- availability of data set definition (FROM Clause)
- number of triple patterns,
- availability of query pattern,
- availability of modifier
- number of modifier

4.2 Query Planner

The key purpose of the query planner is to devise an efficient execution plan which guides the query execution. It uses the outcomes produced by the analyser. The planner carries out three functions: *SPARQL Graph Partitioning*, *SPARQL Sub-graph Ranking*, *Cost Calculation*, and *Generating Execution Plan*.

4.2.1 SPARQL Graph Partitioning

Query partitioning is a process of decomposing a SPARQL graph into sub-graphs. The planner partitions a SPARQL graph by number of variables each triple pattern contains.

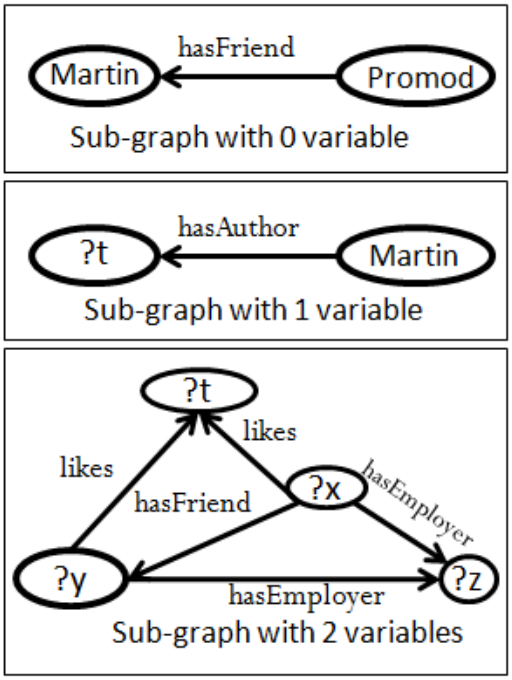


Figure 3: An Example of Query Partitioning.

For instance, the example provided in Section 4.1 can be partitioned as shown in Figure 3.

This figure depicts that the query is partitioned into three sub-graphs. The first partition is a subgraph composed of a triple pattern with zero variable. The second subgraph contains a triple pattern with one variable and the final partition is a subgraph composed of triple patterns with two variables.

4.2.2 SPARQL Sub-graph Ranking

Sub-graph ranking is a process of ordering sub-graphs that were generated in the previous step. The sub-graph ranking process is straightforward. The planner defines the order of sub-graphs by the number of variables. For instance, a sub-graph with 0 variable holds the first position. Eventually, it produces a list of sub-graphs.

In Figure 4, a SPARQL graph G^S is defined as a set of ordered sub-graphs (SG_1, SG_2.. SG_n). The table presents ranking of sub-graphs.

4.2.3 Cost Calculation

Cost calculation is a process of computing the cost of executing queries. In this research, we developed a cost model, which is a sum of three different cost patterns with

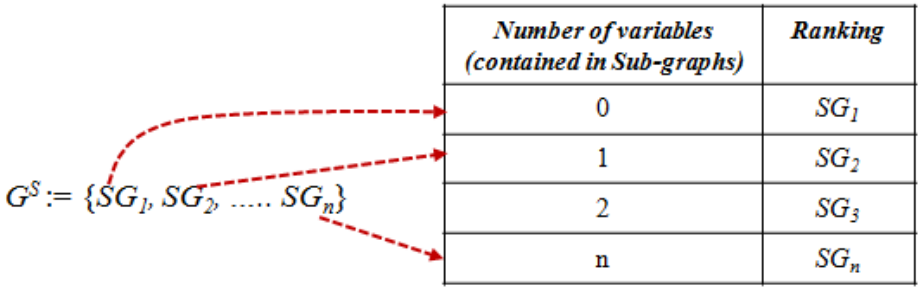


Figure 4: Query Sub-graph ranking

their co-efficient. We define a cost function to estimate the cost of executing queries.

$$C^Q = \sum(\alpha C^T + \beta C^P + \gamma C^V) \tag{4.2.1}$$

Where:

- C^Q denotes the total execution cost,
- C^T, C^P, and C^V denote *Cost of Triple Patterns*, *Cost of Predicate*, and *Cost of variable* respectively, and
- α, β, and γ denote these co-efficients of the cost patterns.

