POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione

TOWARDS A TOP-K SPARQL QUERY BENCHMARK GENERATOR

Relatore: Prof. Emanuele Della Valle
Correlatore: Daniele Dell'Aglio

Tesi di Laurea di:
Shima Zahmatkesh, matricola 770469

Anno Accademico 2012-2013

Acknowledgements

I am grateful for all the support I received during this thesis. I would like to thank my thesis advisor, Prof. Emanuele Della Valle, for his enthusiastic support, encouragement, and guidance through every step of this thesis, and for all he has taught me.

My gratitude to Daniele Dell'Aglio for his time, technical assistance, and the discussion we had. Special thanks to Prof. Alessandro Bozzon for discussion we had which helped me to have a better view of the target of thesis, and thanks to Sara Magliacane for her kindly support. My gratitude also to all my friends in Milan, in particular Hoda Maraghei, and Narges Shahmandi.

Special thanks to my family, particularly my parents and sister, thank them for their love, support, and resolute belief in me. Without them, I would not be the person I am today. The most special thanks goes to my best partner and friend, my husband who gave me his unconditional support and love through all this long process.

Abstract

Top-k queries which returning the top k results ordered according to a user-defined scoring function are gaining more and more attention in the Database and Semantic Web communities. Order is an important property that can be exploited to speed up query processing. Also, state-of-the-art SPARQL engines typically do no exploit order for query optimization purposes: Top-k queries are mostly managed with a materialize-then-sort processing scheme that computes all the matching solutions (e.g. thousands) even if only a limited number k (e.g. ten) are requested.

Recent works have shown that an efficient split-and-interleave processing scheme could be adopted to improve the performance of top-k SPARQL queries. A consistent comparison of those works does not exist in current literature. As often occurs, the main cause for this fragmentation resides in the lack of a SPARQL benchmark covering top-k SPARQL queries. To foster the work on top-k query processing within the Semantic Web community, we believe that it is the right time to define a top-k SPARQL benchmark.

For the benchmark to be meaningful, at least two requirements should hold: 1) the benchmark should resemble reality as much as possible, and 2) it should stress the features that distinguish top-k queries from traditional SPARQL queries both from a syntactic perspective, i.e., queries should contain rank-related clauses, and from a performance perspective, i.e. the query mix should insist on different characteristics of the queried data, which are central to top-k query answering, so to stress the evaluated systems in several running conditions. Recent works on SPARQL query benchmarks help satisfying the first requirement.

In this thesis, I investigate the second requirement, by extending DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] with the capabilities required to compare SPARQL engines on top-k queries. The Top-k DBpedia SPARQL Benchmark (Top-k DBPSB) proposed in this thesis uses the same dataset, performance metrics, and test driver of DBPSB. The innovative part of my work consists in an algorithm to create the Top-k queries from the Auxiliary queries of the DBPSB and its datasets. To generate top-k queries, my algorithm needs to have rankable variable that can be used in the scoring function part of the Top-k queries and some statistical information about the dataset. To this end, I developed an approach that explores the dataset collecting meaningful rankable variables and statistics. Once all the required information is available, it is possible to generate top-k queries for each DBPSB Auxiliary queries. My algorithm randomly generates the scoring function and adds ORDERBY and LIMIT clause to complete the Top-k queries. We have four research hypothesis: 1) the more rankable objects, the more execution time, 2) the more rankable predicates, the more execution time, 3) the more selectivity, the less execution time, and 4) the more the limit value, the same execution time (except ARQ Rank).We run the Top-K DBPSB against different four SPARQL Engines. The results of the extensive experimental evaluation confirm hypothesis 2) and 4) but do not confute the other two assumptions 1) and 3), which require further research.

Riassunto

Le query top-k query che ritornano i k migliori risultati ordinati secondo una funzionedefinita dall'utente stanno ottenendo sempre più attenzione nelle comunità delle basi di dati e del SemanticWeb. L'ordine è una proprietà importante che può essere sfruttata per accelerare l'elaborazione delle query, ma, allo stato dell'arte, i motori SPARQL in genere non usano l'ordine allo scopo di ottimizzarel'elaborazione dellequery: le query top-k sono per lo più elaborateprima materializzando i risultati e poi ordinandoli (materialize-then-sort). In questo modo il motore SPARQL prima calcola tutte le soluzioni corrispondenti (ad esempio migliaia) anche se ne sono richieste solo un numero k limitato (ad esempio dieci).

