ABSTRACT

MashQL is a query-by-diagram mashup language, which collects web data that are expressed in a Resource Description Framework (RDF) and stores them into a backend database, allowing people to query it very easily. MashQL assumes that web data sources are represented in RDF and it can be inquired using a SPARQL query language. Resource Description Framework (RDF) is a language for representing information (metadata) about resources in the World Wide Web[10]. In this paper we present the design and implementation of two important modules of the MashQL, the RDF Loader, which downloads and loads RDF data from the web into an Oracle's RDF model database and the Query Optimizer, which is designed for the purpose of executing all MashQL's queries successfully, efficiently and in a timely fashion. With the RDF Loader, we achieved to design and implement a concrete system that includes a combination of the market's lasted technologies that exist in the Extract-Transform-Load (ETL) process for RDF data, such as Oracle, Java and Jena. On the basis of these technologies, we created a powerful, stable and intelligent RDF loader that loads any RDF data in any format and of any size in a very short time. The Query Optimizer, implements our optimization solution in order to provide MashQL's queries with the highest speed performance execution. Our optimization solution includes the creation of data summaries on top of the RDF data have are already been loaded onto the database and the BR-Algorithm that catches queries' results regarding the most important MashQL's queries. Using the database summaries we have the advantage of, instead of scanning and sorting all the data during the query's execution course, the data have already been sorted and pre-computed. This focuses on MashQL's queries requirements matter. By using the BR-algorithm the most important MashQL's queries acquired high response time, since their results are had already been caught in the database. For the highest algorithm's performance execution course, we achieved to reduce 3 times in average the original graph's size by dividing it in three parts using our graph's partitioning novel idea. This partition concept helps the BR-algorithm to run faster ,producing less and more carefully caught data. Finally, our optimization solution against MashQL's queries has been compared with Oracle's corresponding technology and it presents very good results. More concretely, our solution is performing 10 times faster in MashQL queries and 45 times faster concerning the MashQL's most important queries.

CHALLENGES IN BUILDING AN EFFICIENT RELATIONAL ARCHITECTURE FOR MASHQL

Michael A. Georgiou

A Thesis
Submitted in Partial Fulfilment of the
Requirements for the Degree of
Master of Science
at the
University of Cyprus

Recommended for Acceptance
by the Department of Computer Science
DEC 2010

APPROVAL PAGE

Master of Science Thesis

CHALLENGES IN BUILDING AN EFFICIENT RELATIONAL ARCHITECTURE FOR MASHQL

Presented by

Michael A. Georgiou

Research Supervisor _____

Research Supervisor's Name

Committee Member _____

Committee Member's Name

Committee Member _____

Committee Member's Name

University of Cyprus

DEC 2010

CREDITS

TABLE OF CONTENDS

LIST OF TABLES

LIST OF FIGURES

Chapter 2

Chapter 3

Chapter 4

Chapter 5

Chapter 6

Chapter 1

Introduction

1.1 Motivation and Challenges

Mashups are web 2.0 new features applications. Mashups are used to collect, combine or syndicate data or functionality from more than one web sources in order to create a new service. Mashups are implemented based on various web 2.0 technologies such as Really Simple Syndication feeds (RSS), Application Programs Interfaces (APIs) and Web Services (WS). An example of Mashup can be an integration of business addresses and online maps so that you could quickly see where all the bookstores or hospitals are located in your neighborhood[14].

MashQL a Query-by-Diagram language uses the Mashups system to provide a general-purpose data retrieval on top of Web 2.0. MashQL regards the internet as a database, where a data source is seen as a table and a Mashup as a query. MashQL assumes that web data sources are represented in RDF and it can be inquired using a SPARQL query language[1]. Resource Description Framework (RDF) is a language for representing information (metadata) about resources in the World Wide Web[10].

MashQL consists of the RDF Loader and the Query module. The RDF Loader downloads and loads RDF data from data source[s] into an Oracle's RDF model. A query module consists of a Query Language and Query Formulation Algorithm. The former is a query language which supports all constructs of SPARQL and the latter is used by the MashQL editor for query formulation. MashQL's Query Formulation Algorithm formalizes a background query in each interaction in such a way that users can navigate and query a data graph, without prior knowledge about it. To achieve that, MashQL's Query Formulation Algorithm defines seventeen types of queries, those queries are called background queries[1,2,3,4,5].

Our targets are to design and implement the RDF loader module and to improve the performance execution for all MashQL's background queries using Oracle's 11g RDF engine.

MashQL uses the Oracle's RDF model as a backend database in order to store and retrieve RDF data. Since the backend database technology was determined to be an Oracle , we are obliged to use loading and retrieving technologies that are supported by Oracle. Unfortunately , this fact prevent us from joining or using other similar technologies, thus ,our study in other technologies is poor. Also, Oracle's software is closed to any optimization, thus we focus only on the optimization solutions and technologies that are recommend by Oracle.

MashQL's RDF loader must be capable of loading RDF data from any RDF compatible format such as RDF/XML, N3, NT etc. apart from Oracle's RDF model database. Since Oracle's loader supports only NT format, we use Jena's parser to convert any RDF data to NT format.

Additionally, MashQL's background queries need to be executed on the whole dataset. In reality, the loading time is a key factor and should be short. Thus, the RDF loader is designed to load RDF resources very fast, using Oracle's bulk-loading technology. In addition, the RDF loader must be able to handle a number of RDF data sources simultaneously. As a consequence ,the RDF loader provides a queuing mechanism for asynchronous loading and notification services to MashQL. Also, the RDF Loader must be able to maintain a mechanism to periodically refresh all the stated data sources transparently, due to the fact that, the loaded RDF data sources have the particularity in that their contents may be modified at random time. Finally, due to the importance of the role that it played in the remainder of the system, the RDF loader must be distinguished from having fault-tolerance and High availability capabilities. The system must be continue to operate properly in the event of failure of some of its components and must be implemented primarily for the purpose of improving the availability of services.

MashQL's background queries need to be executed on the whole dataset in real-time situations. Thus, their response time should be short. Achieving such a short interaction time for these types of queries is very challenging for the following reasons:

Firstly, the Oracle's RDF technology is closed to any customary optimizations. Oracle's RDF model uses SQL-based scheme for querying RDF data using the SEM_MATCH table function which has the ability to search for a random RDF graph. Oracle's SQL-based scheme integrates RDF queries into Oracle's SQL queries. Oracle's SQL-based scheme has the advantage of querying RDF data in the same way as it queries traditional relative data. Thus, any optimization against SQL-based scheme can happen only at a whole SQL query and not exclusively for RDF queries , thus , the SEM_MATCH table function cannot be optimized and it is considered as a black box[13].

Secondly, the RDF data is distributed as a directed graph. The graph's adjacency matrix is stored in the database as a related table producing a high degree of data redundancy. Since the database structure is not normalized into smaller objects the database's inquiries for huge RDF graphs have a very poor response time[10].

Finally RDF queries in general , suffer from intensive self-joint that involve more such queries than the majority of RDF queries. For example , if a SPARQL query defines three SPARQL patterns, even though the query is executed as one unit, internally each pattern needs to select the graph's data separately. Internally, the RDBMS manages this situation by joining the graph's data as match to the number of RDBMS patterns (that are) defined by the query. The problem is increasing exponentially for each new SPARQL pattern that is added to the query. Also, the queries are not completed, or their response time is very slow , when the queries need to query graphs that contain a million or billion triples and the queries contain a big number of SPARQL patterns.

Our optimization solution tries to eliminate and solve the above performance issues by using the Query Optimizer(QO) module. The Query Optimizer is a responsible module which executes all MashQL's queries in an optimum way. The Query Optimizer executes all the formulated queries (the final queries) using oracle's technology and the all MashQL's background queries using the Query Optimizer's optimization solution. Additionally , the Query Optimizer is responsible to map the RDF MashQL background query with the corresponding optimized SQL-Based MashQL background query. The mapping between the

RDF-Query and SQL-based query is achieved by using a rule table. The Query Optimizer maintains a rule table of two columns, the first column stores the SQL-BASED template identifier, and the second column contains the actual SQL-Based query template that will run on the database.

Our optimization solution stands on two database techniques:

1. It creates smaller and focuses on data sets using data summaries providing background queries with a faster access to the data and less scanning data.
2. We designed the BR-algorithm with graph-partitioning which fetches beforehand queries' results and stores these results into the database in order to bypass self-joint operations during the queries' execution.

Based on the above techniques, we divided MashQL's background queries into two categories according to their optimization solution.

In the first category belong the background queries 1 to 13 where their optimization solution is supported using the method of summaries. The summaries are created by using the technology of oracle's materialized views.

In the second category belong the background queries 14 to 17 where their optimization solution is supported using the BR-Algorithm.

1.2 Contributions

Summary of Contributions:

1. **RDF Loader:** The first contribution of this document is the creation of the RDF loader for the MashQL. Our RDF Loader is implemented on top of oracle's RDF model. It offers MashQL the capability of caching and asynchronously loading RDF data sources into an oracle's RDF model. In addition, RDF loader provides messaging and notification services to MashQL for the status of loading . The RDF loader keeps track on all data sources' metadata in a database repository that is accessible online by MashQL's administrators. Additionally, the RDF Loader can load any RDF format such as RDF/XML, NT and N3 formats. The RDF loader can load RDF files of unlimited size , but , for the best system performance we concluded that the RDF

loader will be loaded with RDF files up to 1.5G bytes. These files can be loaded in less than 30 minutes. This value is equal to 80000 triples per second. Additionally , the RDF Loader provides a scheduler module, which is responsible for periodically refreshing all the old data sources and resume any unsuccessful RDF data sources transparently.

2. **Query Optimizer for MashQL**: The second contribution of this document is to improve the MashQL's background queries execution time. Our Query Optimizer, for MashQL implements our optimization solution in order to provide MashQL's with queries at the highest speed performance execution. Our optimization solution includes the creation of database summaries on top of the RDF data that are loaded into the database and a data fetching beforehand solution, for the most important MashQL's queries using our BR-Algorithm. By using the database summaries we have the advantage that, instead of scanning and sorting all the data during the query's execution course, the data are already sorted and pre-computed, focusing on MashQL's queries requirements. By using the BR-algorithm the most important MashQL's queries acquired high response time, since their results are already fetched beforehand in the database. For the highest algorithm's performance standard, we achieved to reduce 3 times in average the RDF graph's size by dividing it in three parts using our graph's partitioning novel idea. This partition schema helps the BR-algorithm to run faster ,producing less and more careful fetch beforehand data. Finally, our optimization solution against MashQL's queries has been compared with Oracle's corresponding technology and it presents very good results. More concretely, our solution performs 10 times faster in MashQL queries and 45 times faster regarding the MashQL's important queries.
3. **Experimental results :** The third contribution of this document is to provide experimental results to the following :
    a. Loading time statistics for the RDF Loader against our benchmark .

b. Response time statistics for MashQL background queries using our benchmark and using the Oracle's SEM_MATCH table function and our optimization solution .

1.3 Paper Structure

The paper is laid down as follows :

Chapter 1: In this chapter, we mention motivation and challenges in this work. Also, we summarize the work done by describing in brief the system components. Finally we report on the contribution of this work.

Chapter 2: In this chapter , we present the Related Work and Technologies that exist today .We describe the structure of RDF technology and we provide examples from World Wide Web Consortium (W3C) for better concept comprehension. We explain the various formats that are used to convey RDF data such as RDF/XML , N3 and NT . Additionally, we describe the SPARQL Protocol and RDF Query Language (SPARQL) which provides a protocol and the query language to RDF. In addition , we present the Oracle 11g Semantic Technology which comprises the industry's first open, scalable, secure and reliable data management platform for RDF applications.

Chapter 3: This chapter describes the system architecture of MashQL Server which consists of the RDF Loader and the Query Optimizer. Section 3.1 describes a deep view of how the MashQL editor, the RDF Loader and the Query Optimizer are working together in order to maintain a robust RDF retrieval engine. Section 3.2 shows how the RDF Loader downloads and loads RDF data into the system. Section 3.3 expresses how the Query Optimizer executes the MashQL's queries efficiently. Both sections are supported by design details, components specifications and examples.

Chapter 4: In this chapter we explain in explicit detail the optimization solution that we provide concerning background queries, dividing MashQL's background queries into two

categories according to their optimization solution (General background queries and the N-level properties and N-level objects queries). We explain what database summaries are and how these summaries help general background queries to have a better performance standard. In addition, we explain what the BR-Algorithm is, and how this algorithm helps the N-level properties and N-level objects queries to run faster.

Chapter 5: In this chapter we present the Experimental Methodology that we used in order to implement the optimization solution that we proposed in chapter 4 . Since the Oracle 11g semantic technology has been chosen to be the MashQL's RDF engine, [1,2,3,4,5], our optimization solution's database objects are created and based on this technology. More analytically , in this chapter ,we mention information about the data summaries and BR-algorithm that we created on top of Oracle's technology. Additionally , we state the problems that we found during the BR-algorithm implementation and we provide the solutions that we found in order to solve these problems i.e. we present how our new partitioning schema against any RDF graphs helps BR-algorithm to be executed faster.

Chapter 6: In this chapter , we provide evaluation results regarding the RDF loader and the module that loads RDF data resources into the MashQL database. Also, we present comparison results for the performance of MashQL background queries using Oracle's SEM_MATCH and our optimization solution.

Chapter 7: In this chapter we mention our conclusions and future work challenges.

Appendix A: In this appendix , we show how to enable RDF technology in Oracle 11g.

Appendix B: In this appendix , we provide details of how to Load RDF data into an Oracle 11g Database using Oracle's SQL Loader, Oracle's native insert statement and Java.

Appendix C: In this appendix , we present a Summary of all Background Queries as well as the creation script for all Database summaries that is proposed in our optimization solution.

Appendix D: In this appendix , we give the evaluation results i.e. the real values that we show in the charts, chapter 6.

Appendix E: The last appendix , describes all the queries (in SQL code) that we have run in order to get our evaluation results.

Chapter 2

Background Work and Related Technologies

2.1 Background work

As we already mentioned in our introduction, the use of oracle as an RDF engine for storing and retrieving RDF data prevents us from paying attention to other technologies (that are) recommended in the computer science community ,since, our intention is not to compare the oracle's RDF technology with other technologies but , our targets are to design and implement the RDF loader module using the oracle's tools and oracle's compatible technologies such as java and Jena and to improve the performance execution standards concerning all MashQL's background queries that run on top of oracle's RDF database. Similar to Oracle Database 11g Semantic Technologies are the IBM DB2 database and various open source products such as the Sesame[22] and Jena[15] .

From the state-of-the-art we have investigated similar RDF loading systems, that are implemented in different technologies. We learned their input data, their components and their compatible technologies in order to compare them with ours . Also, we realized how the RDF data are extracted, transformed, and loaded (ETL) into an RDF engine.

Similar RDF loading solutions (ETL point of view) that are proposed at the academy is the RStar [8], an RDF storage and query system for enterprise resource management. RStar's data loader takes RDF/XML files as input and provides both the original and inferred triples to the backend database (IBM DB2). This is respectively realized on the RDF parser, triple importer and inference engine. RDF parser analyzes the statements of an RDF/XML file according to the RDF syntax specification and passes the resulting triples to the triple importer. The triple importer then inserts the triples into the backend database. In the same way, The R-DEVICE [9] system consists of two major components, the RDF loader/translator and the rule loader/translator. The former accepts from the user requests for loading specific RDF documents. The RDF triple loader downloads the RDF document from the Internet and uses the ARP parser to translate it to triples in the N-triple format. Both the RDF/XML and

RDF/N-triple files are stored locally for future reference. Additionally, the RDFPeers [7] a scalable distributed RDF repository ("RDFPeers") that stores each triple at three places in a multi-attribute addressable network by applying globally known hash functions to its subject, predicates, and objects. Thus, all nodes know which node is responsible for storing the triple values they are looking for, and both exact-match and range queries can be efficiently routed to those nodes. The RDFPeers's reads an RDF document, parses it into the RDF triples, and uses MAAN's STORE message to store the triples into the RDFPeers network. When an RDFPeer receives a STORE message, it stores the triples into its Local RDF Triple Storage component such as a related database.

Additionally, our research found that there are many difficulties to retrieved RDF from large RDF graphs. For example, after a SPARQL query is submitted to a relative database, the query engine might determine a nested-loop self-joint on the selected columns accordingly. Once the query and data are much more complex, the cost will increase dramatically[20]. Thus , from the state-of-the-art we have investigated how these problems are solved or reduced , and what techniques are applied in order to improve the queries' performance execution. This research helped us to adapt these techniques to our effort.

The techniques to creating indices, dividing data into property tables (2-column schema), and materializing joint views (e.g., subject-subject and subject-object) are common methods for improving RDF query performance on the vertical database structure [20]. C-Store [25] proposed to partition the RDF table vertically, a table (S,O) for each property. The RDF3X approach [24] proposed to build many Binary Tree indexes and "carefully optimize complex joint queries" [24]. Although these approaches have produced a good performance standard for small to medium graphs and low performance level on very large RDF graphs (millions or billions of triples).

Our data summaries , included in our optimization solution are based on these techniques with the difference that , we created data summaries with 1-column schema and summaries where their contents combine the subject-property and object-property filtering the properties of

specific RDF class (*rdf:type*) .We consider this schema as a ideal design to handle all MashQL's background queries in an efficient way.

In addition, there are many techniques (that are) proposed by the state-of-the-art that helps the performance execution of RDFs' queries in relation to very large graphs , including graph's partitioning, it is accompanied by a graph's signature or a semantic index, and by the creation of a graph's semantic indexes and graph's signatures . For example [22] a proposed graph partitioning technique, which it creates from the original graph a number of overlapping sub-graphs. The contents of the sub-graph are lexicalized, and for each sub-graph is created a virtual document that is added to a Vector Space Models (VSM). Finally, the VSM is used to create a semantic index, which determines the contextual similarities between graph nodes (e.g., URIs and literals).These similarities can be used for finding a ranked list of similar URIs/literals for a given input term which can be used.

A similar approach is proposed in GRIN [23] which creates an indexing mechanism in certain kinds of RDF queries, namely graph-based queries where there is a need to traverse edges in the graph determined by an RDF database. The index is created, based on the idea of drawing circles around selected "center" vertices in the graph where the circle would encompass those vertices in the graph that are within a given distance of the "center" vertex. The "center" vertices are used to identify the radius of the circles and then lever this in to building an index called GRIN.

Another partitioning approach is submitted to [20] which partitions the graph into multiple sub-graph pieces, stores them in a triple table with one more column of group identity, and builds up a signature tree to index them. Based on this infrastructure, a complex RDF query is decomposed into multiple pieces of sub-queries which could be easily filtered into some RDF groups using a signature tree index.

A graph's signature approach is proposed in [1], where its creates two graph signatures the O-Signature SO and the I-Signature SI according to node bisimilarity. The SO summarizes a graph by grouping nodes reachable through all outgoing paths. The SI summarizes a graph by grouping nodes reachable through all incoming paths. The SQL a query is evaluated on each

summary separately, and the intersection of the two answers is equal or a small superset of the target answer.

Graph signatures has the advantage of reducing the original graph size. On the other hand ,the time that is needed to create a signature and the performance that is gained by the queries is not enough to adapt it to our solution. We believe that the Graph partitioning that is proposed in the arts with a combination of an index or a graph signature can provide very good results in RDF's data retrieving especially for very large RDF graphs. Our solution that managed the MashQL's most important queries (N-level objects and N-Level properties Background Queries ,see chapter 4) use the technique of graph partitioning in order to reduce the size of the original RDF graph. As a result, the fetching beforehand algorithm that is applied is executed faster. The fetching beforehand results are partitioned in a database and will be fetched very fast when they are asked for. This solution has very promising results for MashQL's queries but till now , it can not be applied as a general RDF's query solution as the proposed techniques above can.

2.2 An Overview of Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a language for representing information about resources in the World Wide Web. For example, RDF is used to describe information about web pages such as (content, author, created and modified date etc.). In general, RDF is able to represent information about things that can be identified on the Web. RDF is based on the idea of identifying things using Web identifiers (called Uniform Resource Identifiers, or URIs), and describing resources in terms of **triples <subject, predicate, object>.** The subject is the part that identifies the thing .The property is the part that identifies the characteristics of the subject, and the object is the part that identifies the value of that property.

An RDF triple has the following characteristics

1. the subject, must be a URI reference or a blank node
2. the property , must be an RDF URI reference
3. the object, must be an RDF URI reference, a literal or a blank node

For example , the RDF statement **"The web page http://www.example.org/index.html has a creation date August 16, 1999"** can be expressed in the following triple and the corresponding RDF graph is demonstrated in Figure 2.1.

Subject: http://www.example.org/index.html

Property: creation date

Object: August 16, 1999

The combinations of RDF statements enable RDF to represent the web resources **as directed and labeled graph of nodes** (It is also called "The RDF model"). The graph nodes are the subjects and objects and the arcs are the predicates.

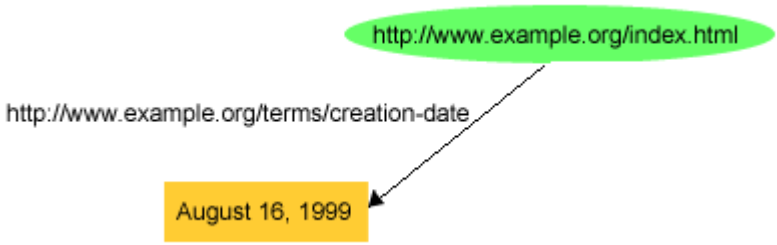


Figure 2.1: A corresponding graph for the RDF statement [10]

As we mentioned above, the RDF concept is a design to be read and understood by computer terminals in order to exchange RDF data between them. RDF itself has been serialized in a number of formats including N3, RDFa, Turtle, and N-triples and they are the easiest way of storing and transmitting RDF data. These formats provide an easier way to scribble (human-readability in mind) an RDF graph rather than RDF/XML.Notation3, or N3 as it is more commonly known, is a shorthand non-XML serialization of Resource Description Framework models[10,11].

2.3 An overview of SPARQL (SPARQL Protocol and RDF Query Language)

SPARQL (SPARQL Protocol and RDF Query Language) provides a protocol and the query language for RDF. SPARQL can be used to express queries across different data sources, whether the data is stored natively as RDF or viewed as RDF via middleware. By using SPARQL you are able to query (reading information and not writing, updating etc.) RDF graphs. Those are saved in persistent storage such as database or file. SPARQL Protocol is a means of transmitting SPARQL queries from query clients to query processors (query client is the side that provides RDF queries against a dataset and the query processor side that RDF dataset is a host). SPARQL protocol has been designed for compatibility with the SPARQL Query Language for RDF (SPARQL). The SPARQL Protocol consists of an abstract interface called *SparqlQuery* which is independent from other protocols. *SparqlQuery* contains an operation called query, which is used to convey the query string and optionally an RDF dataset description. The query string is an instance of XML schema and it must stand on SPARQL syntax. *SparqlQuery* requires protocol binding to become operational. SPARQL Protocol supports HTTP and SOAP bindings. This indicates that you can post a SPARQL

queries and you can get a result from a URI dataset[s] using the HTTP protocol or you can include your SPARQL query into a SOAP message in order to send a query and get a result back (Web Services approach)[12].

2.4 Oracle Database RDF Technologies

2.4.1 Introduction to Oracle RDF Technologies

Oracle's RDF web technologies constitute the industry's first open, scalable, secure and reliable data management platform for RDF and ontologies which enable you to store and retrieve RDF data sources (Ontologies are out of the scope of this document).