  The values of these coefficients (shown in equation 4.2.1) are user defined and depend on the following parameters:

  - *Size of the predicates* refers to the size of the predicate files after carrying out the predicate partitioning on a dataset. If the size of a predicate is bigger (let's say 80% of the dataset) than others, the value of β should be large. It is worth noting that in the preprocessing phase, our triplestore calculates the size of each predicates produced by carrying out the predicate partitioning operation.
  - *Cost of triple patterns*, consider a query with two triple patterns and the costs of these triple patterns, let's say 1 and 5 respectively. In such cases, the value of α should be large.

The cost patterns are briefly explained in the following:

- *Cost of Triple Pattern* (C^T): The number of variable(s) represents the *weight* of triples patterns. The weight determines the execution cost of triple patterns. For instance, the weight of a triple pattern `<?x hasFriend ?y>` is '2' because it contains variables. While processing, this triple pattern will consumes more resources than a triple pattern `<?t hasAuthor Martin>` whose weight is '1'. The weight of a triple pattern can be '0'. For instance, `<Promod hasFriend Martin>` do not contain any variable. Such triple patterns will consume lesser resources than other triple patterns in a query.

- *Cost of Predicate* (C^P): It is determined by the weight of predicates. The weight of predicates is determined by their *size* (after partitioning datasets in the preprocessing phase.)and *the cost of subject and/or object corresponding to a predicate*.

  The size of a predicate depends on the number of Subject-Object (S-O) pairs that can range from some KBs to GBs. Querying a smaller size predicate is always faster than a larger one. The cost of subject and/or object is determined the number of entries (subjects or objects) corresponding to the (subject) variables or (object) variable of a triple pattern. Consider a triple pattern `<?t hasAuthor Martin>`; first the size of the predicate `hasAuthor` is considered and then the number of entries corresponding to the row `Martin` in the O-S matrix (the matrix is created in the preprocessing phase by the BitMat [1] compressor. See [4]) for more detail.) Note that, in an extreme case, a row can contain millions of entries. Thus, the partial execution time of triple patterns may vary from milliseconds to minutes. In a word, the parameter *number of entries* has a positive correlation with the time to fetch them.

- *Cost of Variable* (C^V): The variables in SPARQL graphs are used in JOIN operations which are in fact the most complicated operations both in relational and graph database. The complexity of JOIN queries is determined by the number of times a variable appears in different triple patterns. For example, a variable `?x` can be contained in a triple pattern with one variable and two other triple patterns with two variables. This is the key to measure the cost of variable, which according to our view critical for estimating the cost of executing SPARQL queries. Additionally, we found that the cost of variable relies heavily on cost of predicate. Taking these aspects into consideration, we developed the following cost function,

$$C^V = \sum (K * \sum_{i=1}^{n} (C^P(t_i))) \tag{4.2.2}$$

  Where:

  - K denotes number of times a variable occurs in the number of triple patterns

Note the, the value of C^P is always integer.

4.2.4 Generating Execution Plan

The execution plan is a *specification for executing queries*. The planner generates this specification based on the estimated costs calculated in the previous step. An execution plan specifies the order executing parts of queries (subgraphs) and how to join results produced by the subgraphs.

Unlike other triplestores such as, HadoopRDF [12] and RDFPig [5], CedTMart generates only one plan. The key notion is to reduce the time required to generate more than one plans and then compare them using techniques such as *round-robin scheduling* to find the best plan.