Lavori recenti hanno proposto un approccio differente alla valutazione di query top-k:split-and-interleave. Questo metodo prima divide la funzione di ordinamento in parti e poi fa in modo di valutarla via via che vengono trovati i risultati e non solo alla fine.Questo metodo si è dimostrato in grado di migliorare le prestazioni delle query top- k sia nei database che nel Semantic Web. Nella letteratura corrente non esiste, però, un lavoro di valutazione comparativa di motori SPARQL per quel che riguarda le query top-k. Come spesso accade, la causa principale di questa frammentazione risiede nella mancanza di un benchmark. Per favorire lo sviluppo della ricerca sull'ottimizzazione diquery top-k in SPARQL, crediamo che sia giunto il momento di definire un benchmarkdi querySPARQL top- k.

Un benchmark per essere significativo, dovrebbero avere almeno due requisiti: 1) dovrebbe essere simile, per quanto possibile, alla realtà e 2) dovrebbe insistere sulle caratteristiche che distinguono le query top- k dallequery SPARQL tradizionali sia dal punto di vista sintattico, vale a dire, le query devono contenere clausole di ordinamento, sia dal punto di vista delle prestazioni, ovvero il mix di query dovrebbe stressare i motori SPARQL là dove sono più deboli nel valutare le query top-k.

In questa tesi ho investigato la seconda condizione, estendendo il DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] con le capacità necessarie per confrontare motori SPARQL su query top-k. Il benchmark da me proposto usa lo stesso insieme di dati, gli stessi indicatori di prestazione, e gli stessi test driver di DBPSB. La parte innovativa del mio lavoro di tesi consiste in un algoritmo per creare query top-k a partire dalle query ausiliarie di DBPSB. Per generare le query top-k il mio algoritmo ha bisogno di conoscere un insieme di variabili rispetto a cui ordinare (rankable variabile) i risultati delle query query top-k e di alcune informazioni statistiche sull'insieme di dati. A questo scopo ho sviluppato un approccio che esplora la collezione di dati collezionando sia le statistiche che le rankable variabile. Una volta che l'informazione necessaria a generare le query top-k è disponibile, per ogni query ausiliaria di DBPSB il mio algoritmo genera in modo casuale la funzione di ordinamento e aggiunge alla query le clausole ORDER BY e LIMIT. L'algoritmo assume che: 1) maggiore sia il numero di oggetti diversi da ordinare, maggiore sia il tempo di esecuzione; 2) maggiori siano il numero di rankable variable, maggiore sia il tempo di esecuzione; 3) più alta sia la selettività dei predicati rispetto a cui si ordina, minore sia il tempo di esecuzione; e 4) la clausola limit sia ininfluente. Ho comparato con il benchmark proposto quattro motori SPARQL. La valutazione sperimentale conferma le ipotesi 2) e 4), ma non permette di dare una risposta definitiva rispetto alle ipotesi 1) e 3).

Contents

List of Figures

List of Tables

Chapter 1

Introductione

As the volume of the information on the World Wide Web increase, the need to process the data on the web automatically become an important issues. Semantic Web introduced to solve this problem.

Semantic Web is an evolution of the World Wide Web, which aims to enreaching information by adding a new layer of meta data. Automatic integration of the data and inferring new knowledge are possible through the Semantic Web.

There are several technologies related to the Semantic Web world such as Resource Description Framework (RDF) which give us a conceptual model of the information, and SPARQL query language to query RDF data sources in the Semantic Web.

The Resource Description Framework (RDF) is a standard framework proposed by W3C which is used for data interchange on the web. It is used as a general method for conceptual description of data related to web resources. It is based on creating statement about web resources in the form of subject- predicate-object expression which is known as a triple.

The SPARQL Protocol and RDF Query Language (SPARQL) is the standard query language for RDF, able to retrieve and manipulate data stored in Resource Description Framework format. In January 2008 it became an official W3C Recommendation [Prudhommeaux and Seaborne, 2008].

Top-k queries aim at retrieving only a limited number of the best query results ordered by a given ranking function. A representative example of a Top-k query is the typical user interaction with a search engine. The search engine returns thousands of results, but the user is interested only in the top k results, ranked according to their relevance. Top-k queries are gaining

```
PREFIX ex: <http://example.org>
SELECT ?restaurant ?location ?stars
WHERE {
 ?restaurant rdf:type ex:Restaurant .
 ?restaurant ex:hasLocation ?location .
 ?restaurant ex:hasRating ?stars
}
ORDER BY ?stars
LIMIT 10
```

Listing 1.1: Top-k SPARQL query example

more and more attention in the Database and Semantic Web communities.