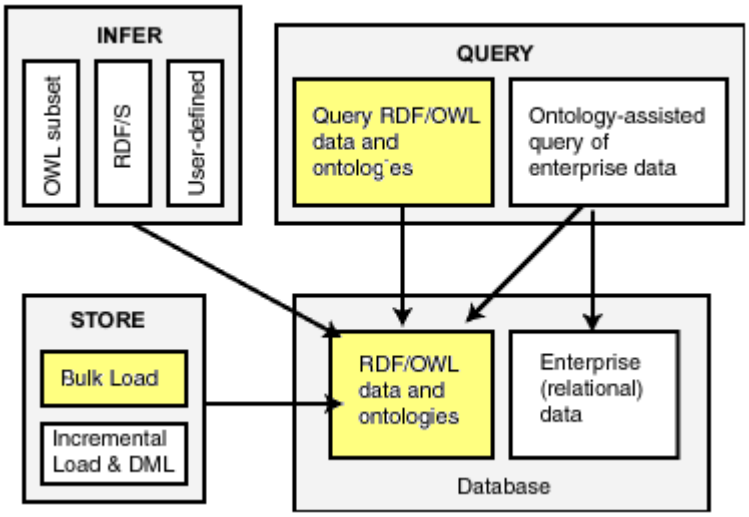


Figure 2.2: Oracle's RDF capabilities [13]

As shown in Figure 2.2, the database contains RDF data as well as traditional relational data. To load RDF data, the bulk loading is the most efficient approach; although you can load data incrementally using transactional INSERT statements. After loading ,you can query these data simultaneously with the traditional relational data[13].

2.4.2 Oracle RDF Data Modeling

RDF data is structured as **directed graphs.** An RDF graph is a set of RDF **triples**. Like a number or string data types, a triple is treated in oracle database as a data type , named **SDO_RDF_TRIPLE_S**. In the Oracle Database RDF Technologies , each RDF graph is

called a **model .** The set of **nodes** of an RDF graph is the set of subjects and objects of triples in the graph. These nodes are used to represent two parts of the triple(the subjects and the objects), and the third part (the properties) is represented by a directed link that describes the relationship between the nodes.

The triples are stored in an **RDF data network**. The RDF Data Network is a **logical universe** that stores all the RDF graphs that exist in the database (There is one universe per Oracle database)[13].

2.4.3 RDF Data in the Database

All RDF graphs' data and metadata are parsed and stored in the system as entries in tables under the **MDSYS** schema (mdsys is an oracle's system user). A user-created model is formed by specifying a **model name**, the **relational table name** and a table's column of type SDO_RDF_TRIPLE_S which contains the RDF data. With this structure , oracle stores all the related relation as data in the table and all the RDF data and metadata in the tables **RDF_MODEL_INTERNAL$, RDF_VALUE$,RDF_LINK$** and **SEMM_<model-name>** (Figure 2.3).

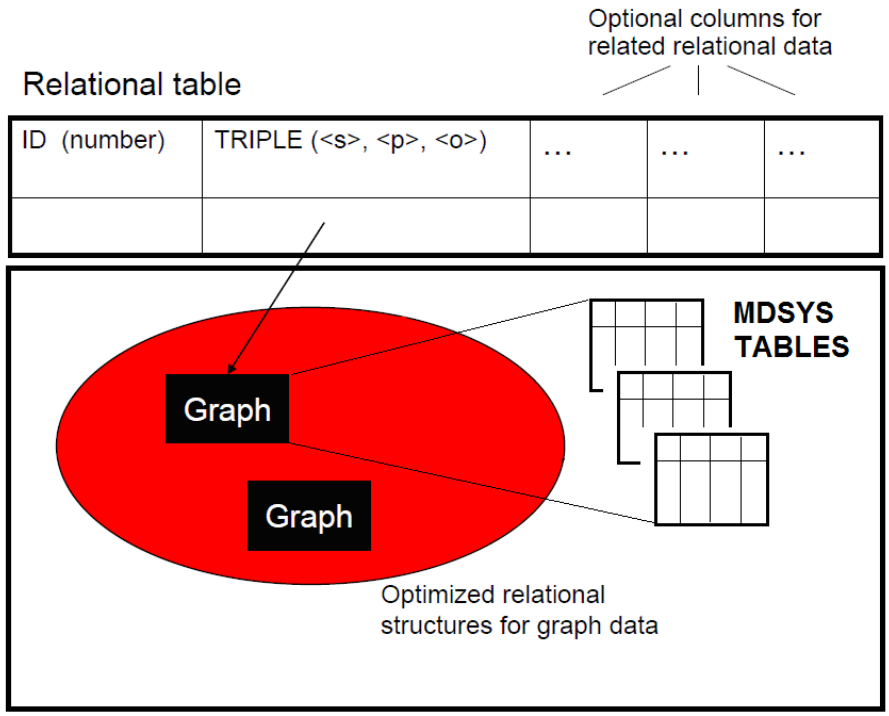


Figure 2.3: How Oracle Stores RDF Data in an RDBMS

The table **RDF_MODEL_INTERNAL$** (Figure 2.3) contains information about all models defined in the database. After the creation of the model, a view **SEMM_<model-name>** (Figure 2.4) is automatically formed and contains all the triples associated with the model. The values for subject, property and object are computed from its corresponding lexical values as high numbers.

COLUMN_NAME	DATA_TYPE
OWNER	VARCHAR2(30)
MODEL_ID	NUMBER
MODEL_NAME	VARCHAR2(25)
TABLE_NAME	VARCHAR2(30)
COLUMN_NAME	VARCHAR2(30)
MODEL_TABLESPACE_NAME	VARCHAR2(30)

Figure 2.4: RDF_MODEL_INTERNAL$'s columns description (The above screenshot comes from Oracle's SQL-Developer which is a graphical tool for database development [19])

COLUMN_NAME	DATA_TYPE
P_VALUE_ID	NUMBER
START_NODE_ID	NUMBER
CANON_END_NODE_ID	NUMBER
END_NODE_ID	NUMBER
MODEL_ID	NUMBER
COST	NUMBER
CTXT1	NUMBER
CTXT2	VARCHAR2(4000)
DISTANCE	NUMBER
EXPLAIN	VARCHAR2(4000)
PATH	VARCHAR2(4000)
LINK_ID	VARCHAR2(71)

Figure 2.5: SEMM_<model-name 's columns description (The above screenshot comes from Oracle's SQL-Developer which is a graphical tool for database development [19])

The table **RDF_VALUE$** (Figure 2.5) contains information about the subjects, properties, and objects used to represent RDF statements. It uniquely stores the text values (URIs or literals under the *value_name* column) for these three pieces of information, using a separate row for each part of each triple[13].

COLUMN_NAME	DATA_TYPE
VALUE_ID	NUMBER
VALUE_TYPE	VARCHAR2(10 BYTE)
VNAME_PREFIX	VARCHAR2(4000 BYTE)
VNAME_SUFFIX	VARCHAR2(512 BYTE)
LITERAL_TYPE	VARCHAR2(1000 BYTE)
LANGUAGE_TYPE	VARCHAR2(80 BYTE)
CANON_ID	NUMBER
COLLISION_EXT	VARCHAR2(64 BYTE)
CANON_COLLISION_EXT	VARCHAR2(64 BYTE)
LONG_VALUE	CLOB
VALUE NAME	VARCHAR2(4000 BYTE)

Figure 2.6: RDF_VALUE$'s columns description(The above screenshot comes from Oracle's SQL-Developer which is a graphical tool for database development [19])

The table **RDF_LINKS$** table (Figure 2.6) stores the graph properties(*p_value_id*) and describes the relationship between the subjects (*start_node_id*) and objects(*end_node_id*). In other words , the RDF_LINKS$ expose the directed graph links.

COLUMN_NAME	DATA_TYPE
P_VALUE_ID	NUMBER
START_NODE_ID	NUMBER
CANON_END_NODE_ID	NUMBER
END_NODE_ID	NUMBER
MODEL_ID	NUMBER
COST	NUMBER
CTXT1	NUMBER
CTXT2	VARCHAR2(4000 BYTE)
DISTANCE	NUMBER
EXPLAIN	VARCHAR2(4000 BYTE)
PATH	VARCHAR2(4000 BYTE)
LINK_ID	VARCHAR2(71 BYTE)

Figure 2.7: RDF_LINKS$'s columns description(The above screenshot comes from Oracle's SQL-Developer which is a graphical tool for database development [19])

When a triple is inserted into an RDF model, the subject, property, and object are first checked against the RDF_VALUE$ table, to see if entries for their text values already exist in the model. If they already exist (due to previous statements in other models) no new entries are made; if they do not exist, three new records are inserted into the RDF_VALUE$ table. If the subject, property, and object text values already exist in the RDF_VALUE$ table, another

check is issued to determine if the actual triple exists. This second check is issued against the RDF_LINK$ table. If the triple for the particular model already exists, no new triple is inserted. Otherwise, a unique ID is generated for the new triple. This ID is stored as the LINK_ID (also known as the RDF_T_ID). The VALUE_ID in the RDF_VALUE$ table corresponding to the subject becomes the START_NODE_ID; and the VALUE_ID corresponding to the object becomes the END_NODE_ID for this link. The VALUE_ID is the same as the VALUE_ID in the RDF_VALUE$ table. The MODEL_ID column logically partitions the RDF_LINK$ table. Selecting all the links for a specific MODEL_ID, returns the RDF network for that specified model [13].

2.4.4 Query RDF Data

You can not query RDF data directly from the relation table using SQL-BASED queries. In order to query RDF data you should use oracle's functions . For instance , the GET_TRIPLE() function returns the SDO_RDF_TRIPLE string. The functions GET_SUBJECT(), GET_PROPERTY() and GET_OBJECT() return the subject, predicate, and object, respectively. In order to use the SPARQL language you need to use the SEM_MATCH table function. An example of a SEM_MATCH table function execution is demonstrated in table 2.1 [13].

```
SELECT x, y
  FROM TABLE(
     SEM_MATCH
     (
         '(?x :grandParentOf ?y) (?x rdf:type :Male)',
          SEM_Models('family'),
          NULL,
          SEM_ALIASES(SEM_ALIAS('','http://www.example.org/family/')),
         NULL
    )
);
```

Table 2.1: Oracle’s SEM_MATCH Example

The Query in table 2.1 returns all grandfathers and their grandchildren from the oracle's family model as defined in SEM_Models('family'), [13].

2.4.4.1 SEM_MATCH attributes

Based on [13] , Oracle's SEM_MATCH table function has the following attributes :

Query: The query attribute is a string literal with one or more triple patterns and correspond to SPARQL syntax for example :

'(?x :grandParentOf ?y) (?x rdf:type :Male) (?y :height ?h)'

Models: The models attribute correspond to the list of graph[s] or models that you need to query.

Rulebases: Rulebases is out of the scope of MashQL , thus is always NULL .

Aliases: The aliases attribute identifies one or more namespaces included in the query. It is the @prefix synonym that is used in SPARQL. Oracle has the following aliases by default.

rdf:http://www.w3.org/1999/02/22-rdf-syntax-ns#

rdfs:http://www.w3.org/2000/01/rdf-schema#

xsd:http://www.w3.org/2001/XMLSchema#

Filter: The filter attribute identifies any additional selection criteria against the variable that is used in the query .For example: '(h >= 6)' to limit the result to cases where the height of the grandfather's grandchild is 6 or greater (using the oracle's family example).

Index_status: Index_status is out of the scope of MashQL , thus is always NULL.

2.4.5 Loading RDF data in Database

There are three ways that can you load RDF data into a database:

1.Bulk load using a SQL*Loader : a direct-path load which gets data from an N-Triple format and load it into a predefined staging table. After that a PL/SQL procedure is used to load or append the data into the database. The bulk load insert using the SQL*Loader is considered the fastest way to load any data into a database.

2.SQL INSERT statements using the SDO_RDF_TRIPLE_S constructor.

3.Batch load using a Java client interface to load or append data from an N-Triple format file into the database [13].

2.5 An overview of MashQL Language

World Wide Web is witnessing an increase in the amount of structured data on the Web. On the other hand, traditional search engines fail to serve such data since their core design is based on keyword-search over unstructured data, thus, any search against these data will not be precise or clean. This trend of structured data is shifting the focus of web technologies towards new paradigms of structured-data retrieval. In order to help people consume more structured data on the Web, the World Wide Web's giants such as Google, Upcoming, Flicker, eBay, Amazon, Yahoo and others have started to make their structured content freely accessible through Application Program Interfaces (API) , really Simple Syndication feeds (RSS),Web Services (WS) and RDF. Examples of World Wide Web's programs that have started to consume the above applications are the Mashups. Mashups are used to collect, combine or syndicate data or functionality from more than one web sources in order to create a new service. Unfortunately, building mashups is an art that is limited to skilled programmers and it cannot be considered as a general solution for structured-data retrieval, regardless the effort put in by various mashups' editors to simplify this art (They were provided by some mashups' editors. Those results have limited possibilities).

MashQL language is building on the success of Web 2.0 mashups and overcome their limitations. MashQL language targets are to be a general solution for structured-data retrieval in order to allow people to mash up and merge RDF data sources very easily. It is regarding the web as a database, where each data source is seen as a table and a mashup is seen as a query over one or multiple web sources expressed in any RDF format. In other words, MashQL helps people, instead of developing a mashup as an application that gets access structured to data through APIs, the MashQL simplified this art by regarding a mashup as a query on the candidate data sources that people need to get information. For example, if some user needs information from a web resource, the MashQL language will help the user to formulate his/her query from these data sources very easily. Firstly, the MashQL language downloads the data sources into a local database , and using the MashQL's editor , the

user ,with out any IT-skills is able to formulate his/her queries , and get back very fast to his/her results. As a result, the main novelty of MashQL is to allow non IT-skilled people to query and explore one or more RDF sources without any prior knowledge about the schema, structure, vocabulary, or any technical details of these sources. Since MashQL is waiting for the data sources to be in RDF format, in the background, all MashQL queries are translated into and executed as SPARQL queries (the corresponding language that is able to query RDF data).

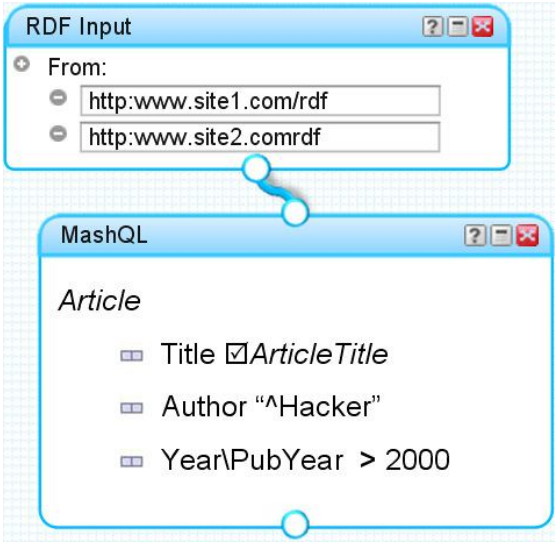


Figure 2.8 : An example of MashQL's query[4]

The MashQL example that is demonstrated in figure 2.8 shows a simple MashQL query using a MashQL's query editor. This query retrieves the recent articles from Cyprus, i.e. the title of every article that is written by an author, who has an address. This address has a country called Cyprus, and the article is published after 2000. The first module specifies the query input, while the second MashQL module specifies the query body. In the query input you can specify a number of valid URIs in any RDF's compatible formats such as RDF/XML, Native Triple (NT) and Notation 3 (N3). Those data represent an online RDF data sources. The query body is designed dynamically and it allows the MashQL users to explore an RDF graph very easily. All MashQL queries are seen as a tree. The root of this tree is called the query subject (e.g. Article), which is the subject matter being inquired. Each branch of the tree is called a query restriction and is used to restrict a certain property of the query subject. Branches can be

expanded to allow the formation of sub trees (called query paths), which enable one to navigate the underlying data sources. When the query is formulated, it is translated in SPARQL and executed on the RDF engine's backend database (MashQL uses the Oracle's 11g Semantic Technology as RDF engine backend database [1]).

The MashQL server consists of three important modules:

1. The MashQL editor which provides the graphical user interface of the MashQL . MashQL's users use the editor's drop-down lists in order to express their queries. These drop-down lists are dynamically generated during the program execution course and their results depend on the users inputs/selections [1].
2. The RDF Loader which downloads and loads RDF data from data source[s] that are defined in MashQL's query input into an Oracle's RDF model [1, 2, 3, 4, 5].
3. The query module which consists of a Query Language and a Query Formulation Algorithm. The former is a query language which supports all constructs of SPARQL and the latter is used by the MashQL editor for query formulation. MashQL's Query Formulation Algorithm formalizes a background query in each interaction in such a way that users can navigate and query a data graph, without prior knowledge about it. To achieve that, MashQL's Query Formulation Algorithm defines seventeen types of queries. Those queries are called background queries [1, 2, 3, 4, 5]. All the background queries are used in order to help users to create the final query or the formulated query that will be executed on the RDF engine database and it contains MashQL's users final result.

2.5.1 MashQL's performance considerations

In this section we describe our analysis according to the factors that may influence the performance of MashQL server focus on MashQL's components i.e. the RDF loader and the Queries performance.

2.5.1.1 MashQL's RDF loading considerations

When users select a remote source, this source must be transferred and stored locally in an RDF engine backend database before executing the query [4]. The users are able to query a data source only when at least one of the sources that are defined in the query input is loaded in the database or at least one of the data sources that are defined in the query input is already loaded in the system and has fresh contents, otherwise the users are not able to use the system. This issue is very critical for MashQL operation , since it influences a lot the availability of the system. The system must be in position to fast load the data sources in order to have the data as fast as possible available to its users. As a result, the loading time of the data sources must be short or humanly acceptable. If we bypass any network related issues such as a server's connection bandwidth, data source geographical location etc., the remaining factors that influence the loading process are the data source size and the efficiency of the RDF engine database to load RDF data. For example , the bigger the size of the RDF file, the more time is needed to load the file (imaging loading million or billion triples). Also , the RDF engine database will delay to load the RDF data since it makes too many transformations against the RDF data.

Additionally, the web resources are modified very frequently , thus the data sources that are already loaded in the system must be refreshed when their contents are staled. As a result the system needs to download the data source again in order to provide fresh data to its users.

In addition, in the majority of internet applications (multi-user environment) many users will be requested to work on the same resources at the same time. For example, in the MashQL,

many users need to download the same data source at the same time. As a result, the loader must be able to manage all these requests efficiently.

Finally, the Oracle 11g database which is proposed for MashQL to be the RDF engine backend database is able to load only Native Triple format (NT) files. Thus all the data sources that are not in the NT format must be converted to NT. This conversion step produces an extra latency and further administration overhead seeing as the file must be downloaded and converted to NT format in order to be loaded in the database.

2.5.1.2 MashQL's queries performance considerations

As we mentioned above, MashQL defines two types of queries, the background queries that are used in the formulation algorithm and the final query or the formulated query. The Background queries are executed very frequently, since they provide their results in the MashQL editor's drop-down lists that are used during the formulation algorithm.As you understand, the background queries play critical role in the system, since their performance influence the system's operation and availability. As a result, all background queries should be executed very fast (few seconds each query) in order to generate as faste as possible the MashQL's diagrams, the core MashQL's function. All MashQL's queries are translated into the SPARQL language and this SPARQL code is executed on the RDF engine backend database. As shown in [4], the performance of MashQL queries is limited to the performance of the used backend database as well as the performance of the queries which is reduced when the data source is very large[25]. In order to optimize their output performance , all background queries must be analyzed in order to find out (a)their execution characteristics, i.e. the number of arguments in the selected list, how many predicates are used per query, (b) their execution requirements i.e. CPU time , Memory consumption, Disk consumption , Sorting consumption, Input/Output consumption, (c) their execution weaknesses i.e. some queries may produce a high I/O consumption and (d) their execution trends i.e. intensive table self-joins problems. As a results, we need to focus on solutions that are specific for the MashQL's queries in order to run these queries faster , regardless the RDF engine database and data source size.

All the MashQL's queries are organized and prepared for execution according to the MashQL formulation algorithm. The formulation algorithm defines four basic steps [1]:

```
Step 0: Specify the dataset G in the Input module
Step 1: Select the query subject S
Repeat Step 2-3 (until the user stops)
Step 2: Select a property P.
Step 3: Add an object filter onto P.
```

At the beginning of the query formulation process (Step 1), the MashQL's users can select a subject from a MashQL editor's drop down list that contains, either:

1. A set of rdf:type objects, types (O) belong to graph G (the rdf:type property state that a resource is an instance of a class, i.e. the class can be an Article, a Person etc) (Table 2.2, query 1).
2. All the unique subjects (S) and objects (O) filtered by objects that are URIs (Table 2.2, query 2).
3. All the subjects (S) that match with an input variable provided by the user (Table 2.2, query 3).

The Background Queries [1...3] transform to the following Oracle's SPARQL code.

```
S=Subject
P=Property
O=Object
V=A value comes from the MashQL's editor drop-down list
F=A variable introduced by users

BQ-1

SELECT o
FROM TABLE(SEM_MATCH(
'(?s rdf:type ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o
order by o

BQ-2

SELECT s
FROM TABLE(SEM_MATCH(
```

```
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
UNION
SELECT o
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
where o$rdfvtyp='URI'  --filter isURI

BQ-3

SELECT s
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
where s like '%<F>%'
```

Table 2.2: Background Queries 1-3

The next step is to select a property (P) (Step 2) for the chosen subject above. The MashQL editor's drop down list will return a list of the possible properties for this subject. There are four possibilities:

1. Users can choose a subject (S) that belongs to an rdf:type class (Table 2.3, query 4).
2. Users can choose a subject (S) that comes from the MashQL editor's drop down list (Table 2.3, query 5).
3. Users can choose a subject (S) to be a variable, by introducing their own value (Table 2.3, query 6).
4. Users can also choose the property to be a variable by introducing their own value (Table 2.3, query 7).

For each case, the MashQL editor's drop down list will display all the properties (P) that are associated with the input subject (S).

The Background Queries [4...7] transform to the following Oracle SPARQL code:

```
S=Subject
P=Property
O=Object
V= A value comes from the MashQL's editor drop-down list
F=A variable introduced by users

BQ-4

SELECT p
FROM TABLE(SEM_MATCH(
'(?s rdf:type < V>)
 (?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by p
order by p

BQ-5

SELECT p
FROM TABLE(SEM_MATCH(
'(<V> ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by p
order by p

BQ-6

SELECT p
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
where s like '%<F>%'
group by p
order by p


BQ-7

SELECT p
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
```

```
null,
null,
null))
where p like '%<F>%'
group by p
order by p
```

Table 2.3: Background Queries 4-7

Finally, (Step 3), MashQL's users are able to add an object filter on property (P). There are three types of filters that users can use to restrict property (P):

1. A filtering function
    a. A filtering function can be selected from a list (e.g., Equals, More Than, one of, not) (Table 2.4, queries 8..9).
2. An object identifier
    a. If users want to add an object identifier as a filter, a list of the possible objects will be generated (Table 2.4, queries 10..13).
3. A query path for n-level properties and n-level objects (Table 2.4, queries 13..17).

The Background Queries [8...13] transform to the following Oracle SPARQL code:

```
S=Subject
P=Property
O=Object
V= A value comes from the MashQL's editor drop-down list
F=A variable introduced by users

BQ-8

SELECT o
FROM TABLE(SEM_MATCH(
'(<V> ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
where o$rdfvtyp='URI'

BQ-9

SELECT o
FROM TABLE(SEM_MATCH(
'(<V> ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
```

```
null,
null,
null))
where o$rdfvtyp='URI'
and p like '%<F>%'
```

BQ-10

```
SELECT o1
FROM TABLE(SEM_MATCH(
'(?s rdf:type <V>)
 (?s1 ?p1 ?o1)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o1
order by o1
```

BQ-11

```
SELECT o1
FROM TABLE(SEM_MATCH(
'(?s rdf:type <V1>)
 (?s1 <V2>)?o1)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o1
order by o1
```

BQ-12

```
SELECT o
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o
order by o
```

BQ-13

```
SELECT o
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
where p like '%<F>%'
group by o
order by o
```

Table 2.4: Background Queries 8-13

During the formulation algorithm, MashQL's users need to expand some properties (P) or some objects (O) in order to declare its path. For example, from the Books of the RDF graph in figure 2.9, the property "Author" has the following valid path *"Author" =>"Affiliation"=> "Country" =>"Name"*.