4.3 Query Executor

CedTMart provides two distinct modes of executing queries: the *centralized execution* and *distributed execution*. The former enables to run queries on a single host whereas in the distribution execution mode, the queries are sent to a cluster of nodes depending on the locations of corresponding data items. The query executor carries out the following steps to process a query,

- STEP 1: From the query launcher, a query is sent to the parser which parses the query into several parts: the query form (*e.g.*`SELECT`), the variables, the triple patterns, data definition clause (`FROM`) and other elements such as, modifiers `FILTER` and `PREFIX`).
- STEP 2: The executor distributes the parts(subgrapgs) of a given query to data node(s) depending on the data location. Note that, in case of centralized execution mode, there is no notion of data node.
- STEP 3: If the `rdf:type` of subgraph is literal, they converted to numerics by assigning an `ID`. For instance, a query `<?x hasAuthor Martin>` is transformed into `<?x isStudent 6319>` where, 6319 is the ID of Martin. The ID is retrieved from the (key,value) indexing store that is hosted on the same node. Then, the data is fetched from the corresponding data block. In the next step, the fetched data is converted back to the literal by querying the key-value indexing store. Finally, the results of the subgraph are returned (`Boolean`, if no variable)
- STEP 4: In the final step, the executor combines results of subgraphs based on variables and returns a result-set or NULL

4.4 Query Optimization

We have developed various optimization techniques for different phases of CedTMart to ensure high-performance in processing complex queries. In preprocessing phase, we compress data using the *Bitmat* technique. In addition, we used D-Gap [2] compression technique which enables querying compressed dataset. This prevents data read from the secondary storage and enables loading dataset onto main memory, which increases query processing speed by reducing. We developed a comparison technique for distributing data intelligently within a cluster of nodes. This essentially reduces the communication costs between the nodes in a cluster. Additionally, the cost models which we developed to generate a highly-efficient query plan, enables processing complex queries faster.

Locating exact data items for instance, a line in a huge file is critical to performance in terms of processing queries. In order to locate data efficiently we developed an indexing technique relying on *Binary Tree* and *Pointer List*. In the preprocessing phase, the compressed sets of data (predicate files) are sliced into variable sized small chunks and the metadata is generated. The metadata contains different information such as, *size of data blocks* and *offset (tail) of data blocks*. These chunks are then distributed to data nodes along with their metadata. A listener program on the data node side loads the meta-data into a temporary in-memory data structure: either a binary tree or a pointer list. Once a subgraph arrives, the program can locate the correct data block(s) and fetch the on-demand data faster.

5 Evaluation

We conducted a small-scale experiment with our query processer. It is worth noting that in this experiment we used the Lehigh University Benchmark(LUBM) [11] dataset that was preprocessed using our triplestore (A detailed information of preprocessing result is available in [4]). We experimented the CedTMart triplestore on distributed execution mode. The cluster consists of three virtual machines (VMs) which are instantiated from the machine with the following specification.

- Processor: I7 960 quad core processor with Hyper-Threading Technology,
- Memory: 16GB main memory
- HDD: 1TB Seagate 7200.12

CedTMart offers HDFS[8] and CEDAR distributor for distributing datasets. For this experiment, we used CEDAR distributor for this experiment. Note that, it relies on *local file system* such as, BTRFS. [2]

We conducted three experiments with a small dataset to test the performance of CedTMart. The experiments are described briefly in the following:

- *Experiment 1*: This experiment relies on the query plan which uses *cost of triple pattern* cost model.
- *Experiment 2*: This experiment relies on the query plan which combines all three cost models. However, it uses non-blocking data storing method. In this method, the subsets of the datasets (predicates files) are stored without partitioning them any further.
- *Experiment 3*: Like the second experiment, the final experiment relies on the query plan that combines all of the three cost models. However, it uses *indexed data blocking* optimization.

[2] BTRFS:https://btrfs.wiki.kernel.org/index.php/Main_Page

Since we conducted experiments with the LUBM dataset, we used queries proposed by the benchmark. Due to the restricted main memory size, we carried out experiments on 2GB dataset. The outcomes of the experiments are presented in Table 1.