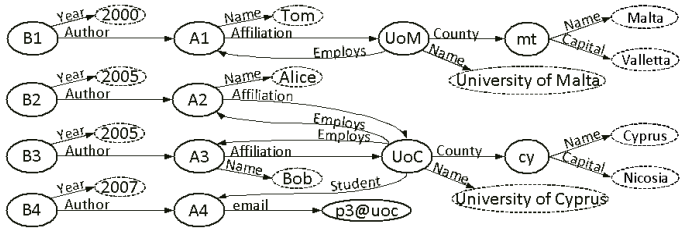


Figure 2.9: Books RDF graph [1]

These types of background queries are called N-level objects and N-Level properties background queries since these queries navigate in the RDF graph having as a root one concrete object or property. The variable N determines the query's expansion level inside the RDF graph or the depth path that the query needs to be discovered or to be expanded. For the path that we have already declared in the graph that is presented in figure 4.2, the root property is the *"Author"* and the level of the query (the value of N) is equal to 3. The corresponding background queries of an N-level properties and N-level objects queries with N level is 3. They are presented in queries 14 to 17 .

```
S=Subject
P=Property
O=Object
V= A value comes from the MashQL's editor drop-down list
F=A variable introduced by users

BQ-14

SELECT p2
FROM TABLE(SEM_MATCH(
'(?s rdf:type ?o)
 (?o ?p1 ?o1)
 (?o1 ?p2 ?o2)
',
SEM_Models('<GRAPH_NAME>'),
null,
```

```
null,
null))
group by p2
order by p2

BQ-15

SELECT o2
FROM TABLE(SEM_MATCH(
'(?s rdf:type ?o)
 (?o ?p1 ?o1)
 (?o1 ?p2 ?o2)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o2
order by o2


BQ-16

SELECT p2
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
 (?o ?p1 ?o1)
 (?o1 ?p2 ?o2)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by p2
order by p2

BQ-17

SELECT o2
FROM TABLE(SEM_MATCH(
'(?s ?p ?o)
 (?o ?p1 ?o1)
 (?o1 ?p2 ?o2)
',
SEM_Models('<GRAPH_NAME>'),
null,
null,
null))
group by o2
order by o2
```

Table 2.5: The general case of an n-level properties and n-level objects background queries

The analysis of the queries that are defined by MashQL formulation algorithm shows that the MashQL background queries can be divided into two categories , the general background queries (queries 1.. 13) and the n-level properties and n-level objects background queries

(queries 14..17) .This separation derives from the way that these categories of queries access the data from the database.In the first category, i.e. the general background queries (queries 1.. 13) , they specifically do not present many problems in the intermediate data sets. Concerning larger data sets, these queries behave more slowly, therefore, they need improvement in order to respond faster. They have always one argument in their select list, either a triple subject , or the object or a the property . Their results are restricted via a filter that exists inside the queries' SPARQL code. These filters correspond to the values that are selected by users from the MashQL's editor drop-down list or the values that are introduced by a users. Finally, these queries return their results sorted and in alphabetical order based on the argument that is selected in the select list. In addition, these types of queries do not need to be discovered or expanded further on the corresponding RDF graph and do not have the problems that appear in the second category. As a result, the latency that appears in this category of the queries may appear due to the fact that the database lacks indexing and the high sorting activity should it occur from the database side in order to return the data sorted and in alphabetical order. Also, the majority of these queries experience follows due to the fact that their result set is sometimes very large. As a result, the database calculates the result very fast but it needs a long time to display it (e.g. it may return 1 million rows).

The second category of queries, i.e. the n-level properties and n-level objects background queries, present very high interest , since for huge data sets these queries sometimes do not respond or they may have a very poor performance standard. The main reason for the bad performance devives from the queries' tendency to make so much table self-join as the number of levels that the queries need to expand. The problem becomes bigger for a very large number of RDF graphs with a million or billion triples.

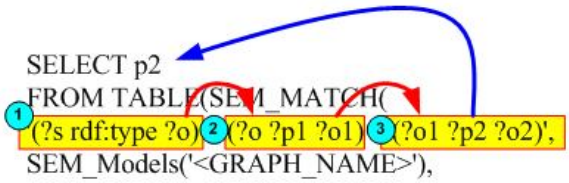


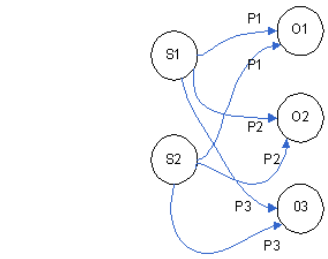
Figure 2.10: Self-Join Example

For example the background query 14 (Figure 2.5) is an example demonstrating this behavior pattern. The query defines three sparql patterns. Even though the query is executed as one unit, internally each pattern needs to select the graph data separately. Firstly, pattern one is executed, which produces its results in pattern two (variable ?o), secondly, pattern two is executed, which uses the previous results as input and produces its results in pattern three (variable ?o1) and finally, pattern three is executed using the previous results as input in order to produce the final result that are belongs in the variable ?p2. Internally, the RDBMS manages this situation by joining the graph data as match to the number of sparql patterns that is defined by the query (In database view this operation is called self-join operation, since the source table is joined by itself).The problem increases exponentially for each new sparql pattern that is added onto the query, especially when we have very large graphs with million and billion triples

2.5.1.3 Representation of RDF data in the oracle database (performance considerations)

The Oracle's RDF model stores the RDF graph in various tables, and it makes the data accessible only by a SEM_MATCH table function (for more details see the previous chapter Related technologies section Oracle Database Semantic Technologies). As you understand, the technology is closed to any customary optimizations. Fortunately, in front of this technology, the Oracle's RDF model provides you with a table for each RDF graph that you loaded into the database which can use only SQL-BASED queries [6] for any customary optimizations. This table consists of three columns, the subject, the property and the object (the table is also called triple table). The table's rows contain the RDF's triples. Figure 2.11 (a) shows a simple RDF graph with two subjects (S1, S2) and three objects (O1, O2, and O3). The subjects and objects are connected together by three types of properties P1, P2 and P3. A table representation of this graph is presented in figure 2.11 (b). As you can see in the table, the subjects S1, S2 and the objects O1, O2, and O3 are repeatedly displayed in the table in order to portray all of the graph triples. For very large graphs the above representation of data created a very huge table that influences negatively any SQL-BASED queries performance since ,the RDBMS must read more data in order to produce the result. Any sorting operations

are very memory-intensive, they are corruptible for self-join operations and they produce big I/O between the database memory buffers and the disk.



(a)

Subject	Property	Object
S1	P1	O1
S1	P2	O2
S1	P3	O3
S2	P1	O1
S2	P2	O2
S2	P3	O3

(b)

Figure 2.11: A simple RDF graph and its database table

Chapter 3

MashQL Server Design

This chapter describes the system architecture of MashQL Server which consists of the RDF Loader and the Query Optimizer. Section 3.1 describes a profound view of how the MashQL editor, the RDF Loader and the Query Optimizer work together in order to maintain a robust RDF retrieval engine. Section 3.2 shows how RDF Loader downloads and loads RDF data into the system. Section 3.3 explains how the Query Optimizer executes the MashQL's queries efficiently. Both sections are supported with design details, components specifications and examples.

3.1 System Architecture

MashQL consists of the MashQL Editor and the MashQL Server. The MashQL Editor is a module which is responsible for the formulation of a query through a formulation algorithm and it provides the MashQL Server with the formulated query for execution (in SPARQL format).

The MashQL Server consists of two important modules, the RDF Loader and the Query Optimizer. The RDF Loader downloads and loads RDF data from web to the system and the Query Optimizer executes in an optimum way all MashQL’s queries. Figure 3.1 represents the MashQL’s system architecture and describes how the MashQL’s components interact between them. The points 1 to 4 in figure 3.1 show the flow that the RDF Loader follows in order to load a data source into a system. The points 5 to 8 show the MashQL's queries execution cycle.

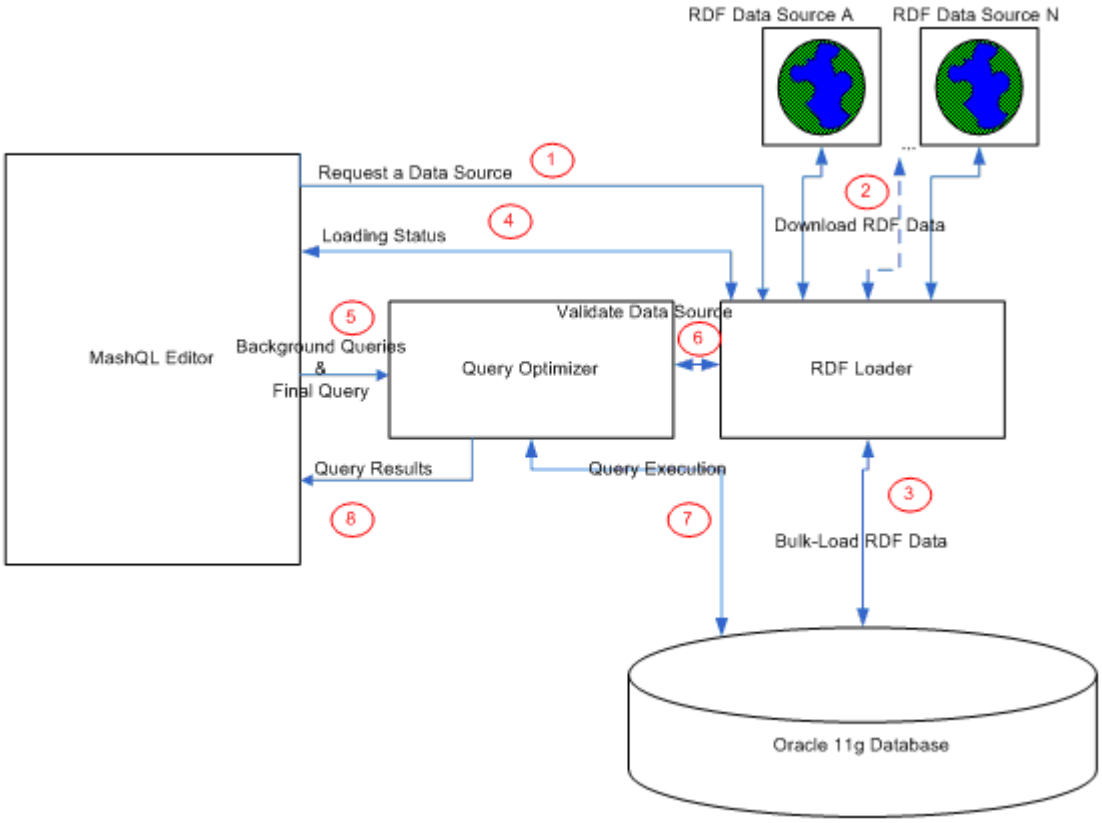


Figure 3.1: MashQL Server system architecture

To initiate a query with MashQL, the MashQL user is capable of choosing a number of RDF resources that are available in the web. These RDF resources can be RDF/XML, Native Triples (NT) and Notation 3 Triples (N3) and they are accessible via a unique Uniform Resource Locator (URL) as defined by W3C. As a result, the MashQL's user inputs in the MashQL editor the URLs strings of the candidate RDF resources that one needs to query (e.g. http://www.ucy.ac.cy/Source_A.rdf, http://www.ucy.ac.cy/Source_B.rdf).The query begins if, and only if, the RDF data is successfully loaded into the database (with the RDF technology enabled). Thus, the MashQL Editor invokes the RDF Loader for the database loading tasks by submitting to it all the requested URLs strings (figure 3.1, point 1). The RDF Loader prepares a number of validation steps before it downloads or loads the RDF data (figure 3.1, point 2). Firstly, it checks the validity of each URL and if the URL does not exist or it is unavailable, the RDF Loader notifies the MashQL editor about the status of URLs that failed.

For each valid URL, the RDF Loader checks if the URL is already cataloged in the system. To achieve that, the RDF Loader maintains a repository in the Oracle database which stores all

URL's metadata (URL string, file name, file type, last modified date, file size etc.). If the candidate URL is not cataloged in the system, the RDF Loader downloads the RDF data locally in the server, converts the RDF data from any RDF format to NT format and using the oracle's Bulk-Load feature it loads the RDF data on top of the Oracle's RDF model . If the Bulk-Load process is completed successfully the RDF Loader additionally creates all the appropriate database objects (tables, indexes, materialized views etc.) that will be used later in the Query Optimization module (optimization solution for MashQL's queries) and notifies the MashQL editor of the completion of the loading process (figure 3.1, point 3).

If the candidate URL data is already cataloged, because the same URL is used in the past from some other user[s] or by the same user, the RDF loader compares the metadata of the RDF data source that is already cataloged with the corresponding metadata of the RDF data source that is available in the web, and if it is stale, the RDF loader refreshes the data by downloading and loading the RDF data again. The old RDF data that is already cataloged in the system is a query accessible by MashQL's users until the end of the database loading of new RDF data. When the new RDF data is loaded in the system, the RDF loader notifies the MashQL editor of the completion of the loading process and it makes the new data available for new queries (figure 3.1, point 4). This technique provides MashQL with a transparent refreshed data loading by allowing MashQL's users the possibility of querying a cataloged URL's data during the refreshing process.

If the candidate URL's data is already cataloged but it is not staled, the RDF Loader notifies the MashQL editor of the completion of the loading process providing MashQL with a caching technique.

If a number of users seek the same URL, the RDF loader downloads and loads the data at once for the first user. At the end, the RDF Loader notifies all users of the completion of the loading process. As from the download phase till the final phase, the RDF Loader is in a position to notify the MashQL editor of any errors and faults that will abnormally happen in the system. The RDF loader does not resume any failed URL data loading for any reason. The MashQL's user has the opportunity to initiate the URL again.

By default, when the URL is used by the system, its data and metadata is retained in the system forever. However, the MashQL administrator has the advantage of marking a URL data as obsolete. All obsolete URL data and metadata are deleted by the system automatically.

As we already mentioned, MashQL's users are able to query RDF data sources if, and only if, the RDF data are loaded once the in system. Consequently, if the URL RDF data is already catalogued, regardless their state, MashQL's users can take advantage of the RDF Loader caching and transparently refreshing mechanisms. As a result, MashQL's users wait for URL data loading only for the fist time and the query process is easier.

The MashQL editor is able to query the candidate RDF data sources that are stored in oracle's RDF model, using two types of queries:

1. The background queries that are used during the formulation algorithm which are run in background
2. The formulated queries or final queries, the queries whose results contain the users' desired information.

During the query process the MashQL editor submits to the Query Optimizer a number of URLs that one needs to take information from, and the actual query in SPARQL format (figure 3.1, point 5). The Query Optimizer validates all input parameters and executes the query using all input URL's data (figure 3.1, point 6). All the non-valid URLs are ignored and are not included in the query. The Query Optimizer executes all background queries in its tables in order to provide the highest performance and all the formulated queries on oracle's RDF model (figure 3.1, point 7). The query's end result is returned to the MashQL editor immediately after the execution process (figure 3.1, point 8).

3.2 Components Specification

3.2.1 RDF Loader

The RDF Loader is MashQL's module that bulk-loads RDF data source[s] into an Oracle 11g database. When a user has specified an RDF data source[s] as input, the MashQL editor invokes the RDF Loader to download and load the source's data into the database. The MashQL editor communicates with RDF the Loader using the addURL and getURLStatus interfaces that are defined in Loader's API (Table 3.1). The addURL interface initiates the loading process by adding the loading request into a queue. The queue elements contain the data source URLs that are candidates for being loaded. After that, a number of de-queuer processes de-queue the messages from the queue and routes the messages for downloading locally in the system. When the RDF files are downloaded, they are converted in to a format that is able to be loaded in the database .During the loading process, the RDF Loader informs the MashQL editor of the status of each candidate data source via the getURLStatus interface that is implemented by the MashQL editor. The MashQL Administrators are able to monitor and tune the system's parameters via the web console that is available (Figure 3.2).

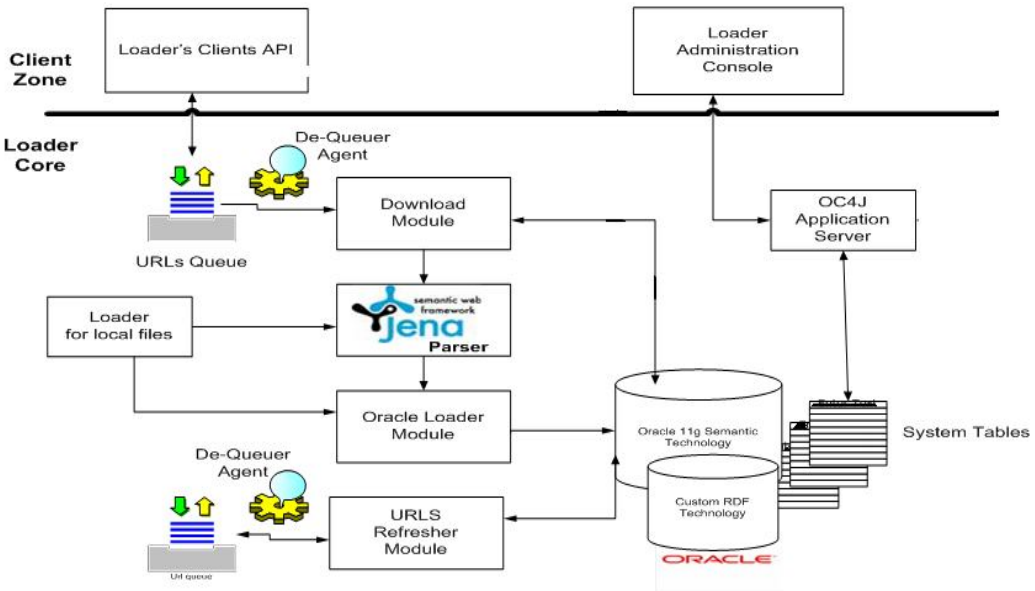


Figure 3.2: RDF Loader components

API Name	addURL
Description	It Loads an RDF data source into a database
Input Arguments	String which describes a valid URL. *http://<domain name>/file.[rdf|nt|n3]*
Output	Use the **getURLStatus PL/SQL function** in order to get the URL status from the system.
Type	Oracle 11g PL/SQL stored procedure
Database Connectivity	This procedure is created and stored in an Oracle 11g database, thus any programme languages that are able to execute a database PL/SQL procedures are supported. Use your programme language documentation for the database connectivity.

API Name	getURLStatus
Description	Returns the database loading status for a candidate RDF data source.
Input Arguments	String ,which describes a valid URL. **Argument format** *http://<domain name>/file.[rdf|nt|n3]*
Output	String, with the following format *Module's identifier #Message description* e.g. 400#URL: <URL name>successfully loaded.
Type	Oracle 11g PL/SQL function
Database Connectivity	This procedure is created and stored in an Oracle 11g database, thus any programme languages that are able to execute a database PL/SQL procedures are supported. Use your programme language documentation for the database connectivity.

Table 3.1: The definition of the addURL and getURLStatus RDF loader APIs.

The RDF Loader has the following components:

1. The RDF Loader API
2. URL Queue
3. Download module
4. The Jena Parser
5. Oracle SQL*Loader Module
6. Data source refresh module
7. Administration console

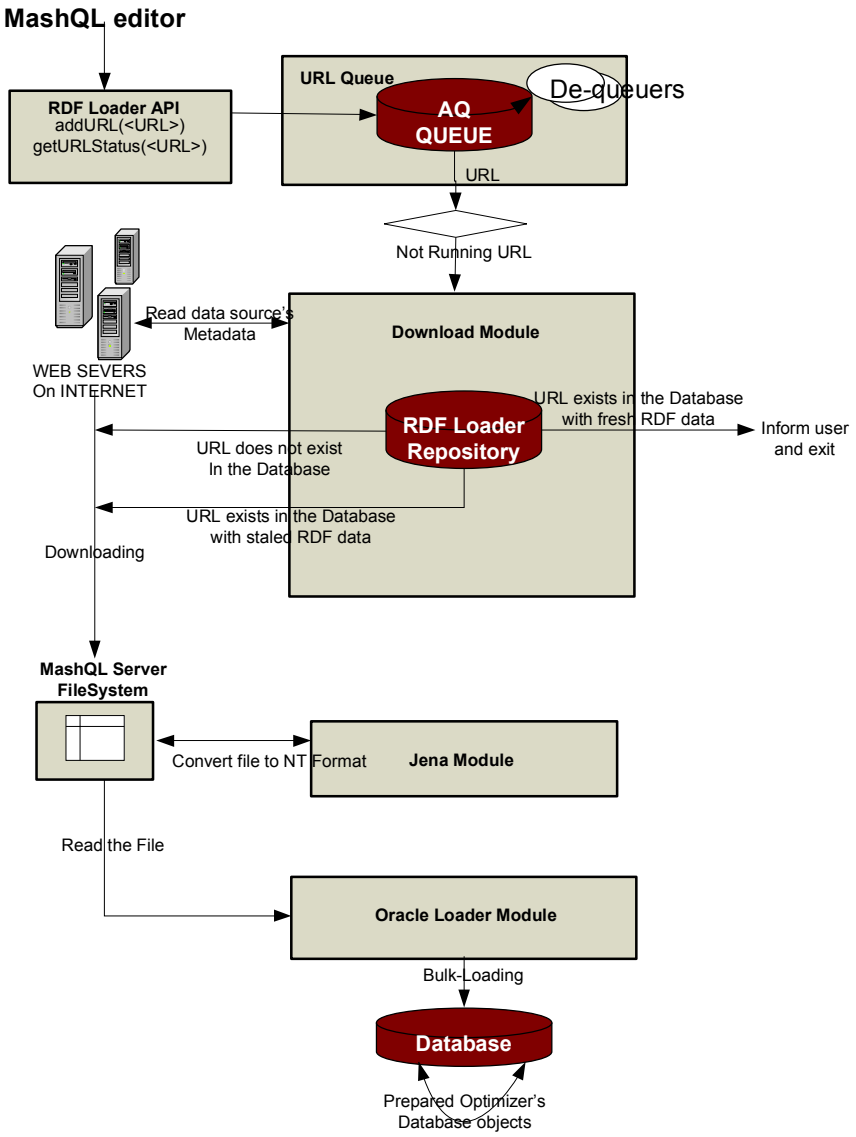


Figure 3.3: RDF Loader components design

3.2.1.1 The RDF Loader API

The RDF Loader API is an oracle's PL/SQL procedure and function. The API defines and describes all the interactions between the MashQL components and the RDF loader in order to successfully load a data source[s] into a database.

When the MashQL user needs to query a data source[s] from the web, the MashQL editor calls the addURL PL/SQL stored procedure. The addURL procedure is responsible for the en-queuing the candidate URL's string into the URL Queue. When the URL's string is en-queued into URL Queue, it waits its order for loading and further processing. The addURL

procedure is used for only one URL at a time; consequently, the MashQL editor will call the addURL separately for each data source until it satisfies all the candidate data sources.

During the loading process, the MashQL editor can get the status of a loading process via getURLStatus API. The getURLStatus PL/SQL function reads a centralized table from the RDF Loader repository and returns the appropriate message (Figure 3.3). The definitions of these APIs are listed on table 3.1.

3.2.1.2 URL Queue

The queuing functionality of the RDF Loader enables asynchronous communication between the RDF Loader and the MashQL editor. It offers guaranteed delivery of messages along with exception handling in case messages can not be delivered. URL Queue is stored on top of Oracle Advanced Queuing technology. The elements or messages of the URL Queue contain a requested URL string (figure 3.3).

3.2.1.2.1 De-queuer Agent

The URL Queue messages will be de-queued through the De-queuer Agent (figure 3.3). The de-queuer agent is a multithread java program which communicates with URL Queue in order to route the URL request. The de-queuer agent spawns sixteen (threads per CPU X CPU counts) daemons each of them running on different java thread and is connected with a database via an oracle's connection pooling mechanism. The system automatically pullulates more daemons when the throughout number is high (the initial number and the maximum number of daemons can be configured using the MashQL administration console). Each daemon de-queues a URL string from Queue. If the same URL is already de-queued from another session (the URL is running) the daemon stops the process for this URL and de-queues a new one.The MashQL's user who requests the URL will reuse the URL contents from the active session when it is completed.(figure 3.3).

3.2.1.3 Download module

The Download module simply downloads URL's RDF data from the web, locally in the server. The module accepts as input a URL string. When a new URL string is received, the module fetches the URL's metadata from the URL's location (Web). The URL's metadata are compared with URL's metadata that exists in the RDF loader repository and the comparison can produce three possible results:

1. The URL's data does not exist in the repository
2. The URL's data exists in the repository and has fresh data
3. The URL's data exists in the repository and has stale data

In case number 2, the system stops the process and notifies the system that the specific URL is available to include in MashQL queries. Otherwise, in cases number 1 and 3 the system downloads the file locally in the server. If the file's download is completed successfully the module notifies the system of the status of the download process and moves the request to Jena module for further processing (figure 3.3).

3.2.1.4 The Jena Parser

When the URL's data is downloaded, it is converted to N-Triple (NT) format. This can achieve usage of the Jena's parser (figure 3.3). The conversion is an inevitable task because NT format is a prerequisite for the Oracle SQL*Loader Module in the next step. Jena is a Java framework for building Semantic Web applications. It provides a programmatic environment for RDF and SPARQL. Jena is an open source and grown out of working with the HP Labs Semantic Web Programme. There are many tools in the market that do precisely the same work (figure 3.3) [15].

3.2.1.5 Oracle SQL*Loader Module

This module prepares the loads of URL's RDF data, on top of oracle's RDF model and, additionally, it prepares all the database objects that are needed by the Query Optimizer module.

As we already mentioned, the fastest way to load RDF data into the oracle database is the SQL*Loader tool. The SQL*Loader uses a direct-path loading method which gets data from a N-Triple (NT) format and it loads it into a predefined staging table called STAGING table. Subsequently, Oracle's SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE PL/SQL procedure is invoked in order to load or append the data from the staging table into the oracle's RDF technology tables. All RDF's data, irrespective of which data source they belong to, can be stored in one RDF graph (it is also called RDF model). In our system, each URL's RDF data is stored in a separate RDF model. We choose the latter technique because it is recommended by Oracle, it is more flexible, it provides faster access to RDF data, and it helps us in the transparent URL's refreshing and implementation.

During the transparent URL refreshing, the system maintains two versions of URL's RDF data. The old RDF data, is the active version and the new RDF data, is the inactive version. During the loading process of the new RDF data, MashQL's users are able to query only the active RDF data (old). The new RDF data become active when they are loaded successfully (figure 3.3) [13]

3.2.1.6 The data source refresh module

The data source refresh module is a daemon which periodically checks the freshness of the data sources which are already loaded into database. If the data source's data is stale, (the last modified date stored in the database is shorter than the current modified date of the data source). The system, automatically, updates the data source through own its source by adding the URL again onto the URL queue (figure 3.3).

3.2.1.7 Administration console

Any administration issues against the RDF LOADER can be completed using the administration console. The administration console is a web-based application written in Java. Through the console, MashQL's administrators can do the following:

1. Monitor all the system logs and actions

2. Add, delete, and update RDF data sources.
3. Tune system's parameters.
4. Start/stop/status and configure the de-queuer daemons.
5. Monitor URL Queue.

(figure 3.3).

3.3 Query Optimizer

The Query Optimizer is a module which predetermines the most efficient way of producing the result of MashQL's queries. During the query process, the MashQL editor submits to the Query Optimizer all the information that is needed to execute a MashQL's query.

The Query Optimizer validates all inputs and parameters and executes the query using the input of the URL's data. All non-valid URLs are ignored and are not included in the query. If the input query is a background query, the Query optimizer executes it on its database objects using the BQUERY table function. On the other hand, if the query is a formulated query, the Query Optimizer executes it on Oracle's technology using the SEM_MATCH table function (Figure 3.4).

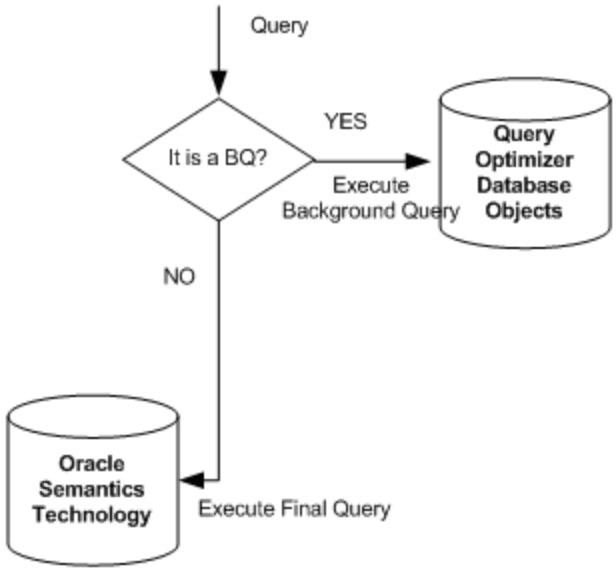


Figure 3.4: Query Optimizer Module Architecture

3.3.1 The executed_query interface

The communication between the Query Optimizer and the MashQL editor is achieved by using the executed_query API defined by the Query Optimizer (Table 3.2) .When a MashQL editor needs to execute a query, the executed_query API must be invoked. The API accepts as input a list of URLs, and the select list that contains the named columns that are included in

the query, the SPARQL query, queries conditions and filters (e.g. AND spy's name like '%007%'), RDF aliases which identifies one or more namespaces that are included in the query, and finally, the row to row limits which eliminate the rows that returned by the query and are used in the event that the query's result is too large. The executed_query API returns an Oracle's reference cursor. The reference cursor is a data type in the Oracle PL/SQL language and it represents a cursor or a result set in an Oracle Database.

API Name	**executed_query**
Description	Execute all MashQL Queries
Input Arguments	String which describes a URLs List separated by comma String which describes a query's Select List, String which describes the SPARQL Query , String which contains the query's conditions, String which describes the RDF Aliases List separated by comma, Integer From row, Integer To row
Output	Oracle's reference cursor
Type	Oracle 11g PL/SQL stored procedure
Database Connectivity	This procedure is created and stored in an Oracle 11g database, thus any programme languages that are able to execute a database PL/SQL procedure are supported. Use your language's programme documentation for the database connectivity.

Table 3.2: The definition of the executed_query interface

Table 3.2 contains an example of a background query execution using the executed_query procedure. The query returns all the articles ' properties from two RDF data sources .The query returns only the results from position 20 to position 70.

```
execute_query
(
  'http://cs.uoc.ac.cy/sem_src1.rdf, http://cs.uoc.ac.cy/sem_src2.rdf',
  'ArticlesProperties',
   (?all ?ArticlesProperties ?ArticlesObjects)',
  null,
  null,
  20,
  70
)
return reference cursor
```

Table 3.3: Background query example using executed_query procedure

3.3.2 Query Optimizer System Design

When the MashQL sends the query to the Query Optimizer (Figure 3.5), the Query Optimizer uses its own simple algorithm to calculate the hash value of the query. The hash value is an integer number which is calculated using the query's SPARQL, the selection's named columns and the Query's condition. Each hash value is associated with a background query template stored in the database.

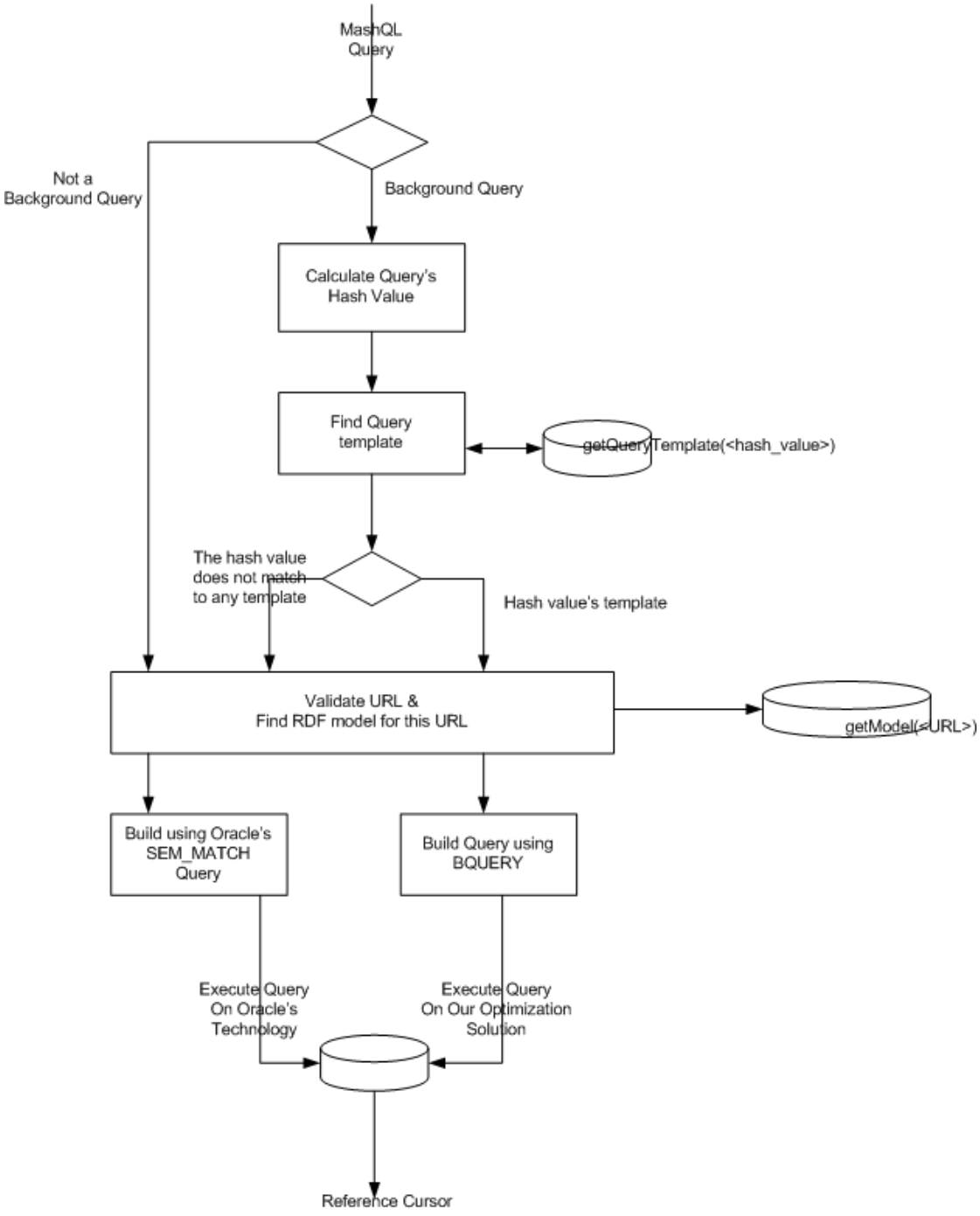


Figure 3.5: Query Optimizer components specification

The Query Optimizer uses the technique of the hash value in order to determine which background query to run. The query template contains a pre-defined tuning query, that is ready to run on a Query Optimizer's table. All formulated queries do not have any hash values associated with it, thus their queries are created dynamically and the query runs using Oracle's SEM_MATCH table function. If the query's selection named columns are more than one, the hash value is not calculated, and the query is executed on the Oracle's technology (all the MashQL background Queries have only one named column in the query's selection list). If the hash value does not correspond to any background query, a default query template is created in order to be executed on Oracle's technology.

As we already mentioned, in the section called “Oracle’s RDF technology”, the Oracle technology keeps all URL's RDF data into a central storage schema and saves it as graphs (or models). Each URL has its own graph associated with it. As a result, in order to get access to a specific URL's data inside the oracle database, you need to specify the model name or the graph name. Thus, the Query Optimizer checks if the selected URLs are loaded successfully into the database. If they are, for each successful URL, it gets the correlate Oracle's RDF model name , if not, it is ignored , and it is not included it in the query. The mapping between the URL and RDF model name is to keep track of the database. The model name is used in the query for both the Oracle’s SEM_MATCH query and the Query Optimizer’s BQUERY .The SEM_MATCH uses the model name to specify the access data through from the Oracle’s technology. On the other hand, the BQUERY uses the model in order to determine the optimizer’s tables and views that it needs to access in order to satisfy the query. The SEM_MATCH query uses the SEM_MODELS attribute in order to include more models in the query (Table 3.5).

The BQUERY cannot combine more than one URL simultaneously (like SEM_MATCH). Thus, for each model it creates identical queries and each of them get access to different database objects that are executed all together as one query using the UNION operator (Table 3.4).

```
select datum from bquery
(
        'MVN$P_UOC_EXAMPLE1',
        'p',
        '(?s ?p ?o)',
        null,
        20,
        70
 )
 UNION
select datum from bquery
(
        'MVN$P_UOC_EXAMPLE2',
        'p',
        '(?s ?p ?o)',
        null,
        20,
        70
);
```

Table 3.4: The query that is created for the corresponding example in table 3.3

```
SELECT ArticlesProperties
        FROM TABLE(SEM_MATCH
        ('(?all ?ArticlesProperties ?AuthorsObjects)',
        SEM_Models('model1','model2'),
        null, null,null)
        )
group by ArticlesProperties
order by ArticlesProperties;
```

Table 3.5: How to invoke the SEM_MATCH function in order to execute a background query concerning the example in table 3.3.

Table 3.6 shows the query that is created for the corresponding background query in the example in table 3.3. As you can see, the MashQL's background query is replaced with the corresponding query template and has exactly the same meaning as the original MashQL's query. The URLs are translated through materialized views MVN$P_<*model_name*> which contain the URL's data, and the variables inside the SPARQL are replaced by s, p, o variables in order to match the MVN$P_<*model_name*> columns. This query is passed to the executed_query procedure as a variable of characters (query_template) and is executed inside the query_result cursor which returns to the user.

```
execute_query(..)
. . . .
  type r_cursor is REF CURSOR;
  query_results r_cursor;
  query_template varchar2(4000);
begin
. . . .
  query_template:=getQueryTemplate(..);
  open query_results for query_template ;
. . . .
 return query_results;
end;
```

Table 3.6: A part of the executed_query API which executes a query template.

Chapter 4

Optimization Solution

In this chapter, we explain in particular detail the optimization solution that we provide regarding the background queries, by dividing MashQL's background queries into two categories according to their optimization solution (General background queries and the N-level properties and N-level object queries). We explain what database summaries are, and, how these summaries help the general background queries to have a better performance standard. In addition, we explain what the BR-Algorithm is, and how this algorithm helps the N-level properties and N-level objects queries to run faster.

4.1 Optimization Solution

Our optimization solution gives emphasis on the performance issues that arise in section 2.5.1 MashQL's performance considerations , and it aims at bypassing these issues in order to execute all MashQL's queries successfully, efficiently and in a timely fashion . The module that bears this responsibility is called Query Optimizer.

Our optimization solution stands on two database techniques:

3. It creates smaller and focuses on data sets using data summaries providing background queries with a faster access to less data.
4. It fetches beforehand queries results and storing these results into the database in order to bypass self-joined operations during the queries execution.

Based on the above techniques, we divided MashQL's background queries into two categories according to their optimization solution. In the first category belong the background queries 1 to 13 where their optimization solution is supported using the method of summaries. The summaries are created by using the technology of materialized views. (We called these types of queries "General Background Queries"). In the second category belong the background queries 14 to 17 where their optimization solution is supported by using the BR-Algorithm, our novel fetching beforehand results algorithm. We will recall details about the BR-

Algorithm in a section further on in this chapter (We called these types of queries "n-level properties and n-level objects background queries").

4.2 General Background Queries Optimization Solutions

For the general background queries we use the summaries solution. All the data summaries are created using the technology of materialized views since materialized views have the following serious advantages[16]:

1. the purpose of the materialized view is to increase query execution performance.
2. the existence of a materialized view is transparent to SQL applications, so a database administrator can create or drop materialized views at any time without affecting the SQL applications.
3. a materialized view consumes storage space and must be updated when the underlying detail tables are modified .

Data summaries inherit all the materialized view advantages ,plus, instead of scanning and sorting all the data during the queries' execution course, the data are already sorted and pre-computed according to the general background queries requirements. With this technique (a) we have smaller data sets , as a result, less scanning data, less join data. (b)We save performance time, since the data are already sorted during the summaries creation course. (c) Most of the queries' results are already pre-computed. (d) Materialized views provide faster access time for data in relation to a normal table , and, (e), with our new database schema we eliminated the self-join problems that can exist in some general background queries, since the triple table is normalized into smaller data sets and the queries are transformed based on the new database schema, thus executed differently .

Our optimization solution proposes three categories of data summaries implemented as materialized views in the database tables used to support the RDF technology.

The first category of summaries creates 1-column schema materialized views for each column of the triple table. This category enumerates three summaries. The first summary in this category contains all the unique subjects , the second summary contains all the unique properties and the third summary contains all the unique objects.

These summaries help the background queries whose query is related only to the triple's subjects or objects or properties. For example , find a subject[s] which is/are equal to an input variable V introduced by a user or from a label L that is selected from a drop-down list. It is obvious that these types of queries will be answered very fast since all the graph's subjects are sorted in one summary which is by far shorter than the triple table. Additionally, these summaries can be combined or joined with other summaries in order to answer background query questions. For example the query: find an object[s] whose properties are equal to an input variable V. First, the query will get access to the summary that contains all the unique properties in order to bind the variable V and after that it will use these properties in order to find the correct object[s] joining the last category of summaries that will be explained in this section.

The second category of summaries creates a 2-column schema of materialized views. This category enumerates two summaries. The first summary in this category combines the subject-property filtering the properties of a specific RDF class (*rdf:type*), and the second combines the object-property filtering the properties of a specific RDF class (*rdf:type*) . These summaries answer the background queries that search for objects or subjects whose property belongs to the RDF class rdf:type.

The third category of summaries creates a 3-column schema of materialized views and enumerates only one summary. This summary is a snapshot of the triple table and contains the entire data source RDF graph. This summary answers global background queries questions and it is always used with a combination (e.g. join) with the above summaries. In this way, instead of the self-join triple table it-self we join the triple table(where in its place we have this summary) with one or more summaries above which are shorter in size. As a result, the query is executed faster since the summaries' are by far smaller rather than the triple table.

As we mentioned above, our solution enumerated six summaries in total that are created during the loading process and recreated every time that the data source is updated in the system. For faster data manipulation, the values for the subject, the property and the object are converted from their corresponding lexical values in to big numbers which are faster in logical

comparisons rather than the characters, and also, all summaries are sorted and indexed very carefully. As we already mentioned, all MashQL's queries are translated in to the SPARQL language and this SPARQL code is executed on the RDF engine's backend database, as results. In order to run this SQARQL code into our solution we need to transform the background queries SPARQL to SQL-BASED query according our optimization solution. Thus, for each background query we create a corresponding SQL query template that is ready to run in our optimization solution and it is stored as text in a MashQL Query Optimizer's repository . When the MashQL Query Optimizer has received a background query for execution , it binds the query's variables in the background query's corresponding template and instead of running the background query it executes the background query's corresponding template on our optimization solution summaries.

4.3 N-level objects and N-Level properties Background Queries Optimization Solution

As we already mentioned, during the formulation algorithm, MashQL's users need to expand some properties (P) or some objects (O) in order to declare its path. This action creates the n-level properties and n-level objects background queries , where they present a very high interest , since for huge data sets these queries sometimes are not responding or they have a very poor performance. The main reason for the bad performance derives the queries' tendency to make so much self-joining summaries as the number of levels that the queries need to expand to. The problem becomes bigger regarding very large RDF graphs with million or billion triples, since the million or billion triple table is self-joins itself to many times and the RDBMS is not able to handle too many data very fast . The table 4.1 shows the query's execution plan of a background query 16 at Level 5 that is executed by using Oracle's SEM_MATCH. The Oracle's query optimizer's execution plan shows that, the Oracle , self-joins five times the internal table RDF_LINK$ that contains the graph's data , and after that ,it sorts the results and returns the data back to the system. The same behavior pattern appears in all the n-level properties and n-level objects background queries. As a conclusion, when these types of queries are executed, the RDBMS must read and scan n * number of triples table data , a value that the RDBMS is not able to handle very fast.

Background Query 16 in Level 5

```
SELECT o4 FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)(?O3 ?P4 ?O4)',
SEM_Models('YAGO'), null, null, null))
group by o4;
```

Oracle's Query Optimizer Execution plan

```
SELECT STATEMENT
 SORT AGGREGATE
  VIEW
   HASH GROUP BY
    NESTED LOOPS
     NESTED LOOPS
      VIEW

HASH JOIN
       PARTITION LIST SINGLE
        TABLE ACCESS FULL      RDF_LINK$
       HASH JOIN
        PARTITION LIST SINGLE
         TABLE ACCESS FULL      RDF_LINK$
        HASH JOIN
         PARTITION LIST SINGLE
          TABLE ACCESS FULL     RDF_LINK$
         HASH JOIN
          PARTITION LIST SINGLE
           TABLE ACCESS FULL    RDF_LINK$
          PARTITION LIST SINGLE
           TABLE ACCESS FULL    RDF_LINK$
     INDEX UNIQUE SCAN          C_PK_VID
    TABLE ACCESS BY INDEX ROWID RDF_VALUE$
```

Table 4.1: Oracle's execution plan for the background query 16 Level 5 using Oracle's SEM_MATCH table function

The Oracle's SEM_MATCH query in the table 4.1 is an example of the n-level objects background queries. These queries have the characteristic that , the query's resulting objects (variable O) in the first SPARQL pattern become the input subjects in the second SPARQL pattern and the resulting objects (variable O1) in the second pattern become the input subjects in the third SPARQL pattern. This movement of resulting objects continues till the SPARQL pattern N. The last SPARQL pattern always returns the query's results. Those results are contained in SPARQL's variable O4 (for the objects) and the P4 (for the properties). As a result, the queries tend to create a parent-child relation or a chain of blood-related objects from the first SPARQL pattern to the N SPARQL patterns (the objects of objects and so on) and they return the results that are found in the N SPARQL pattern. This

relation illustrates that, the objects or properties that exist in the first pattern have 1- Blood-Relation with one or more subject[s] in the first SPARQL pattern. The objects or properties that exist in the second pattern have 2- Blood-Relation with one or more subject[s] in the first SPARQL pattern, and the objects or properties that exist in the N pattern have an N- Blood-Relation with one or more subject[s] in the first SPARQL pattern. Thus, based on the above relations, if we calculate all the Blood-Relations between all the graph's subjects with the rest of the graph's objects, we can easily answer very fast any n-level objects and n-level properties background queries, since we will need to return only to the query's N-Blood-Relation results.

We probed the Floyd-Warshall algorithm, a classical graph's algorithm that calculates the graph's transitivity closure and all graph's pairs shortest paths , for a weighted graph. The graph's transitivity closure provides reachability information about a graph's nodes i.e. a node v can reach the node x, but it cannot tell you how the connection is made between any two nodes. For example [28], given a directed graph G = (V, E) with vertex set V = {1, 2,...,n}, we may wish to find out whether there is a path in G from i to j for all vertex pairs $i, j \in V$. The transitive closure of G is defined as the graph G* = (V, E*), where E*= {(i, j) : there is a path from vertex i to vertex j in G}. The Floyd-Warshall algorithm finds the lengths (summed weights) of the shortest paths between all pairs of vertices while it does not return details of the paths themselves. As a result, by running the Floyd-Warshall we will get all the N-Blood-Relations between the graphs' nodes. The problem that has been given birth is focusing on the fact that RDF graphs contain different types of data rather than the data that are hosted by the classical directed graphs i.e. RDF graphs can host biological data, social networks, web data and lot of other categories of data [29]. To understand the problem , imaging a graph, where its edges are expressed in numbers, characters or user-defined class-data types. Since the graph's edges in the RDF can belong in different classes e.g. rdf:type , parent-of, brother-of, sister-of, friend-of, employee-of, etc the graph's node shortest paths is not possible to be calculated[30]. Based on the above issue, the simplest way to compute the transitive closure of

an RDF graph is to assign a weight of 1 to each edge of E and run the Floyd-Warshall algorithm. If there is a path from vertex i to vertex j, we get $d_{ij} < n$. Otherwise, we get $d_{ij} = \infty$ [28].