Table 1: Query Execution Results

Executor	Elapsed Time (in millisecond)
C^T	2023
$(C^T + C^P + C^V)$	1792
$(C^T + C^P + C^V)$ & Indexed Data Block	503

The table shows that the combination of our cost models and indexed data block technique improved performance significantly. However, more experiments with massive-scale datasets is necessary to validate the performance of CedTMart. Note than, we have not used HDFS in our experiments, because it promotes latency of *random read* due to its *internal functionalities*.

6 Conclusion & Future Work

In this technical report, we described the query processing phase of CedTMart triplstore. We provided a detailed explanation of the query processing techniques which we have developed in this research. We described the query optimization techniques which has been developed to improve the performance of CedTMart. We provided the results of the experiments which we conducted to evaluate the performance of the triplestore. The outcomes showed that the cost models are not sufficient to optimize the performance of complex queries in distributed environment. The indexing of data blocks is a very efficient technique for reducing query processing time.

We should conduct more experiments and perform a comparative analysis with existing triplestores. Also, we need to refine our techniques through experimenting the triplestore. Since we integrated HDFS with our triplstore and it guarantees a highly scalable and fault-tolerant environment, we should develop a MapReduce-based query executor to investigate the performance of our triplestore in the Hadoop ecosystem.

References

[1] Medha Atre, Vineet Chaoji, and Mohammed J. Zaki. Bitpath – label order constrained reachability queries over large graphs. *Computing Research Repository (CoRR)*, abs/1203.2886, 2012. [See online[3]].

[2] Jinlin Chen, Ping Zhong, and T. Cook. Compressing inverted file index using mixed delta/flat binary code. In *Proceedings of the 1st International Conference on Digital Information Management*, pages 338–343, December 2007.

[3] 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.

[4] Minwei Chen, Rafiqul Haque, and Mohand-Saïd Hacid. CedTMart—a triplestore for storing and querying blinked data. CEDAR Technical Report Number 7, Université Claude Bernard Lyon 1, Lyon, France, July 2014. [See online[4]].

[5] Long Cheng, Spyros Kotoulas, Tomas E. Ward, and Georgios Theodoropoulos. Robust and skew-resistant parallel joins in shared-nothing systems. In *Proceedings of the 23rd ACM International Conference on Conference on Information and Knowledge Management*, CIKM'14, pages 1399–1408, New York, NY, USA, 2014. ACM. [See online[5]].

[6] World Wide Consortium. RDFQ: RDF query (v2.0). [See online[6]].

[7] World Wide Web Consortium. N3QL—RDF data query language. [See online[7]].

[8] Apache Software Foundation. Apache software foundation. [See online[8]].

[9] Inc. Fourthought. RDQL—a query language for RDF. [See online[9]].

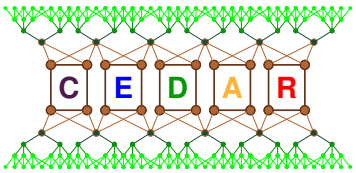
[10] Inc. Fourthought. Versa query language. [See online[10]].

[11] Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. LUBM: A benchmark for OWL knowledge base systems. *Web Semantics: Science, Services and Agents on the World Wide Web*, 3(2–3):158–182, 2005.

[12] 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. [See online[11]].

[13] The Doctrine Project. Doctrine query language. [See online[12]].

[3] http://arxiv.org/pdf/1203.2886v1.pdf
[4] http://cedar.liris.cnrs.fr/interns/MinweiChen/ctr7.pdf
[5] http://doi.acm.org/10.1145/2661829.2661888
[6] http://lists.w3.org/Archives/Public/www-rdf-rules/2003Mar/att-0021/RDFQ.html
[7] http://www.w3.org/DesignIssues/N3QL.html
[8] http://hadoop.apache.org/
[9] http://www.w3.org/Submission/RDQL/
[10] http://xml3k.org/Versa
[11] http://adsabs.harvard.edu/abs/2009LNCS.5931..680F
[12] http://www.doctrine-project.org/



Technical Report Number 15

Development of the CedTMart Query Processor

Minwei Chen, Rafiqul Haque, and Mohand-Saïd Hacid

November 2014