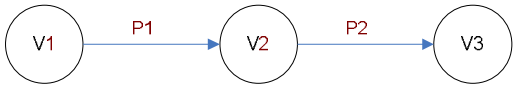


Figure 4.1: A simple RDF graph (G)

Consider the sample graph in figure 4.1 , the Floyd-Warshall algorithm implementation produces the following 3 X 3 matrix , which contains the algorithm's results

	V1	**V2**	**V3**
V1	-	1	2
V2	∞	-	1
V3	∞	∞	-

The Floyd-Warshall matrix shows that the node V1 can reach V2 at cost 1, and the V1 can reach node V3 at cost 2. By the same token, node V2 can reach V3 at cost 1. On the other hand, node V3 is not able to reach any node. As a result , from the Floyd-Warshall matrix we can easily answer any n-level objects and n-level properties background queries, since we need to select all the objects that belong to the query's n-level.

Unfortunately , the Floyd-Warshall algorithm can not be applied in our problem for the following reasons:

1. As far as very large RDF graphs are concerned, the algorithm needs as an input a very huge matrix. Thus, speaking about real data, these large RDF graphs will produce a million X million matrix(graph's subjects UNION graph's objects X graph's subjects UNION graph's objects) and if we take into consideration that traditional transitive closure algorithms are mostly designed for main memory operation, it will be very difficult for the memory to handle this huge data.
2. The RDF's graphs are already stored in the database as 3 X N relational table making the RDF data permanently available (subject, property, object X rows). Thus, in order to avoid the big I/O between Main Memory and Disks, we decided to use the RDBMS services in order to find a solution to the problem.

We proposed the Blood Relation Algorithm (BR-Algorithm) a database approach which

calculates the blood relation level between all graph's subjects in relation to the graph's objects that belong to a graph. The algorithm ignores the graphs' edges' weight i.e. the graph's property value ,and it measures the blood relation level from a subject to the rest of the objects that can be found in the same path. If there is a path from subject i to object j, the BR_{ij} is the number of neighbour objects that the subject i needs to leave behind in order to reach the object j .All the objects that are unreachable from subject i are ignored. The results are caught in a partition table ,in the form of {BR level, Si,<triple>} , where the table's partition key is the BR level .

```
# s = Subject
# p= Property
# o= object
# TEMP is a temporary table
# BR$ is a partition table which contains algorithm's results
# maxlevel(G) returns the graph's maximum BR's level
# maxL an integer number indicating graph's maximum reach level
FOR each si in G DO
  Level=0
   SELECT s,p,o FROM G WHERE s=si
   INSERT o IN TEMP
   INSERT Level,si,s,p,o IN BR$
    LOOP
      Level ++
      SELECT s,p,o FROM G WHERE s IN
                        ( SELECT o FROM TEMP where o != si)
      DELETE TEMP
      INSERT o IN TEMP
      INSERT Level,si,s,p,o IN BR$
  EXIT WHEN Level<=maxlevel(G) OR Level=maxL
END FOR
```

Table 4.2: BR-Algorithm pseudo-code

Table 4.2 shows the algorithm's pseudo-code which behaves as follows:

For each unique subject that belongs to graph G , the following actions take place:

1. The algorithm selects all the objects and properties that are directly connected with examined subject i.
2. The algorithm saved the Si's results {BR,si,<triple>} in the BR table , and the triple's objects in the TEMP table.
3. The algorithm loops , by selecting all the objects and properties that are directly connected with the objects that are found in TEMP , increasing each time the BR level by 1 and storing the results {BR,si,<triple>} in the BR table. For every loop the

TEMP table is resetting with object results of the previous objects. The algorithm exits the loop when the S[i] path is discovered.

4. The algorithm continues to select all the objects and properties that are directly connected with examined subject i+1 and finished when all subjects are discovered.

For example , consider the graph illustrated in figure 4.1. It is has the triples <*V1->P1->V2*> and <*V2->P2->V3*>. The degree of neighborhood between a subject V1 and the object V2 is 0 (This is denoted as BR (V1, V2) =0 or V1 level 0 V2) and the degree of neighborhood between a subject V1 and the object V3 is 1. In the same way, the degree of neighborhood between a subject V2 and the object V3 is 0. Node V3 can not be used because it does not have any neighbor object[s]. If we apply the BR-algorithm in this graph , it will create two groups or partitions , the group 0 and 1 , with the following BR entries:

BR	Si	S	P	O
0	V1	V1	P1	V2
0	V2	V2	P2	V3
1	V1	V2	P2	V3

Table 4.3: BR$ table results for the graph G

When the BR-algorithm calculates the graph's paths for each subject, the n-level properties and n-level objects background queries become simple in their execution , since the Query Optimizer needs to fetch only the result from the BR$ table, calculating only the query's n value which is the number of SPARQL's patterns - 1 that exists in the original SPARQL query. In the same way, as the general background queries , the Query optimizer maintains SQL-BASED templates for each n-level property and n-level object background query that are ready to run in our optimization solution and it is stored as text in a MashQL Query Optimizer's repository . When the Query Optimizer has received a background query for execution , it binds the query's variables in the background query's corresponding template and instead of running the background query, it executes the background query's corresponding template in our optimization solution summaries. The following SQL code compares the execution of the 2 levels of objects expansion background query using Oracle's

SEM_MATCH and the MashQL Query Optimizer against the graph G that we used in our example.

Oracle

```
=======
SELECT o2 FROM TABLE
(SEM_MATCH
 (
    '(?S1 ?P1 ?O1)(?O1 ?P2 ?O2)',
     SEM_Models('G'), null, null, null)
)
group by o2;
```

Our Solution

```
===========
SELECT o FROM BR$G where br=1
```

Chapter 5

Experimental Methodology

In this chapter we present the Experimental Methodology that we used in order to implement the optimization solution that we proposed in chapter 4 . Since the Oracle 11g semantic technology is chosen to be the MashQL's RDF engine [1,2,3,4,5] our optimization solution's database objects are created based on this technology. More analytically , in this chapter ,we mention information about the data summaries and BR-algorithm that we created on top of Oracle's technology. Additionally , we state the problems that we found during the BR-algorithm implementation and we provide the solutions that we found in order to solve these problems i.e. we present how our new partition schema against any RDF graphs helps BR-algorithm to be executed faster.

5.1 Summaries Implementation

Our optimization solution for general background queries creates three types of data summaries. These summaries are created using oracle's materialized views technology (their advantages are presented in chapter 4 section 4.2) and their names starting from prefix MVN$ and ending with suffix _<graph_name> (the graph's name is the same name that is given when the graph is loaded into the Oracle database. The oracle technology uses the term model to express the RDF graph's names). As you understand, each graph has its own summaries that are created after the graph's been loaded in the oracle database (their creation script can be found in appendix D).

The first type of summaries creates a summary of data for each column of the triple table. The first summary contains all the unique subjects (MVN$S_<graph_name>), the second summary contains all the unique properties (MVN$P_<graph_name>), and the third summary contains all the unique objects (MVN$O_<graph_name>).The second type of summaries creates a summary of data contains all the unique subjects and objects that their properties belong to the rdf:type class(the views are the MVN$T_S_<graph_name> for the subjects set

and the MVN$T_O_<graph_name> for the objects set).The third summary type is a snapshot of the triple table (MVN$<graph_name>) that contains the entire RDF graph.

5.2 BR-Algorithm Implementation

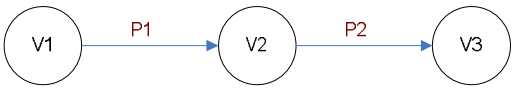
In the section of chapter 4 , we presented the BR-Algorithm's speudo-code. In this section we will present how the algorithm gets access to the database summaries in order to calculate the n-level objects and n-level properties background queries results and stores them in the database. Initially, the algorithm reads the MVN$S_<graph_name> summary which contains all the graph's unique subjects. For each subject Si that belong to the candidate graph, the algorithm finds all the Si's neighbor objects by selecting the MVN$_<graph_name> summary. All the neighbor Si's objects that are selected from the MVN$_<graph_name> are stored in the database under the BR$<graph_name> table and these objects are considered to belong to level 0 in relation to the subject Si. After that, the algorithm calculates the Si's level 1 objects, by finding the neighbor objects of the Si's neighbor objects that have been discovered before. All the results are appended in the BR$ table. This process continues until the algorithm discovers all the relations between subject Si and the nodes that exist in the graph. The algorithm is terminated when all the subjects are discovered. The final product of this algorithm is a table of adjacencies between the subjects and the graph's nodes and it contains the following information

1. All the available paths from subject to its neighbor objects.
2. The BR level between a subject and their neighbor objects.

In using Figure 5.1, we demonstrate how the algorithm works using the graph G that we introduced in chapter 4. The algorithm first reads the subjects from the summary MVN$S_G (V1,V2). Starting from subject V1 , the algorithm finds all the V1's neighbor objects by selecting the MVN$_G summary. The result of this selection is node V2. The algorithm stores into table BR$G2 the level number (level 0),the examined node (V1) , and V2's triple information (V1->P1->V2). After that, the algorithm calculates V2's neighbor objects by selecting the MVN$_G summary. The result of this selection is node V3. The algorithm stores into table BR$G the level number (level 1),the examined node (V1) , and V3's triple

information (V2->P2->V3). After that, the algorithm calculates V3's neighbor objects by selecting the MVN$_G2 summary , a selection that does not return any value , indicating that the V1 path is completed. After that the algorithm takes the next subject that is the V2. For the subject V2 the algorithm finds all the V2's neighbor objects by selecting the MVN$_G2 summary. The result of this selection is node V3. The algorithm stores into the BR$G table the level number (level 0),the examined node (V2) , and V3's triple information (V2->P2->V3). After that, the algorithm calculates V3's neighbor objects by selecting the MVN$_G2 summary , a selection that does not return any value , indicating that the V2 path is completed. Finally the algorithm is terminated since all subjects in MVN$S_G2 are examined.

RDF graph "G"



Triple Table for graph "G"

MVN$_G

S	P	O
V1	P1	V2
V2	P2	V3

BR$G

BR	NODE	S	P	O
0	V1	V1	P1	V2
0	V2	V2	P2	V3
1	V1	V2	P2	V3

MVN$S_G

S
V1
V2

Figure 5.1: Explaining BR-Algorithm by using the simple RDF graph "G"

In the Oracle RDBMS the BR-Algorithm can be computed by using hierarchical queries [17] with the START WITH and CONNECT BY clauses as shows query below . The START WITH clause is optional and specifies the rows that are the root(s) of the hierarchical query. If you omit this clause, then Oracle uses all rows in the table as root rows. The CONNECT BY clause specifies the relationship between parent rows and child rows of the hierarchy. The CONNECT BY PRIOR is a condition it refers to the parent row .The BR-Algorithm's results are saved during the algorithm's execution , an advantage that is provided by the oracle's CREATE TABLE command.

The initial implementation of the algorithm according to the example above is demonstrated using the following oracle's SQL code:

```
CREATE TABLE BR$G
as
SELECT a.s as NODE ,LEVEL as BR ,g.S,g.P,g.O
FROM MVN$_G g,( SELECT distinct s FROM MVN$S_G) a
start with g.s=a.s
CONNECT BY prior g.o=g.s
/
```

5.2 BR-Algorithm's problems

We run the BR-algorithm on our datasets (see chapter 6 , section for dataset description) and we came confront with the problem that the algorithm could not completes for the majority of our datasets, as well as , the size of the BR$ table was increasing disproportionately in relation with graph's triple table (The algorithm completes only for the datasets FLICKR, DBLP). It is obvious that , the algorithm produces a huge number of BR entries (<br,si,s,p,o>) due to fact that the algorithm explores all the possible paths per level for each subject , but using the statistics on table 5.1 we realized that the size of the triple table and the number of the subjects that a graph has are not the main causes of our problem.

Dataset name	Number of triples	Number of unique Subjects	Est. Number of graphs' levels	Est. Percentage of Graphs' cycles
SEMDUMP	10083	1196	More than 70	More than 90%
FLICKR	2298849	2298849	1	0
DBLP	8424187	1156727	2	More than 1%
YAGO	18343546	4339591	More than 60	More than 35%
DBPEDIA	130822521	10534382	More than 3	More than 60%

Table 5.1: Datasets statistics (Number of triples, Number of unique Subjects , Estimated Number of graphs' levels, Estimated Percentage of Graphs' cycles)

For example the problem has also appeared in the SEMDUMP dataset which has a very small number of triples and a very small number of subjects. On the other hand , the SEMDUMP dataset has most profound graph (more than 70 levels) and the most number of cycles in its graph (90% of its subjects and objects are included in a cycle). Based on this observation, we

conclude that any optimization solution against the BR-algorithm must focus on the following factors:

1. Explore fewer triples.
2. Explore fewer subjects.
3. Explore the graph until we reach a constant depth.
4. Ignore graph's cycles

5.2.1 BR-Algorithm's optimization

The BR-Algorithm's optimization tries to eliminate and get around the problems that we focus on the previous sections.

5.2.1.1 Explore fewer triples and fewer subjects

In order to explore fewer triples and fewer subjects during the algorithm's runtime, we break the RDF graph into three sub-graphs. The first sub-graph contains the triples from those subjects which have no incoming arcs. They have only outgoing arcs and we call them "thin subjects" (figure 5.2, B). The second sub-graph contains the triples from those objects which have no outgoing arcs, they have only incoming arcs and we call them "thin objects" (figure 5.2, D). The third sub-graph contains the triple from those subjects or objects that have incoming and outgoing arcs and we call these nodes "fat nodes" (figure 5.2, C). Consider the RDF graphs G(s,p,o) the thin subjects (TS), the thin objects (TO) and fat-nodes (FN) deriving the following sets' relations :

TS= G(s) MINUS G (o)

TO=G (o) MINUS G(s)

FN = G(s) UNION G (o) WHERE G(s) NOT IN TS AND G(o) NOT IN TO

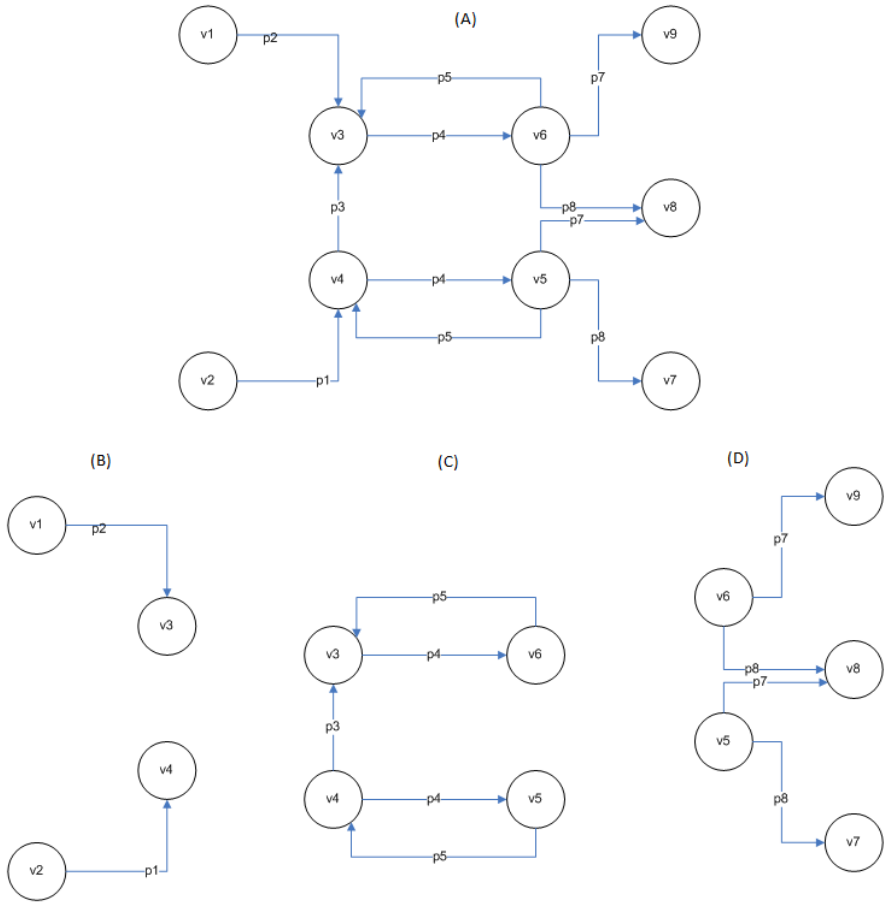


Figure 5.2: RDF graph's partitioning. (A) The original graph,(B) Thin subject sub-graph,(C) Fat nodes sub-graph,(D) thin object sub-graph.

The idea behind this partition schema derives from the thin subjects and the thing objects constituting the graph's edges, the thin subjects are the initial edges and the thin objects are the final edges. By moving outside these categories of nodes, we create a new graph that is much smaller than the initial. Think of the cutting edge photography technique , where its cut photograph edges are so in order to minimize its size. (up to 4 times smaller). The BR-algorithm runs against the new graph which has fewer subjects and fewer triples. If the new graph has zero triples, the BR-algorithm is not executed since the max graph's path level is 1 indication that the graph is discovered. When the algorithm completes it, it will create the BR$ table that contains the paths of all the subjects that include fat nodes. Additionally, the algorithm merges the BR$'s results with the thin objects of the sub-graph. The algorithm checks from the BR$ table, which nodes have in their path an object node and which is the

level of any subject from the thin object sub-graph . Regarding the matching nodes, the algorithm adds in BR$ table one level after all the corresponding objects to the matching subjects. The thin subjects will inherit their paths from the fat nodes sub-graph, since their object nodes are included as subjects in the fat nodes sub-graph and they will be calculated using the BR-algorithm. The results for the subjects that are included in the thin subjects sub graph are fetched during the SQL-Based NL-O and NL-P Background Queries at queries' runtime.

Dataset name	Number of triples	Number of triples (new graph)	Number of unique Subjects	Number of unique Subjects(new graph)
SEMDUMP	10083	10079	1196	1194
FLICKR	2298849	0	2298849	0
DBLP	8424187	902464	1156727	445551
YAGO	18343546	5493650	4339591	2169718
DBPEDIA	130822521	71420074	10534382	6495600

Table 5.2: Graphs' size and graphs' number of subjects before and after graphs' partitioning.

By using this partition schema we achieved to reduce the number of subjects and the total number of triples during graph's discovering reducing the algorithm's completion time and producing less BR entries in the BR table.

5.2.1.2 Explore the graph until we reach a constant depth

Even though , the reduction of number of subjects and the total number of triples during graph's discovering make the algorithm's completion time faster, for the graphs expanding in at many levels, the BR$ table continues to be growing worryingly. It is obvious, that the subject's path exploration up to the end, is a very difficult task, especially for very complex RDF graphs that expand at many levels. Having this in mind, first, we do not calculate level 1 paths for each subject, since the results of level 1 paths are the triple table itself, and, second, we limit the path's depth to a constant level according to the max graph's level, taking into consideration that, if any NL-O and NL-P Background Queries need to expand further, they have to be able to do it in real time. With this solution we reduce by far the number of BR$ table entries and, as a result, the algorithm is implemented on completion time.

5.2.1.3 Ignore graph's cycles

RDF graphs are comprised by a large number of directed cycles (a large number of nodes are connected in a closed chain).The presence instances of cycles between nodes produce infinite valid paths for the subjects since max levels of the graph touch the infinite. This fact creates enormous problems in the algorithm that we already mentioned, thus, the rejection of cycles is considered necessary. In order to achieve that, we used the NOCYCLE option coming with the CONNECT BY clause which can handle graphs that contain cycles by generating the row in spite of the loop in user data.

5.3 The Final implementation of a BR-Algorithm

The following SQL code provides the final BR-Algorithm implementation. During the algorithm runtime, all the data are stored in the BR$ table which is a partitioning table having as a partition key the BR value. The BR$ table is created during the BR-algorithm execution course and, it can be altered after its creation (e.g. to add a new partition)

The clause WHERE LEVEL between 2 and 10 defined the constant maximum graph's exploration level (For example, the value 10 is dependent on graph's max level. Some other graphs may have lower or higher value).

```
CREATE TABLE BR$_<GRAPH_NAME>
 (
  BR   ,
  SI,
  S    ,
  P    ,
  O
 )
PARTITION BY LIST (BR)
(
        PARTITION q0 VALUES (2),
        PARTITION q1 VALUES (3),
        PARTITION q2 VALUES (4),
        PARTITION q3 VALUES (5),
        PARTITION q4 VALUES (6)
        PARTITION q5 VALUES (7),
        PARTITION q6 VALUES (8),
        PARTITION q7 VALUES (9),
        PARTITION q8 VALUES (10)
```

```
)
TABLESPACE <RDF_TBS>
NOLOGGING
COMPRESS
PARALLEL <CPU_COUNT>
AS
SELECT sbj.s as SI, LEVEL as BR , g.S,  g.P, g.O
FROM MVN$FAT_<GRAPH_NAME> g,
          ( SELECT s FROM  MVN$S_<GRAPH_NAME>)  sbj
WHERE LEVEL between 2 and 10
start with g.s= sbj.s
CONNECT BY NOCYCLE PRIOR g.o=g.s
/
```

Chapter 6

Evaluation

In this chapter we provide evaluation results for the RDF loader, the module that loads RDF data resources into the MashQL database. Also, we present comparison results in relation to the performance of MashQL background queries, using Oracle's SEM_MATCH and our optimization solution.

6.1 Benchmark Definition and Machine's specification

Our Benchmark includes datasets from DPBEDIA, YAGO, DBLP and from Semantic Web Conference Corpus(SEMDUMP).The table 6.1 provides a description for these datasets. The datasets are in Native Triple format (NT), they were downloaded and stored in a local server (The MashQL server, see table 6.2 on machine's specifications). Our Benchmark includes a small, medium and huge datasets (their size are according to the number of triples per dataset , see figure 6.1 for more details) .The Benchmark is created in order to evaluate the RDF loader loading time and to compare the performance of MashQL background queries using Oracle's SEM_MATCH and our optimization solution.

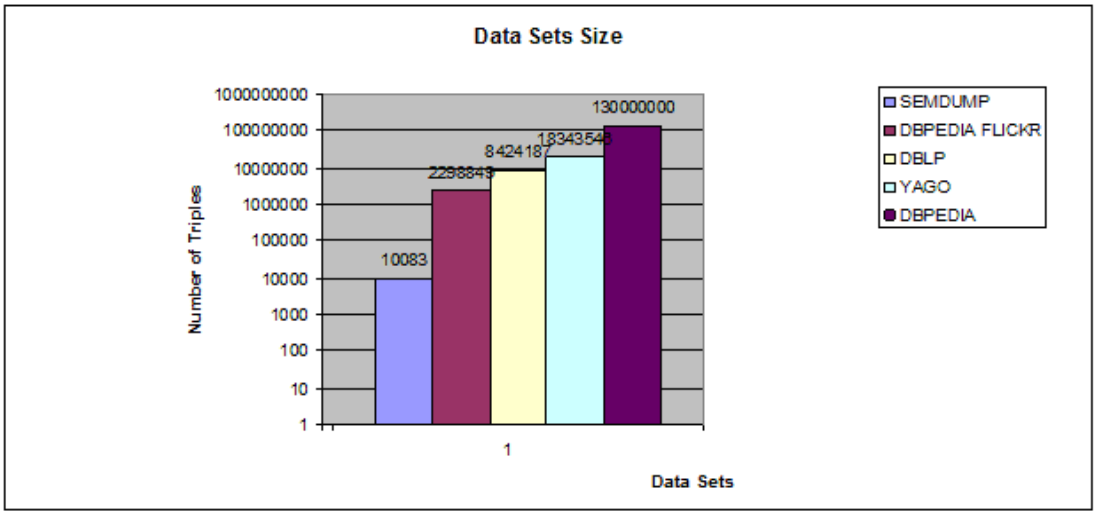


Figure 6.1: Datasets number of triples used for the Loader evaluation

SemDump	From Semantic Web Conference Corpus that contains information on papers that were presented, people who attended, and other things that have to do with the main conferences and workshops in the area of Semantic Web research.
Yago	A light-weight and extensible ontology with high coverage and quality. YAGO builds on entities and relations and currently contains more than 1 million entities and 5 million facts.
Dblp	Contains a large number of bibliographic descriptions on major computer science journals and proceedings. The server indexes contains more than half a million articles and several thousand links to home pages of computer scientists.
Dbpedia	DBpedia knowledge base describes more than 3.4 million things, out of which 1.47 million are classified in a consistent ontology, including 312,000 persons, 413,000 places, 94,000 music albums, 49,000 films, 15,000 video games, 140,000 organizations, 146,000 species and 4,600 diseases.
Dbpedia-Flickr	A part of dbpedia dataset and it is used for evaluation purposes only.

Table 6.1: Datasets Description

Table 6.2 describes the machine's specifications that we use to evaluate the MashQL Loader.

Processor	Intel (R) Core(TM) i7 CPU Q720 @ 1.60GHz 4 cores 1.6GHz , 2 threads per core
Physical Memory	8GB DDR3
Hard disk	320G 7200 rpms
Operating System	Windows 7 Home Premium 64 Bits , 320GB Hard disk
Database	Oracle Database 11g Enterprise Edition Release 11.1.0.7.0 - 64bits
Version Database	Buffer Cache Area and Others 4,5G
Memory	Users' Sort Area 1,5G
Distribution Database	20G Bytes
Temporary	
Space for sorting	
In Disk	

Table 6.2 RDF Loader 's machine specifications

6.2 Experimental Results & Discussion

6.2.1 RDF Loader Evaluation

In this section we provide evaluation results for RDF loader, the corresponding module that loads RDF data resources into the Oracle's RDF model database.

6.2.1.1 Methodology

In order to evaluate the RDF Loader, we used a very simple methodology. We stored our benchmark locally in a MashQL Server and later we loaded it into the MashQL database using our RDF Loader. Our experiment , measures the time that the RDF Loader needs to load and register a dataset into the Oracle's RDF model database.

The experiment does not measure the time that is wasted during the dataset downloading from the web, the time that is wasted for any RDF files transformations (i.e. convert RDF/XML to NT), and also, it does not calculate the time that will be wasted during the creation of Query Optimizer Objects (Summaries and BR-Algorithm's results).

6.2.1.2 RDF Loader Experimental Results

The RDF Loader's experiment aims at measuring the RDF loader's loading performance standard for small, medium and huge datasets and at finding the proportion of the loading time of files concerning their size in order to give priority to the files with the faster loading time.

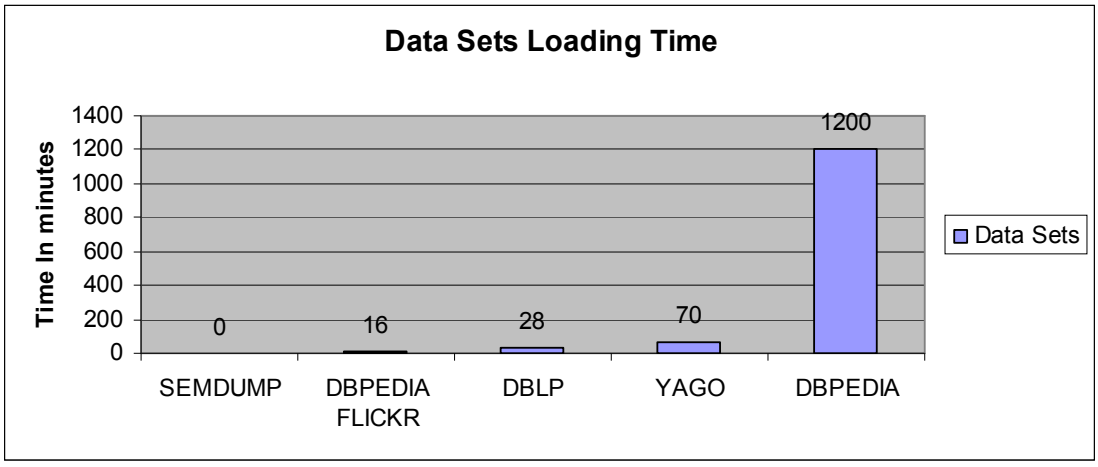


Figure 6.2: Datasets Loading Time results

Figure 6.2 shows the loading time results in minutes for each dataset. It is clear that the loading time is proportional to the dataset's size, a fact that is to be expected. For example, for the largest dataset, the loading time reaches up to 1200 minutes (DBPEDIA), for the midium-sized dataset, the loading time reaches up to 28 minutes (DBLP), and for the smallest dataset the loading time is less than 0 minutes (SEMDUMLP). Based on these results, the statistical number of 80000 triples per second that derived from the experiment's results is considered a very good number to satisfy MashQL's loading needs.

Since the MashQL is a web application , the loading phase must be fast .Of course, the loading phase is dependent on the data source's size which is requested by the MashQL user, thus, we want to give priority to the files having the less loading time in relation to their size.

According to the results in figure 6.2, we set 30 minutes as the maximum loading time. This value derives from the loading results for the small and the midium-sized datasets (10,000 triples are loaded after 0.5 minutes time (SEMDUMP) and up to 9,000,000 triples are loaded after 28 minutes time (DBLP)).The proportional size in bytes for these datasets are 1MBytes to 1,5GBytes respectively, thus , these sizes are considered as our boundaries for the RDF resources that can be accepted by the system. All the rest RDF data sources, whose file size is larger than 1.5GBytes will be loaded on a very low priority and will be have different management criteria (such as refresh interval time etc.).An example of these data sources is the DBPEDIA which needs 1200 minutes time to be loaded in the system.

From statistics in figure 5.3 , which are Oracle's official results according to the Oracle 11g bulk loader which is exactly the same component as those have been that already presented , it is clear that there is more space to improve further our RDF loader performance, making internal changes inside the Oracle's loader configuration files that are suggested by [13]. The statistics results derive from Oracle New England Development Centre [26] and show the performance loading time on various version of LUBM [27] dataset . The experiment uses a Linux-based commodity personal computer (1 CPU 3GHz, 2GB RAM) and it shows the efficiency of Oracle's 11g to load RDF data.

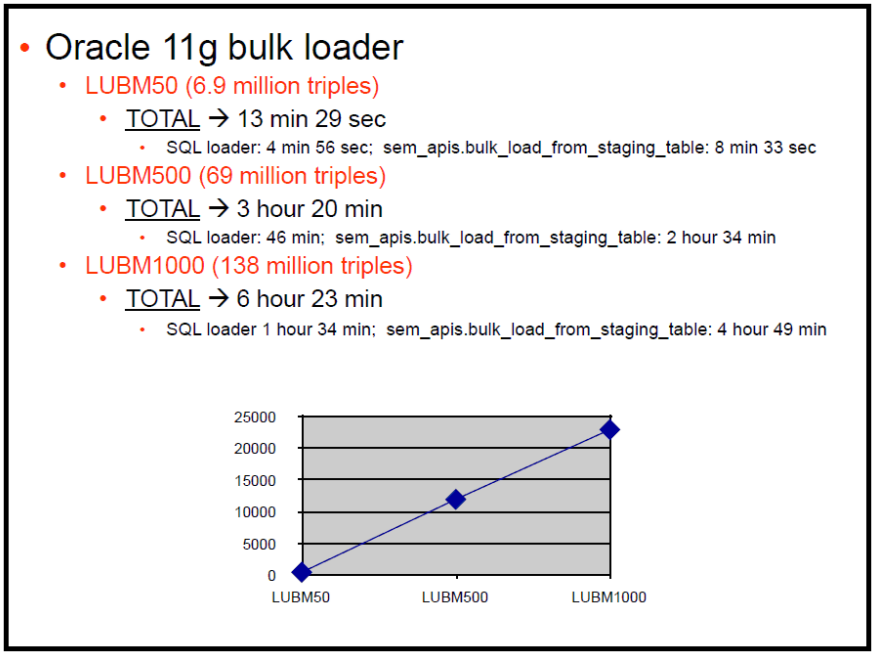


Figure 6.3: Datasets Loading statistics from Oracle New England Development Center [26]

6.2.2 MashQL Background Queries Evaluation

6.2.2.1 Methodology

We evaluated the MahQL background queries by splitting the Background queries into two groups , the general queries (Queries 1-13) and the n-level objects and n-level property queries (Queries 14-17).

General Queries (Queries 1-13) : For the general queries we created two sets of queries, the Oracle's queries using the SEM_MATCH table function and a set of background queries according to the structure of our optimization solution. On both solutions, we just count the number of return rows with out displaying the results (SELECT count(*) from (*BQ_QUERY*)). Both sets (oracle's and ours, are compared when returning the same number of rows. In this way, we assess the queries' correctness. For each group of queries, we get a response time (in seconds) of queries by running it five times for each set. We flush the database's cache in order to have a clear response time for each query.

N-Level Objects and N-Level Properties Queries (Queries 14-17) : For these queries, we created two sets of queries, the Oracle's queries, using the SEM_MATCH table function, and, a set of background queries, according to the structure of our BR-Algorithm solution. For each query we fetch data from the graph' levels 2, 3,4,5. On both solutions we just count the number of return rows with out displaying the results (SELECT count(*) from (*BQ_QUERY*)). Both sets (oracle's and ours are compared when returning the same number of rows. In this way, we assess the queries' correctness. For each group of queries, we get a response time (in seconds) of queries by running it five times for each set. We flush the database's cache in order to have a clear response time for each query.

6.2.2.2 General Queries Evaluation

In the general query evaluation we use three datasets the SEMDUMP, DBLP and YAGO. Figures 6.3 , 6.4 and 6.5 present the results of the queries' response times in seconds on both sets (oracle's and ours for each dataset (Appendix D contains the tables with the actual values for each chart below).

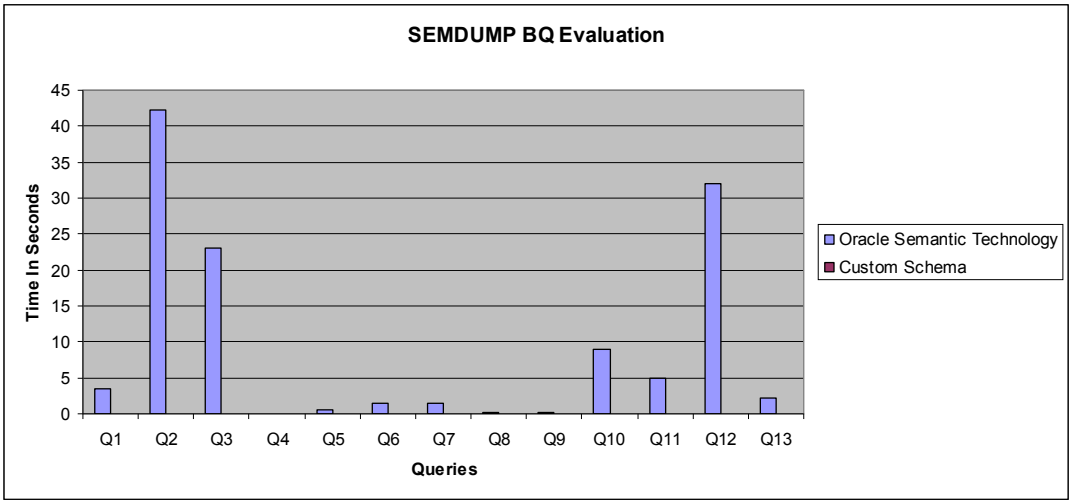


Figure 6.4 : SemDump response time results for queries 1-13

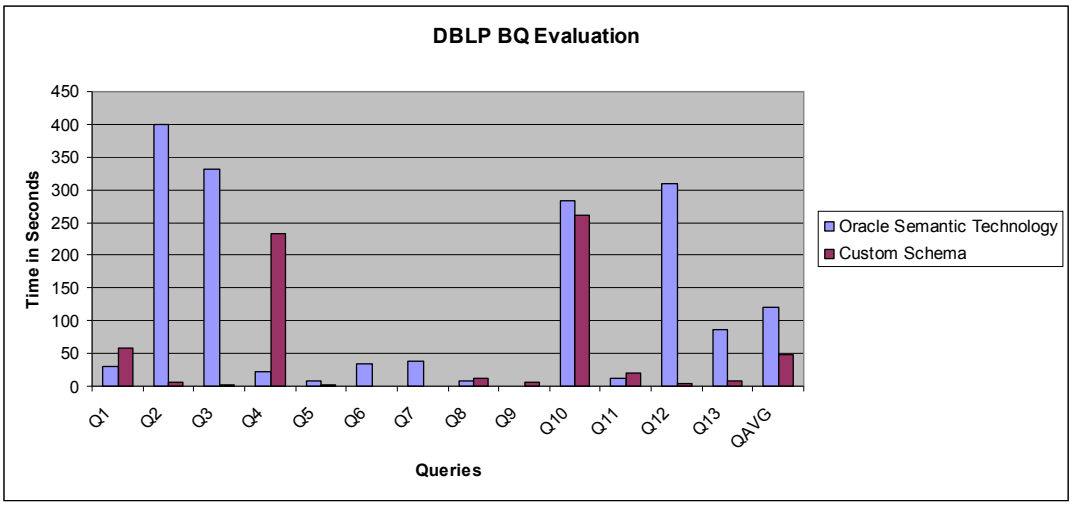


Figure 6.5: DBLP response time results for queries 1-13

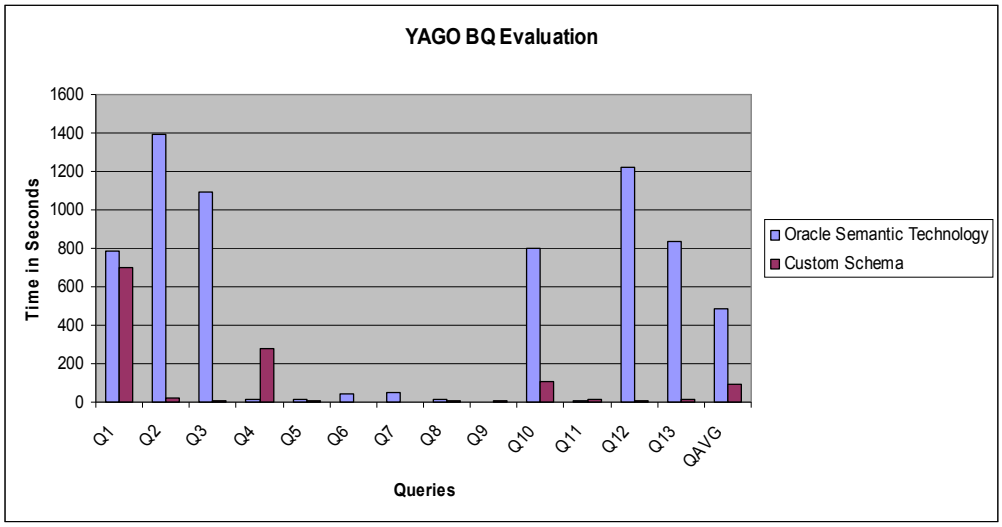


Figure 6.6: YAGO response time results for queries 1-13

5.2.2.3 Discussion for General Queries Evaluation

From the experiments that are presented in the above charts, it is obvious that for all datasets our optimization solution response is faster than oracle's SEM_MATCH, except for query 4. The reasons for this performance improvement concentrated on the followings factors:

1. Our optimization solution is created in order to help the execution course of all the MashQL's background queries. On the other hand, oracle's technology is designed to be able to query any RDF graph at random.
2. For each graph that is loaded into the database we created a set of summaries. These summaries are created by using the oracle's material view technology which provides faster access to the data concerning relational tables [16].On the other hand, Oracle's technology uses a partition table (the triple table) to store all of he graphs data .From our results, even if, Oracle's SEM_MATCH get access to a single partition during its execution process (each partition contains data from one RDF source only), the MashQL queries that run on material views are executed faster.
3. The database summaries have the advantage that, instead of scanning and sorting all the data during the query's execution the data are already sorted and pre-computed according to the general background queries' requirements. The sorting of data is done during the summaries creation course. With this technique the queries' performance time is improved, since, (a) there is no sorting activity during the queries' execution

course (b) the created materialized views are smaller in size rather than the oracle's triple table. As a result, the queries scan less data, (c) the queries that need to join various summaries, manage less data, and (d) the majority of the queries' results are already pre-computed, thus, its results are returned very fast.

4. Some of the MashQL background queries suffer from the RDF's self-join problem which reduces the queries' performance standard, since the triple table self-joins itself during the queries' execution course. Using the summaries solution, any filtering and joining of each MashQL's query always happen within the smaller scope rather than the whole RDF graph. It would dramatically reduce the self-join cost, since the triple table is divided into smaller sets (the data summaries). During the queries' execution course, a set of summaries join in order to produce the queries' result. The performance of these types of queries is achieved since the aggregation of joining summaries are scanning less data rather getting than access to the triple table.

Unfortunately, query 4's performance standard is very poor and it is considered an exception concerning the rest of the queries. The reasons for the query's bad performance standard is due to the fact that the query needs to join two times the summary MVN$_<graph_name> which contains all the graph's data. As a result, the query's performance is proportional to the MVN$_<graph_name>'s size, in other words, the bigger the size of MVN$_<graph_name>, the later will the query be executed. Since, our target is to execute all the background queries very fast, we force the Query Optimizer to execute the query 4, using Oracle Technology as if we ran all the MashQL's formulated queries.

More analytically, the main results are the following:

1. For SEMDUMP dataset, our optimization solution performs 9.3 times faster in average of the total of all queries.
2. For DBLP dataset, our solution performs 2.5 times faster in average of total of all queries and it takes 262 seconds (~4 minutes) the slowest query to be executed.
3. For YAGO dataset our solution performs 5.4 times faster in average of the total of all queries and 697 seconds (~11 minutes) for the slowest query.

From the above experiment results, it is clear that the advantages of data summaries in a majority of MashQL's general background queries, plus the Oracle's SEM_MATCH flexibility to run any RDF query fast can offer to MashQL server reliability and speediness against any RDF query regardless the sizing of the candidate RDF graphs.

6.2.2.4 N-Level Objects and N-Level properties Queries Evaluation

We evaluate the n-level object and n-level properties background queries in the same way as with the general queries. For each query we fetch data from the graph' levels 2, 3, 4, 5, thus, we have created twenty queries per dataset.

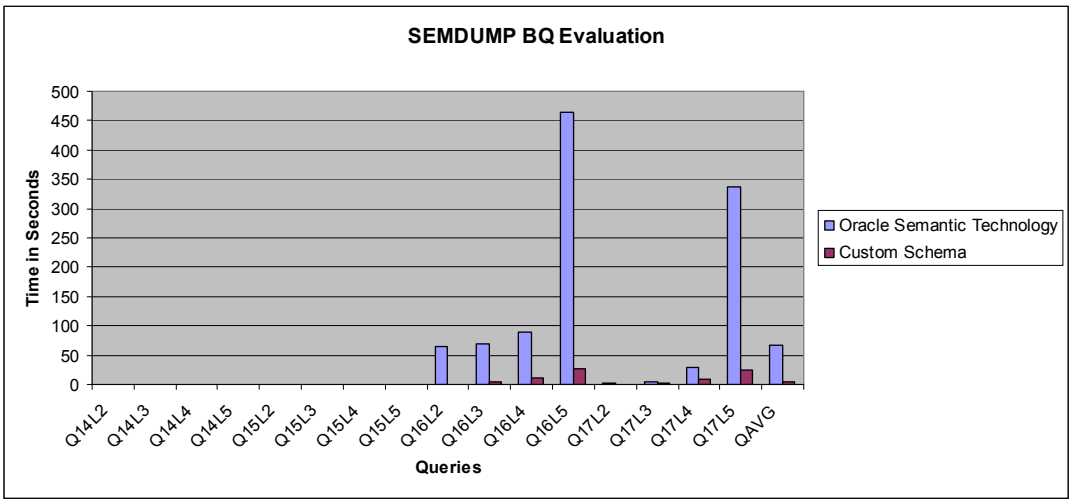


Figure 6.7: SemDump response time results for queries 14L2-5-17L2-5

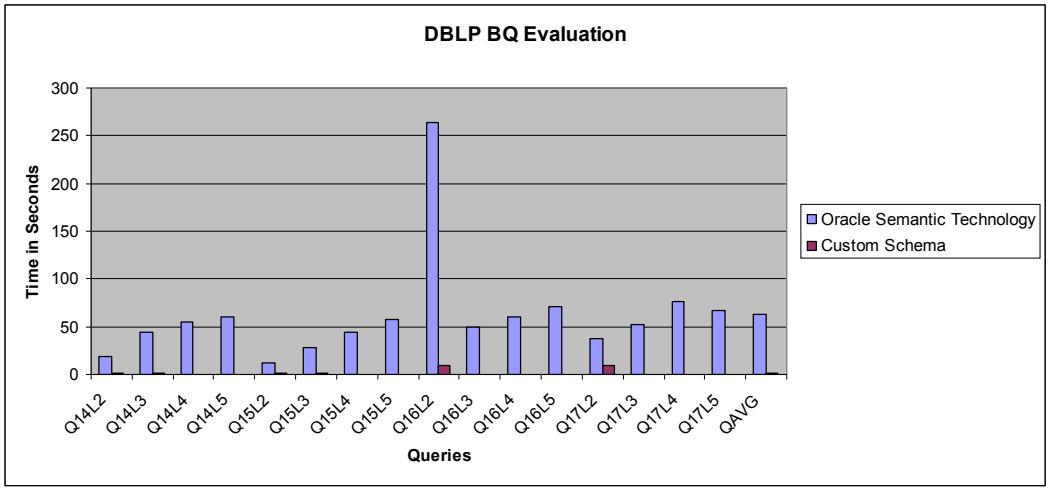


Figure 6.8: DBLP response time results for queries 14L2-5-17L2-5

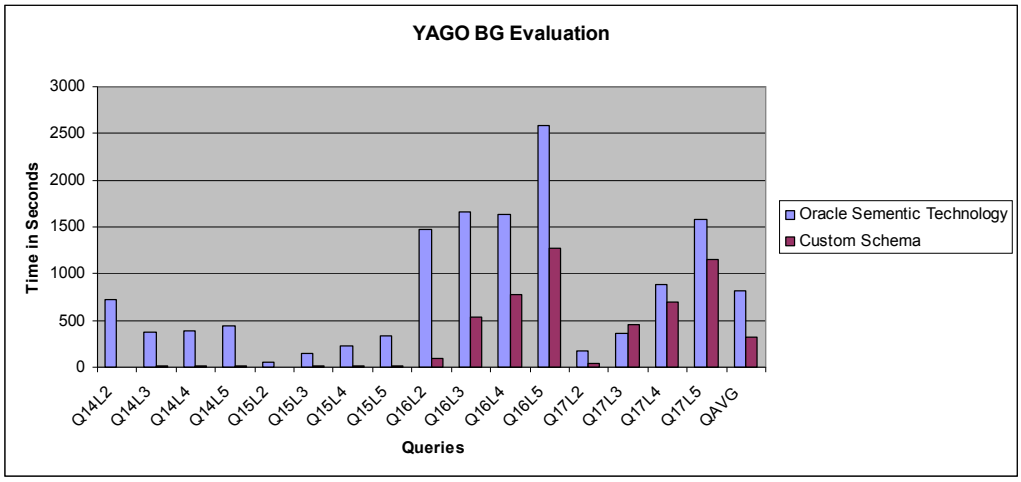


Figure 6.9: YAGO response time results for queries 14L2-5-17L2-5

6.2.2.5 Discussion for N-Level Objects and N-Level properties Queries Evaluation

From the experiments that are presented in the above charts, it is obvious that for all datasets, our optimization solution response faster than oracle's SEM_MATCH. The reasons for this performance improvement concentrated on two main factors:

1. Our optimization solution is created in order to help the execution of all the MashQL's background queries. On the other hand, Oracle's technology is designed to be able to query any RDF graph at random.
2. The BR-algorithm is able to fetch beforehand the queries' results and store these results into the database in order to bypass self-join operations during the queries' execution course. Since, the queries' results are already pre-computed and stored permanently in a database, they return very fast when they are asked for. In contrast, Oracle's SEM_MATCH will be calculated in real time. Thus, the queries' results, as a result, are delayed for longer a time.

More analytically, the main results are the following:

1. The SEMDUMP dataset performs 14 times faster in average of the total of all queries in all levels.
2. The DBLP dataset, our solution, performs 45 times faster in average of the total of all queries.

3. The YAGO dataset, our solution, performs 2.5 times faster in average of total of all queries.

It is clear, that the BR-algorithm provides high response time for all MashQL's N-Level Objects and N-Level Properties Queries because of the data being fetched beforehand. Unfortunately, the algorithm can fetch data beforehand until a graphs' constant level. All the data that can not be fetched by the BR-algorithm will be fetched using the Oracle's SEM_MATCH via the Query Optimizer.

Chapter 7

Conclusions and future work

7.1 Conclusions

In this work, we designed, implemented and evaluated two important components of the MashQL server, the RDF Loader and the Query Optimizer module.

With the RDF Loader module, we achieved to design and implement a concrete system that includes a combination of the market's lasted technologies that exist in the Extract-Transform-Load (ETL) process for RDF, such as Oracle, Java and Jena .Base on these technologies, we created a powerful, stable and intelligent RDF loader that loads any RDF data in any format and of any size in a very short time. This fact is proved from our experiment's results that became our benchmark , they consisted of different datasets in various sizes and it shows that our RDF loader is able to load a 1.5G bytes of RDF data in less than 30 minutes (80,000 triples per second), a value that is very promising according to the state-of-art RDF loaders.

Our Query Optimizer module, implements our optimization solution and it was designed to execute all the MashQL background queries very fast in real time.The rest MashQL's queries , the formulated queries, are executed using Oracle's SEM_MATCH table function. The Query Optimizer module includes the database summaries and our novel BR-algorithm .

Using the database summaries we have the advantage that, instead of scanning and sorting all the data during the query's execution the data are already sorted and pre-computed according to the general background queries requirements. With this technique, we have smaller data sets. As a results, less scanning data, we save performance time, since the data are already sorted during the summaries creation process, most of the queries' results are already pre-computed and finally, Oracle's materialized views provide faster access time on data in relation with a normal tables.

The BR-algorithm is an algorithm that explores, for each subject, all the possible paths between the examined subject in relation to the rest graph's nodes grouping their results into a

graph's levels. For each level, the algorithm stores the result (BR's entries) into BR$ table whose each level will represent the NL-O and NL-P Background Queries patterns. The idea behind the BR-Algorithm is the characteristics of the NL-O and NL-P Background Queries that, if you know the neighbour objects for each subject and for each level, you will know the results of each pattern (defined by the query) since each pattern expands the graph one level deeper. Knowing this information you need to return only to the results of the last level (the last pattern) without calculating the results of the previous levels. This technique will bypass RDBMS self-joins, the queries will be executed lightly, and, the performance will be improved.Additionally, The BR-algorithm is supported with a novel idea of RDF graph's partitioning by breaking any RDF graph into three parts. The idea behind this partition schema is the separation of triples that comprise the graph's edges without influencing the final result. In this segregation are included the triples which are comprise the graph's initial edges and the graph's final edges. This graph's segregation of nodes creates a new graph that is much smaller than the initial. We used this partition schema on our datasets, and we managed to reduce ~3 times in average the source graphs. This achievement helps us to run BR-algorithm faster (~38 times faster rather than the absent RDF graph partitioning) and to fetch beforehand less data in bigger graph's depths that used for the N-Level Objects and N-Level properties Queries.

Both data summaries and BR-algorithm have managed to overcome the problem of RDF's self-joins during the RDF queries course. They achieved to normalize the triple table in smaller objects, thus, the MashQL background queries get access to less data faster and finally they extend the Oracle's RDF technology in order to put forward the best performance execution for the MashQL background queries according to their requirements. This achievement is proved from our experiment's results that became our benchmark, and comprised was consisted of different datasets in various sizes and it shows that, the general background queries of our optimization solution performs ~10 times faster rather than the oracle's SEM_MATCH table function and also, the N-Level Objects and N-Level properties

Queries of our optimization solution performs ~45 times faster rather than the oracle's SEM_MATCH table function.

The experiment's results, concerning data summaries and BR-algorithm, were both based on our optimization solution and they are very encouraging. We believe that their good performance standards during the background queries execution course will add value to MashQL server providing to it stability and speed.

7.2 Future work

As we already mentioned, the BR-algorithm is able to fetch beforehand graph's data until we reached a constant path or Level for the MashQL's N-Level Objects and N-Level properties queries. This limitation is not considered a disadvantage, since, in this way the MashQL's N-Level Objects and N-Level properties queries have a very good response time against huge RDF graphs. On the other hand, the Oracle's SEM_MATCH even if it does not have any limitations in the query, its queries do not complete or its queries response time is very slow, when it comes to query huge RDF graphs.

Consequently, future work will be supposed as providing the possibility of increasing the BR-algorithm's maximum discovering Level. However, such possibility will increase much more the BR$'s table size and the algorithm's completion time, since the algorithm will fetch beforehand more data. Thus, in order to increase the BR-algorithm's maximum discovering Level, first we need to find a solution to how we keep constant the size of BR$'s table constant. Likely solutions are techniques to compress the table's data or to normalize the BR$'s table cutting it down to smaller objects.

Additionally, we are very optimistic that the BR-algorithm combined with graph's partitioning after a persistence scientific study can be adapted in order to query any RDF graph at random.

In addition, we believe that the RDF technology will continue growing, as a result, the RDF graphs for various data sources will continue increasing in size, causing their management to be very complicated. A likely confrontation of this problem can be found in our proposal. Our new graph's partitioning schema that we proposed makes a step toward breaking an RDF

graph into smaller sub-graphs. Breaking the RDF graphs into smaller and more manageable pieces of data, offers to any queries faster access to the data, more choices in the parallel algorithms, and better comprehension of RDF graphs' characteristics.

Our future effort will be to divide the RDF graphs in more than three parts as our partition schema offers today certain benefits (we need this in order to gain ground from the above advantages). If we achieved this objective, we would feel positive that our optimization solution would run faster, but, it is an issue that needs more scientific study.

Appendix A

Enable RDF in Oracle 11g

To enable RDF Support (Oracle Semantic Technology) you need to login as a privilege database user by running the database script @?/md/admin/catsem11i.sql . To ensure that the installation is completed successful , the database components Spatial must be valid .

```
SQL>select comp_name,status
from dba_registry
where comp_name='Spatial';
COMP_NAME     STATUS
---------              ------------
Spatial                 VALID
```

Since RDF data store tends to be very large, oracle recommends to creating a separate tablespace* for all RDF tables.

The following example creates a tablespace named RDFTBS.

```
SQL>CREATE TABLESPACE  RDFTBS
DATAFILE 'C:\APP\MGEORGIOU\ORADATA\ORCL\RDFTBS.dbf' SIZE 1024M
AUTOEXTEND ON NEXT 256M MAXSIZE UNLIMITED
SEGMENT SPACE MANAGEMENT AUTO;
```

A **tablespace** is a logical storage unit within oracle database. A tablespace space consists of at least on physical datafile e.g. C:\APP\MGEORGIOU\ORADATA\ORCL\RDFTBS.dbf . A datafile belongs to exactly on tablespace.

Creating an RDF network enables RDF store in the Oracle database. Only users with DBA privilege can create an RDF network. Create the network only once for an Oracle database instance. The following example creates an RDF network using a tablespace named RDFTBS[13]

```
SQL>EXECUTE SDO_RDF.CREATE_RDF_NETWORK('RDFTBS');
```

Create your database user (the user who is responsible to host RDF graphs) and grants all the appropriate privilege in order to be able to creates RDF tables.

```
SQL>create user rdf identified by rdf;
SQL>grant connect,resource,dba to rdf;
SQL>GRANT EXECUTE ON MDSYS.RDF_APIS_INTERNAL TO RDF;
```

The following example shows how to load in the oracle 11g database two graphs RDF graphs using conventional sql insert statements. The first graph is created by Dr. Mustafa Jarrar as an technology example for presenting MashQL editor and consider the main example in my thesis and the second graph is produces by Oracle as RDF demo example in its Semantics documentation.

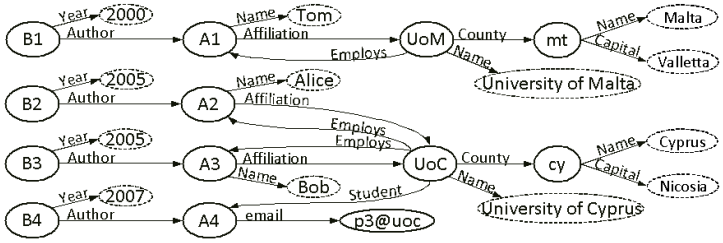


Figure A.1 : Books RDF graph [1]

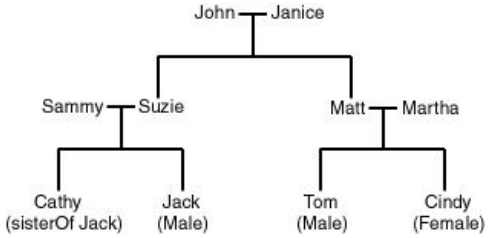


Figure A.2 : Family tree [13]

For each graph create a table to store references to the RDF data .This table must contain a column of type SDO_RDF_TRIPLE_S, which will contain references to all data associated with a single RDF model. It is recommended that this table include a column named ID of type NUMBER and a column named TRIPLE of type SDO_RDF_TRIPLE_S .

```
---for Universities graph—
SQL>CREATE TABLE UOC
(id NUMBER, triple SDO_RDF_TRIPLE_S);
create sequence id_auto_number_UOC_graph;
CREATE OR REPLACE TRIGGER UOC_TRG
BEFORE INSERT on UOC FOR EACH ROW
BEGIN
        SELECT id_auto_number_UOC_graph.nextval
        INTO :NEW.ID FROM DUAL;
END ;
/
---for family graph—
```

```
SQL>CREATE TABLE FAMILY
(id NUMBER, triple SDO_RDF_TRIPLE_S);
create sequence id_auto_number_FAMILY_graph;
CREATE OR REPLACE TRIGGER FAMILY_TRG
BEFORE INSERT on FAMILY FOR EACH ROW
BEGIN
        SELECT id_auto_number_FAMILY_graph.nextval
        INTO :NEW.ID FROM DUAL;
END ;
/
```

An RDF graph is created by specifying a model name, the table name to hold references to RDF data for the graph, and the column of type SDO_RDF_TRIPLE_S in that table. The following command creates a model named FAMILY_MODEL and UNIV_MODEL which will use the tables created in the preceding step.

```
SQL>EXECUTE SDO_RDF.CREATE_RDF_MODEL('FAMILY_MODEL', 'FAMILY', 'TRIPLE');
SQL>EXECUTE SDO_RDF.CREATE_RDF_MODEL('UNIV_MODEL', 'UOC', 'TRIPLE');
```

Appendix B

Loading RDF data into an Oracle 11g Database

Oracle supports three ways to RDF data sources into a database.

1. **Bulk load using a SQL*Loader** direct-path load to get data from an N-Triple format into a staging table and then use a PL/SQL procedure to load or append the data into the database. The bulk load insert using the SQL*Loader is consider the faster way.
2. Load into tables using **SQL INSERT** statements that call the SDO_RDF_TRIPLE_S constructor.
3. Batch load using a **Java client interface** to load or append data from an N-Triple format file into the database.

Insert statements for family graph :

```
SQL>INSERT INTO family(triple) VALUES (
SDO_RDF_TRIPLE_S('family_model',
'http://www.example.org/family/John',
'http://www.example.org/family/fatherOf',
'http://www.example.org/family/Suzie'));
```

Batch Loading Semantic Data Using the Java API

You can perform a batch load operation using the Java class oracle.spatial.rdf.client.BatchLoader. The class is running on JDK version 1.5 and Oracle 11g Database Enterprice edition. The class's class path is located under <ORACLE_HOME>/md/jlib/sdordf.jar.The class accept files only in NT format, thus you need to convert the RDF/XML to NT or N3 to NT.

Batch Loading Semantic Data Using the Java API : Example on Windows

```
java
-Ddb.user=<database_user>
-Ddb.password=<database_user_password>
-Ddb.host=<database_ip_address>
-Ddb.port=<database_tcpip_port>
-Ddb.sid=<instance_name>
-classpath %ORACLE_HOME%\md\jlib\sdordf.jar;%ORACLE_HOME%\jdbc\lib\ojdbc5.jar
oracle.spatial.rdf.
```

Batch loading is faster than loading semantic data using INSERT statements. Batch loading is typically a good option when the following conditions are true:

1. The data to be loaded is less than a few million triples.
2. The data contains significant amount long literals (longer than 4000 bytes).

Bulk Loading Using SQL*Loader

The SQL*Loader is consider the fastest way to load RDF data into a database and it is recommended by Oracle when the dataset is very huge. SQL*Loader is the main loading method that we use inside the MashQL loader.SQL*Loader can loads only NT format files, thus, we convert any RDF file (RDF/XML or N3) into NT format. The NT file is consider as the SQL*Loader input file (data).The SQL*Loader is used to loads the input files into a staging table and after the SEM_APIS.BULK_LOAD_FROM_STAGING_TABLE database procedure move the loaded data into Oracle's Semantics Technology and creates the RDF graph.

The SQL* Loader can be invoked using the sqlldr command and can has three mandatory options . The userid that you can specify the database credentials, the control file* option that you specify the destination of a control file and finaly the data option that you can specify the destination of the data file (in our case the NT file).

Control file: The SQL*Loader control file is a repository that contains the DDL instructions that you have created to control where SQL*Loader will find the data to load, how SQL*Loader expects that data to be formatted, how SQL*Loader will be configured (memory management, rejecting records, interrupted load handling, etc.) as it loads the data, and how it will manipulate the data being loaded.

SQL*Loader available options:

```
sqlldr
userid=username/password
control=
data=
direct=
skip=
load=
```

```
discardmax=
bad=
discard=
log=
errors=
```

SQL*Loader control file example :

```
UNRECOVERABLE
LOAD DATA
TRUNCATE
into table  <table_name>
when (1) <> '#'
(
RDF$STC_sub   CHAR(4000) terminated by whitespace
\(
CASE
WHEN substr(:RDF$STC_sub,1,1)='<' AND substr(:RDF$STC_sub,-1,1)='>' AND
length(:RDF$STC_sub)>2
THEN :RDF$STC_sub
WHEN substr(:RDF$STC_sub,1,2)='_:' AND
REGEXP_LIKE(:RDF$STC_sub,'^(_:)[[:alpha:]][[:alnum:]]*$')
THEN :RDF$STC_sub
WHEN substr(:RDF$STC_sub,1,1) NOT IN ('\\\','<','#') AND
substr(:RDF$STC_sub,-1,1) NOT IN ('\\\','>')
THEN ('<' || SDO_RDF.replace_rdf_prefix(:RDF$STC_sub) || '>')
WHEN substr(:RDF$STC_sub,1,1)='#'
THEN SDO_RDF.raise_parse_error(
'Ignored Comment Line starting with ', :RDF$STC_sub)
ELSE SDO_RDF.raise_parse_error('Invalid Subject', :RDF$STC_sub)
END
)\,
RDF$STC_pred  CHAR(4000) terminated by whitespace
\(
CASE
WHEN substr(:RDF$STC_pred,1,1)='<' AND substr(:RDF$STC_pred,-1,1)='>' AND
length(:RDF$STC_pred)>2
THEN :RDF$STC_pred
WHEN substr(:RDF$STC_pred,1,2) != '_:' AND
substr(:RDF$STC_pred,1,1) NOT IN ('\\\','<') AND
substr(:RDF$STC_pred,-1,1) NOT IN ('\\\','>')
THEN ('<' || SDO_RDF.replace_rdf_prefix(:RDF$STC_pred) || '>')
ELSE SDO_RDF.raise_parse_error('Invalid Predicate', :RDF$STC_pred)
END
)\,
--
-- right-trimming of WHITESPACEs is reqd for \RDF$STC_obj\
-- (due to absence of \TERMINATED BY WHITESPACE\)
--
-- For ease of editing below replace
--   \rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))\ with \:xy\.
-- and then replace back
--
RDF$STC_obj   CHAR(5000)
\(
CASE
WHEN substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,1)='<' AND
substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),-1,1)='>' AND
length(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)))>2
```

```
THEN rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))
WHEN substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,1)='\\\' AND
substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),-1,1)='\\\' AND
length(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)))>1
THEN rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))
WHEN substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,2)='_:' AND
REGEXP_LIKE(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),
'^(_:)[[:alpha:]][[:alnum:]]*$')
THEN rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))
WHEN substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,1)
NOT IN ('\\\','<') AND
substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),-1,1)
NOT IN ('\\\','>')
THEN ('<' ||
SDO_RDF.replace_rdf_prefix(
rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))) ||
'>')
WHEN substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,1)='\\\' AND
substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),-1,1)
NOT IN ('\\\','>') AND
instr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),'\\\\\@',-1)>1 AND
REGEXP_LIKE(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),
'^\\\[[:print:]]*\\\\\@[[:alpha:]](-[[:alnum:]])?$')
THEN rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13))
WHEN (substr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),1,1)='\\\' AND
instr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),'\\\^^',-1)>1 AND
(length(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)))-
(instr(rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)),'\\\^^',-1)4)
)>0)
THEN SDO_RDF.pov_typed_literal(
rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)))
ELSE SDO_RDF.raise_parse_error(
'Invalid Object', rtrim(:RDF$STC_obj,'. '||CHR(9)||CHR(10)||CHR(13)))
END
)\
);
```

Appendix C

Database summaries creation for MashQL's background queries

BQ-1 O1 :{(?S1 <:Type> ?O1)} **Description:** This query returns a set objects types (O) belong to graph G that are rdf:type.
BQ-2 S1 :{(?S1 ?P1 ?O1)} UNION O1:{(?S1 ?P1 ?O1). Filter isURI(?O1)} **Description:** This query returns all the unique subjects (S) and objects (O) filter by objects which are URL.
BQ-3 S Є V **Description**: This query returns all the subjects (S) which are equal with the variable/label V
BQ-4 P2 :{(?S1 <:Type> <S>)(?S2 ?P2 ?O2)} **Description:** This query returns all properties (P) depending on the chosen subject in previous queries.
BQ-5 P1 :{(<S> ?P1 ?O1)} **Description:** This query returns a set of properties (P) according a variable subject (<S>).
BQ-6 P1 :{(?S1 ?P1 ?O1)} **Description:** This query returns a set of properties (P) in G.
BQ-7 P Є V **Description:** This query returns all properties (P) which are equal with the variable/label V
BQ-8 O1:{(<S> ?P1 ?O1) Filter isURI(?O1)} **Description:** This query returns all objects (O), filter by URI, according a variable (<S>).
BQ-9 O1:{(<S> <P> ?O1) Filter isURI(?O1)} **Description:** This query returns all objects, filter by URI, according a variable subject (<S>) and variable property (<P>).
BQ-10 O1:{(?S1 <:Type> <S>)(?S1 ?P2 ?O2)} **Description:** This query returns all objects types (O), according the variable subject (<S>), expanded in the next path.
BQ-11 O:{(?S <:Type> <S>)(?S <P> ?O)} **Description:** This query returns all objects types (O), according the variable subject (<S>), expanded in the next path according the variable property (<P>).
BQ-12 O1:{(?S1 ?P1 ?O1)} **Description:** This query returns a set of objects (O) in G.
BQ-13 O1:{(?S1 <P> ?O1)} **Description:** This query returns a set of objects (O) in G, according the variable property (<P>)
BQ-14 Pn:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3) … (On-1 ?Pn ?On)} **Description:** This query returns properties (P) where the root predicates are rdf:type after n-level expanding where n > 1.

BQ-15 On:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)} **Description:** This query returns objects (O) where the root predicates are rdf:type after n-level expanding where n > 1 .
BQ-16 Pn:{(S1 P1 O1)(O1 P2 O2 … (On-1 ?Pn ?On)} **Description:** This query returns properties (P) after n-level expanding where n > 1.
BQ-17 On:{(S1 P1 O1)(O1 P2 O2 … (On-1 ?Pn ?On)} **Description:** This query returns objects (O) after n-level expanding where n > 1.

Table C.1: Summary of all Background Queries [1]

Materialized View Name	Description
MVN$T_S_<graph_name>	**Synopsis** An oracle's materialized views contain all unique subjects that their properties belong to the rdf:type class. **Columns** NODE number VALUE_NAME varchar2(4000) **Usage** It is included in background queries 4,10,11 **Creation Script:** create materialized view MVN$T_S_<graph_name> as select l.start_node_id as node, v.value_name from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id= (select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.start_node_id=v.value_id and l.p_value_id IN (select l.p_value_id from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.p_value_id=v.value_id and v.vname_suffix='type' group by l.p_value_id);
MVN$T_O_<graph_name>	**Synopsis** This mview contains all the unique objects (o) that belong to the rdf:type class. **Columns:** NODE number VALUE_NAME varchar2(4000) **Usage** It is included in background queries 1

	Creation Script: create materialized view MVN$T_O_<graph_name> as select l.end_node_id as node , v.value_name from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.end_node_id=v.value_id and l.p_value_id IN (select l.p_value_id from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.p_value_id=v.value_id and v.vname_suffix='type' group by l.p_value_id);
MVN$O_<graph_name>	**Synopsis** This mview contains all unique objects (o) that belong on graph G with out any restrictions **Columns:** END_NODE_ID number VALUE_NAME varchar2(4000) VALUE_TYPE varchar2(10) **Usage** It is included in background queries 4,8,9,10,11,12 **Creation Script:** create materialized view MVN$O_<graph_name> as select l.end_node_id, v.value_name, v.value_type from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.end_node_id=v.value_id group by l.end_node_id, v.value_name, v.value_type;
MVN$S_<graph_name>	**Synopsis** This mview contains all unique subject (s) that belong on graph G with out any restrictions **Columns:** START_NODE_ID number VALUE_NAME varchar2(4000) VALUE_TYPE varchar2(10) **Usage** It is included in background queries 3,5,8,9 **Creation Script** create materialized view MVN$S_<graph_name>

	as select l.start_node_id, v.value_name, v.value_type from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.start_node_id=v.value_id group by l.start_node_id, v.value_name, v.value_type;
MVN$P_<graph_name>	**Synopsis** This mview contains all unique properties (p) that belong on graph G with out any restrictions **Columns:** P_VALUE_ID number VALUE_NAME varchar2(4000) VALUE_TYPE varchar2(10) **Usage** It is included in background queries 6,7,9,11,13 **Creation Script** create materialized view MVN$P_<graph_name> as select l.p_value_id, v.value_name, v.value_type from mdsys.rdf_link$ l,mdsys.rdf_value$ v where l.model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>) and l.p_value_id=v.value_id group by l.p_value_id, v.value_name, v.value_type;
MVN$<graph_name>	**Synopsis** This Mview contains all the graphs triples s,p,o **Columns:** S number P number O number **Usage** It is included in background queries 4,5,8,9,10,11,13 **Creation Script** create materialized view MVN$<graph_name> as select start_node_id as s, p_value_id as p, end_node_id as o from mdsys.rdf_link$ where model_id=(select model_id from mdsys.RDF_MODEL_INTERNAL$ where model_name=<graph_name>);

Table C.2 : Summaries for the General background Queries

Appendix D

Evaluation results

(Actual values that are show in the charts, chapter 6)

Query	Oracle's	Our
Q1	3,5	0
Q2	42,25	0
Q3	23	0
Q4	0	0
Q5	0,5	0
Q6	1,5	0
Q7	1,5	0
Q8	0,25	0
Q9	0,25	0
Q10	9	0
Q11	5	0
Q12	32	0
Q13	2,25	0
QAVG	9,30	0

Table D.1 : SemDump response time results in seconds for queries 1-13 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Query	Oracle's	Our
Q1	30,2	58,2
Q2	400,6	6,2
Q3	332	2
Q4	21,6	232,6
Q5	8,8	2,4
Q6	35	0
Q7	38,4	0
Q8	7,6	13
Q9	0,2	6,4
Q10	282,6	261,2
Q11	12,4	20,6
Q12	309,4	4,6
Q13	86	8,2
QAVG	120,36	47,33

Table D.2: DBLP response time results in seconds for queries 1-13 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Query	Oracle's	Our
Q1	788,33	697
Q2	1394,33	22
Q3	1095,33	4
Q4	17	281
Q5	16,33	4

Q6	**45**	**0**
Q7	**53**	**0**
Q8	**16**	**4**
Q9	**0,3**	**4**
Q10	**798,66**	**109**
Q11	**6**	**13,33**
Q12	**1222,66**	**10,33**
Q13	**834,33**	**14**
QAVG	**483,64**	**89,43**

Table D.3: YAGO response time results in seconds for queries 1-13 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Query	**Oracle's**	**Our**
Q14L2	**0,5**	**0**
Q14L3	**0,5**	**0**
Q14L4	**0,75**	**0**
Q14L5	**1**	**0**
Q15L2	**0**	**0**
Q15L3	**0**	**0**
Q15L4	**0**	**0**
Q15L5	**0**	**0**
Q16L2	**64,5**	**0**
Q16L3	**69,25**	**3,5**
Q16L4	**88,5**	**11**
Q16L5	**465,25**	**26**
Q17L2	**2**	**0**
Q17L3	**4**	**2,75**
Q17L4	**29,75**	**10**
Q17L5	**336,5**	**23,5**
QAVG	**66,40**	**4,79**

Table D.3 : SemDump response time results in seconds for queries 14L2-5-17L2-5 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Query	**Oracle's**	**Our**
Q14L2	**19,2**	**1**
Q14L3	**44**	**0,8**
Q14L4	**55,4**	**0**
Q14L5	**60,6**	**0**
Q15L2	**12**	**1**
Q15L3	**28,6**	**0,8**
Q15L4	**43,6**	**0**
Q15L5	**57,8**	**0**
Q16L2	**264,2**	**9,4**
Q16L3	**49,4**	**0**
Q16L4	**60,8**	**0**
Q16L5	**70,8**	**0**
Q17L2	**37,6**	**9,2**
Q17L3	**52,8**	**0**
Q17L4	**76,2**	**0**
Q17L5	**67**	**0**
QAVG	**62,5**	**1,3875**

Table D.4 : DBLP response time results in seconds for queries 14L2-5-17L2-5 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Query	Oracle's	Our
Q14L2	**717,66**	**4**
Q14L3	**369,66**	**7,33**
Q14L4	**387,33**	**9**
Q14L5	**441,66**	**10**
Q15L2	**60**	**3,66**
Q15L3	**143,33**	**7**
Q15L4	**228**	**8**
Q15L5	**329,33**	**9,33**
Q16L2	**1473,66**	**92,66**
Q16L3	**1657**	**535,33**
Q16L4	**1637,66**	**771,66**
Q16L5	**2586,66**	**1271**
Q17L2	**176,66**	**33,66**
Q17L3	**365**	**452,66**
Q17L4	**880**	**702,66**
Q17L5	**1583**	**1146,33**
QAVG	**814,79**	**316,52**

Table D.5: YAGO response time results in seconds for queries 14L2-5-17L2-5 (the 0 value on the column "our" indicate that query is completed in a few milliseconds)

Appendix E

All background queries for Oracle's SEM_MATCH

compared with our optimization solution

(for YAGO Dataset)

Oracle's SEM_MATCH

```
set timing on
set heading off
set echo off
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q1 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT o FROM TABLE
(SEM_MATCH('
(?s rdf:type ?o)',
SEM_Models('YAGO'), null, null, null))
group by o
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q2 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT s FROM TABLE
(SEM_MATCH('
(?s ?p ?o)',
SEM_Models('YAGO'), null, null, null))
UNION
SELECT o FROM TABLE
(SEM_MATCH('
(?s ?p ?o)',
SEM_Models('YAGO'), null, null, null))
where o$rdfvtyp='URI'
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q3 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------

alter system flush buffer_cache;
```

```
select count(*) from (
SELECT s FROM TABLE
(SEM_MATCH('
(?s ?p ?o)',
SEM_Models('YAGO'), null, null, null))
where s like '%Michael%'
group by s
);
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "Oracle Q4 Started ..."
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT  P FROM TABLE
(SEM_MATCH('
(?S rdf:type <wikicategory_American_film_actors>)(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by P
);
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "Oracle Q5 Started ..."
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT  P FROM TABLE
(SEM_MATCH('
(<William_Penn_Patrick> ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by P
);

--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "Oracle Q6 Started ..."
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
SELECT  P FROM TABLE
(SEM_MATCH('
(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by p
);

--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "Oracle Q7 Started ..."
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
```

```
alter system flush buffer_cache;

select count(*) from (
SELECT  P FROM TABLE
(SEM_MATCH('
(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
where p like '%type%'
group by p
);

--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q8 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT o
FROM TABLE(SEM_MATCH(
'(<William_Penn_Patrick> ?p ?o)
',
SEM_Models('YAGO'),
null,
null,
null))
where o$rdfvtyp='URI'
group by o
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q9 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
SELECT o
FROM TABLE(SEM_MATCH(
'(<The_Dismemberment_Plan> <y:created> ?o)
',
SEM_Models('YAGO'),
null,
null,
null))
where o$rdfvtyp='URI'
group by o
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q10 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
SELECT  O FROM TABLE
```

```
(SEM_MATCH('
(?S rdf:type <wikicategory_American_film_actors>)(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by O
);

--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q11 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
SELECT  O FROM TABLE
(SEM_MATCH('
(?S rdf:type <wikicategory_Spy_novels>)(?S <http://www.w3.org/1999/02/22-rdf-syntax-
ns#type> ?O)',
SEM_Models('YAGO'), null, null, null))
group by O
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q12 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
SELECT  O FROM TABLE
(SEM_MATCH('
(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by o
);

--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q13 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
SELECT  O FROM TABLE
(SEM_MATCH('
(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
 where p like '%type%'
group by o
);
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
Prompt "Oracle Q14 Started ..."
--------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------
---(14') On:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)}
--------------------------------------------------------------------------------------------------
/*
```

```
Prompt "Oracle Q14 L1 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT o FROM TABLE
(SEM_MATCH(
'(?S rdf:type ?O)',
SEM_Models('YAGO'), null, null, null))
group by o
);

*/
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q14 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  o1 FROM TABLE
(SEM_MATCH('
(?S rdf:type ?O)(?O ?P1 ?O1)',
SEM_Models('YAGO'), null, null, null))
group by o1
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q14 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  o2 FROM TABLE
(SEM_MATCH('
(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)',
SEM_Models('YAGO'), null, null, null))
group by o2
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q14 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT o3 FROM TABLE
(SEM_MATCH(
'(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)',
SEM_Models('YAGO'), null, null, null))
group by o3
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q14 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT o4 FROM TABLE
(SEM_MATCH(
'(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)(?O3 ?P4 ?O4)',
SEM_Models('YAGO'), null, null, null))
group by o4
);
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q15 Started ..."
```

```
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
------(15') Pn:{(?S1 <Type> O)(?S1 P2 O2)(O2 P3 O3) … (On-1 ?Pn ?On)}
-------------------------------------------------------------------------------------------------------
Prompt "Oracle Q15 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  p1 FROM TABLE
(SEM_MATCH('
(?S rdf:type ?O)(?O ?P1 ?O1)',
SEM_Models('YAGO'), null, null, null))
group by p1
);
-------------------------------------------------------------------------------------------------------
Prompt "Oracle Q15 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  p2 FROM TABLE
(SEM_MATCH('
(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)',
SEM_Models('YAGO'), null, null, null))
group by p2
);
-------------------------------------------------------------------------------------------------------
Prompt "Oracle Q15 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT p3 FROM TABLE
(SEM_MATCH(
'(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)',
SEM_Models('YAGO'), null, null, null))
group by p3
);
-------------------------------------------------------------------------------------------------------
Prompt "Oracle Q15 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT p4 FROM TABLE
(SEM_MATCH(
'(?S rdf:type ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)(?O3 ?P4 ?O4)',
SEM_Models('YAGO'), null, null, null))
group by p4
);
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
Prompt "Oracle Q16 Started ..."
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
--(16') On:{(S1 P1 O1)(O1 P2 O2)(O2 P3 O3)…(On-1 Pn ?On)}
-------------------------------------------------------------------------------------------------------
/*
Prompt "Oracle Q16 L1 Started ..."
alter system flush buffer_cache;
```

```
select count(*) from (
SELECT o FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by o
);
*/
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q16 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  o1 FROM TABLE
(SEM_MATCH('
(?S ?P ?O)(?O ?P1 ?O1)',
SEM_Models('YAGO'), null, null, null))
group by o1
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q16 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  o2 FROM TABLE
(SEM_MATCH('
(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)',
SEM_Models('YAGO'), null, null, null))
group by o2
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q16 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT o3 FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)',
SEM_Models('YAGO'), null, null, null))
group by o3
);
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q16 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT o4 FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)(?O3 ?P4 ?O4)',
SEM_Models('YAGO'), null, null, null))
group by o4
);
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "Oracle Q17 Started ..."
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
---(17') Pn:{(S1 P1 O1)(O1 P2 O2 … (On-1 ?Pn ?On)}
-----------------------------------------------------------------------------------------------------
```

```
/*
Prompt "Oracle Q17 L1 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT p FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)',
SEM_Models('YAGO'), null, null, null))
group by p
);
*/
-----------------------------------------------------------------------------------------------
Prompt "Oracle Q17 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  p1 FROM TABLE
(SEM_MATCH('
(?S ?P ?O)(?O ?P1 ?O1)',
SEM_Models('YAGO'), null, null, null))
group by p1
);
-----------------------------------------------------------------------------------------------
Prompt "Oracle Q17 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT  p2 FROM TABLE
(SEM_MATCH('
(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)',
SEM_Models('YAGO'), null, null, null))
group by p2
);
-----------------------------------------------------------------------------------------------
Prompt "Oracle Q17 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT p3 FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)',
SEM_Models('YAGO'), null, null, null))
group by p3
);
-----------------------------------------------------------------------------------------------
Prompt "Oracle Q17 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
SELECT p4 FROM TABLE
(SEM_MATCH(
'(?S ?P ?O)(?O ?P1 ?O1)(?O1 ?P2 ?O2)(?O2 ?P3 ?O3)(?O3 ?P4 ?O4)',
SEM_Models('YAGO'), null, null, null))
group by p4
);
-----------------------------------------------------------------------------------------------
spool off
```

Our Optimization Solution

```
set timing on
set heading off
set echo off
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
Prompt "BR Q1 Started ..."
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getobject(node) from mvn$t_o_yago group by getobject(node)
);

-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
Prompt "BR Q2 Started ..."
-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getobject(start_node_id) from (
select start_node_id
from mvn$S_YAGO
union
select end_node_id
from mvn$O_YAGO
where VALUE_TYPE='UR'
)
);

-----------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------
Prompt "BR Q3 Started ..."
```

```
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
select getobject(start_node_id) from (
select start_node_id from mvn$s_YAGO where value_name like '%Michael%'
));
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "BR Q4 Started ..."
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------

alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select L1.p as p
    from (select s,p from mvn$yago ) L1
                        ,
                                (select s from mvn$yago
                            where s IN ( select node from mvn$t_s_yago)
                                    and o in ( select end_node_id from mvn$o_yago where
value_name='wikicategory_American_film_actors')
                                ) L2
                            where L1.s=L2.s
group by L1.p));
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "BR Q5 Started ..."
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getobject(p) from (
select p from mvn$yago
where s in ( select start_node_id from mvn$s_yago where value_name='William_Penn_Patrick')
group by p
));

-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "BR Q6 Started ..."
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------

alter system flush buffer_cache;
select count(*) from (
select value_name from mvn$P_YAGO group by value_name
);
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
Prompt "BR Q7 Started ..."
-----------------------------------------------------------------------------------------------------
-----------------------------------------------------------------------------------------------------
alter system flush buffer_cache;
```

```
select count(*) from (
select value_name from mvn$P_YAGO where value_name like '%type%'
);

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q8 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from mvn$yago
where o IN (select end_node_id from mvn$O_YAGO where VALUE_TYPE='UR')
and s in ( select start_node_id from mvn$s_yago where value_name='William_Penn_Patrick')
group by o
));
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q9 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from mvn$yago where o IN (select end_node_id from mvn$O_YAGO where
VALUE_TYPE='UR')
and s in ( select start_node_id from mvn$s_yago where value_name='The_Dismemberment_Plan')
and p in ( select p_value_id from mvn$p_yago where value_name='y:created')
group by o
));

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q10 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select L1.o as o
    from (select s,o from mvn$yago ) L1
                        ,
                                (select s from mvn$yago
                            where s IN ( select node from mvn$t_s_yago)
                                    and o in ( select end_node_id from mvn$o_yago where
value_name='wikicategory_American_film_actors')
                                ) L2
                            where L1.s=L2.s
group by L1.o));

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q11 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
```

```
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select L1.o as o
   from (select s,o from mvn$yago where p in ( select p_value_id from mvn$p_yago where
value_name='http://www.w3.org/1999/02/22-rdf-syntax-ns#type')) L1
                        ,
                                (select s from mvn$yago
                             where s IN ( select node from mvn$t_s_yago)
                                      and o in ( select end_node_id from mvn$o_yago where
value_name='wikicategory_Spy_novels')
                                ) L2
                        where L1.s=L2.s
group by L1.o));


-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q12 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
alter system flush buffer_cache;

select count(*) from (
select value_name from mvn$o_YAGO group by value_name
);


-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q13 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
 alter system flush buffer_cache;

 select count(*) from (
 select getObject(o) from (
 select o from mvn$yago
 where p in ( select p_value_id from mvn$p_yago where value_name like '%type%')
 group by o
 ));

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q14 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------


-------------------------------------------------------------------------------------------------
Prompt "BR Q14 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from br2_so_yago
where s IN
(
   select node from mvn$t_o_yago
)
```

```
group by o
));
---------------------------------------------------------------------------------------------------
Prompt "BR Q14 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from  br3_so_yago
where s IN
(
   select o from br2_so_yago
    where s IN
    (
       select node from mvn$t_o_yago
    )
    group by o
)
group by o
));
---------------------------------------------------------------------------------------------------
Prompt "BR Q14 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from  br4_so_yago
where s IN
(
select o from br3_so_yago
where s IN
(
        select o from br2_so_yago
        where s IN
        (
           select node from mvn$t_o_yago
        )
        group by o
)
)
group by o
));
---------------------------------------------------------------------------------------------------
Prompt "BR Q14 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from  br5_so_yago
where s IN
(
select o from br4_so_yago
where s IN
(
select o from br3_so_yago
where s IN
(
           select o from br2_so_yago
           where s IN
            (
```

```
            select node from mvn$t_o_yago
          )
          group by o
)
)
)
group by o
));
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "BR Q15 Started ..."
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
Prompt "BR Q15 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from br2_sp_yago
where s IN
(
    select node from mvn$t_o_yago
)
group by p
));
--------------------------------------------------------------------------------------------------------
Prompt "BR Q15 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from  br3_sp_yago
where s IN
(
   select o from br2_so_yago
    where s IN
    (
       select node from mvn$t_o_yago
    )
    group by o
)
group by p
));
--------------------------------------------------------------------------------------------------------
Prompt "BR Q15 L4 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from  br4_sp_yago
where s IN
(
select o from br3_so_yago
where s IN
(
        select o from br2_so_yago
        where s IN
        (
          select node from mvn$t_o_yago
```

```
        )
       group by o
)
)
group by p
));
-------------------------------------------------------------------------------------------------------
Prompt "BR Q15 L5 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from  br5_sp_yago
where s IN
(
select o from br4_so_yago
where s IN
(
select o from br3_so_yago
where s IN
(
           select o from br2_so_yago
           where s IN
            (
              select node from mvn$t_o_yago
            )
            group by o
)
)
)
group by p
));
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
Prompt "BR Q16 Started ..."
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------------
Prompt "BR Q16 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from br$_yago where br=2
union
select o
from mvn$yago
where s IN
(select s from fat_nodes$yago)
group by o
));
-------------------------------------------------------------------------------------------------------
Prompt "BR Q16 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(o) from (
select o from br$_yago where br=3
union
select o from br$_yago where 1=1
```

```
      and node IN
      ( select s from mvn$yago
        where 1=1
        and s IN
                        (select s from fat_nodes$yago)
     )
      and br between 2 and 3
group by o
));
-------------------------------------------------------------------------------------------------
Prompt "BR Q16 L4 Started ..."
alter system flush buffer_cache;

 select count(*) from (
 select getObject(o) from (
 select o from br$_yago where br=4
 union
 select o from br$_yago  where 1=1
       and node IN
       ( select s from mvn$yago
         where 1=1
         and s IN
                        (select s from fat_nodes$yago)
      )
      and br between 3 and 4
group by o
));

-------------------------------------------------------------------------------------------------
Prompt "BR Q16 L5 Started ..."
alter system flush buffer_cache;

 select count(*) from (
 select getObject(o) from (
 select o from br$_yago where br=5
 union
 select o from br$_yago  where 1=1
       and node IN
       ( select s from mvn$yago
         where 1=1
         and s IN
                        (select s from fat_nodes$yago)
      )
      and br between 4 and 5
group by o
));

-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
Prompt "BR Q17 Started ..."
-------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
/*
Prompt "BR Q17 L1 Started ..."
alter system flush buffer_cache;

select count(*) from (
select value_name from mvn$p_yago
);
```

```
*/
-----------------------------------------------------------------------------------------------------
Prompt "BR Q17 L2 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from br$_yago where br=2
union
select p
from mvn$yago
where s IN
(select s from fat_nodes$yago)
group by p
));
-----------------------------------------------------------------------------------------------------
Prompt "BR Q17 L3 Started ..."
alter system flush buffer_cache;

select count(*) from (
select getObject(p) from (
select p from br$_yago where br=3
union
select p from br$_yago where 1=1
     and node IN
     ( select s from mvn$yago
       where 1=1
       and s IN
                    (select s from fat_nodes$yago)
    )
     and br between 2 and 3
group by p
));
-----------------------------------------------------------------------------------------------------
Prompt "BR Q17 L4 Started ..."
alter system flush buffer_cache;

 select count(*) from (
 select getObject(p) from (
 select p from br$_yago where br=4
 union
 select p from br$_yago  where 1=1
      and node IN
      ( select s from mvn$yago
        where 1=1
        and s IN
                    (select s from fat_nodes$yago)
     )
     and br between 3 and 4
group by p
));


Prompt "BR Q17 L5 Started ..."
alter system flush buffer_cache;

 select count(*) from (
 select getObject(p) from (
 select p from br$_yago where br=5
 union
```

```
 select p from br$_yago  where 1=1
       and node IN
       ( select s from mvn$yago
         where 1=1
         and s IN
                        (select s from fat_nodes$yago)
      )
      and br between 4 and 5
group by p
));
--------------------------------------------------------------------------------------------------------
--------------------------------------------------------------------------------------------------------
spool off
```

References

[1] M. Jarrar, M. D. Dikaiakos, A Query Formulation Language for the Data Web, IEEE 2009.

[2] M. Jarrar, M. D. Dikaiakos, A Data Mashup Language for the Data Web. Proceedings of LDOW, at WWW'09, ACM, 2009.

[3] M. Jarrar, M. D. Dikaiakos, MashQL: A Query-by-Diagram Language Specification, Implementation, and Use Cases,2008

[4] M. Jarrar, M. D. Dikaiakos, MashQL: A Query-by-Diagram Topping SPARQL Towards Semantic Data Mashups, LDOW2009, April 20, 2009, Madrid, Spain.

[5] M. Jarrar, M. D. Dikaiakos , MashQL: A Query-by-Diagram Language Towards Semantic Data Mashups,2009

[6] E. I. Chong,S. Das,G. Eadon,J. Srinivasan, An Efficient SQL-based RDF Querying Scheme, Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

[7] M.Cai ,M.Frank ,RDFPeers: A Scalable Distributed RDF Repository based on A Structured PeertoPeer Network,WWW2004, May 17–22, 2004, New York, New York, USA.ACM 158113844X/04/0005

[8] L. Ma, Z. Su, Y. Pan, L. Zhang, T. Liu, RStar: An RDF Storage and Query System for Enterprise Resource Management, CIKM'04, November 8-13, 2004, Washington D.C., U.S.A.Copyright 2004 ACM 1-58113-874-1/04/0011

[9] N. Bassiliades, I. Vlahavas, R-DEVICE: A Deductive RDF Rule Language,2004

[10] W3C, *RDF Primer*, http://www.w3.org/TR/rdf-primer/, as of 2010.

[11] W3C, *Semantic Web,* http://www.w3.org/2001/sw/, as of 2010.

[12] W3C, *Semantic Web* ,http://www.w3.org/TR/rdf-sparql-protocol/, as of 2010

[13] Oracle Corporation, Semantic Technologies Developer's Guide 11g Release 1, Edition 2009

[14] S. Peenikal, Mashups and the Enterprise Mashups,Sept 2009

[15] Jena, Jena – A Semantic Web Framework for Java, http://jena.sourceforge.net/, as of October 2010

[16] Oracle Corporation , Oracle Materialized Views & Query Rewrite , http://www.oracle.com/technetwork/database/features/bi-datawarehousing/twp-bi-dw-materialized-views-10gr2--131622.pdf, May 2005

[17] Oracle Corporation , Oracle® Database SQL Reference 10g Release 2 (10.2) , Part Number B14200-02 ,chapter 9 ,Hierarchical Queries, December 2005

[18] Oracle Corporation, Oracle® Database SQL Language Reference
11g Release 1 (11.1) , Part Number B28286-06 ,chapter 16 ,Create Table, August 2010

[19] Oracle Corporation, http://www.oracle.com/technetwork/developer-tools/sql-developer/what-is-sqldev-093866.html , as of October 2010

[20] Y.Yan, C.Wang, A.Zhou, W.Qian, L.Ma,Y.Pan , Efficiently Querying RDF Data in Triple Stores, April 21-25, 2008 Beijing, China

[21] Sesami , http://www.openrdf.org, as of October 2010

[22] D.Damljanovic,J.Petrak,H.Cunningham ,Random Indexing for Searching Large RDF Graphs, EU-funded LarKC (FP7-215535) project.

[23] O. Udrea,A. Pugliese,V.S. Subrahmanian, GRIN: A Graph Based RDF Index, copyright Association for the Advancement of Artificial Intelligence (www.aaai.org), 2007

[24] Neumann T, Weikum G: RDF3X: RISC style engine for RDF. VLDB'08

[25] Abadi D, Marcus A, Madden S, Hollenbach K , Scalable semantic web
data management using vertical partitioning. VLDB, 2007.

[26] Oracle New England Development Center , The Semantic Web for Application Developers, 2007

[27] Lehigh University Benchmark (LUBM), http://swat.cse.lehigh.edu/projects/lubm , as of October 2010

[28] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein , Introduction to Algorithms, Third Edition , Chapter 25: All-Pairs Shortest Paths, September 2009, ISBN-10:0-262-03384-4 ,ISBN-13:978-0-262-03384-8

[29] Renzo A, Claudio G, and Jonathan H, RDF Query Languages Need Support for Graph Properties , 2003

[30] Ruoming J, Hui H,Haixun W,Ning R, Yang X , Computing Label-Constraint Reachability in Graph Databases, SIGMOD'10, June 6–11, 2010, Indianapolis, Indiana, USA.