In order to avoid the evaluation of a ranking function on the whole result set produced by a query, different optimization procedures showed in recent works [Magliacane et al., 2012; Wagner et al., 2012; Cheng et al., 2010; Cedeno, 2010], to improve the performance of top-k SPARQL queries. Such optimizations aim at ranking results incrementally during query execution, so instead of computing and ordering all the possible results only the necessary solutions be retrieved and evaluated. Top-k query can be achieved in SPARQL by adding an ORDER and LIMIT clause.

1.1 Problem Statement

Although Recent works tried to improve the performance of the Top-k queries, a consistent comparison of those works is not possible, as each of them relied on a custom data set and query mix for experimental evaluation. The main cause for this fragmentation resides in the lack of a SPARQL bench-mark covering top-k SPARQL queries. To foster the work on top-k query processing, we believe that it is right time for the Semantic Web community to define a top-k SPARQL benchmark.

For the benchmark to be meaningful, at least two requirements should hold: 1) the benchmark should resemble reality as much as possible, and 2) it should stress the features that distinguish top-k queries from tradition SPARQL queries, both from a syntactic perspective, i.e., queries should contain rank-related clauses, and from a performance perspective, i.e. the query mix should insist on different characteristics of the queried data so to stress the evaluated systems in several running conditions.

Recent works on SPARQL query benchmarks satisfied the first requirement; for instance, the DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] introduces a procedure for benchmark creation based on query-log mining, clustering and SPARQL feature analysis. To fulfill the second requirement, we decided to extend DBPSB with the capabilities required to compare SPARQL engines on top-k queries.

1.2 Original Contribution

The main contribution of this thesis is an automated method to create a benchmark for Top-k SPARQL queries. To achieve this benchmark we extend the DBpedia SPARQL Benchmark (DBPSB) by generating Top-k queries from the existing queries of the DBPSB. To the best of our knowledge, this is the only existing automated method considering the Top-k SPARQL queries in benchmarking. In details, we propose a Top-k SPARQL query benchmark that: In details, we propose a Top-k SPARQL query benchmark that:

- Extend the DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] benchmark which is based on query log mining, clustering and SPARQL feature analysis, for Top-k SPARQL query.
- Introduce an automated method to generate Top-k SPARQL query from SPARQL query with the help of statistic information of the query. Our generating method consist of finding rankable variable, statistic computation, generating scoring function, and generating new Top-k queries.
- Provide theoretical foundation in other to gather statistical information about query such as selectivity, and distribution of values, and in the next step creating the scoring function and generating new Top-k query.
- Includes experimental evaluation for Top-k SPARQL query benchmark based on the test driver of the DBPSB in order to evaluate our research hypotheses.

1.3 Outline of the Thesis

This thesis is organized as follows:

- Chapter 2 introduces the background concepts of the presented work, such as the Semantic Web, the SPARQL Query Language and top-k queries, providing the foundations for further discussion.

- Chapter 3 present Top-k query in relational data bases and SPARQL
- Chapter 4 describes the SPARQL Benchmarking by introducing the existing benchmarks.
- Chapter 5 introduces the research question, hypothesis, challenges, and success criteria.
- Chapter 6 presents our Top-k SPARQL query generating method consist of finding rankable variable, statistic computation, generating scoring function, and generating new Top-k queries.
- Chapter 7 reports the results of the experimental evaluation for Top-k SPARQL query benchmark.
- Chapter 8 draws the conclusion of the work and proposes future extensions of the present work.

Chapter 2

Background

In this chapter will introduce an overview of the main research areas related to the thesis work. The two most important field related to this work is Semantic Web, and Benchmarking. In this chapter we focus on Semantic Web and the related technologies and will introduce benchmarking in the following chapter.

This chapter is divided into the following sections: a brief presentation of the Semantic Web and its technologies (Section 2.1), an introduction to Semantic Web technologies like RDF (Section 2.2), and OWL (Section 2.3), an introduction to SPARQL, the standard query language in Semantic Web (section 2.4), and finally introduce different implementations for SPARQL Query Engines (Section 2.5).

2.1 Semantic Web

Semantic Web is an extension for World Wide Web which enables people to find, share, and combine information across applications. Tim, the inventor of the World Wide Web and director of the World Wide Web Consortium (W3C), supervises the development of proposed Semantic Web standards. He defines the Semantic Web as "a web of data that can be processed directly and indirectly by machines" [Berners-Lee et al., 2001]. In the Semantic Web vision, the information can be processed and manipulated by machines. They enable to understand and respond to complex human requests. To have such an understanding, we require that the relevant information sources be structured semantically. W3C proposed a layered structure, in which each layer contains a set of standards used in semantic web. Figure 2.1 shows this structure. The bottom layers contain technologies that are well known from hypertext web and that without change provide basis for the semantic web.

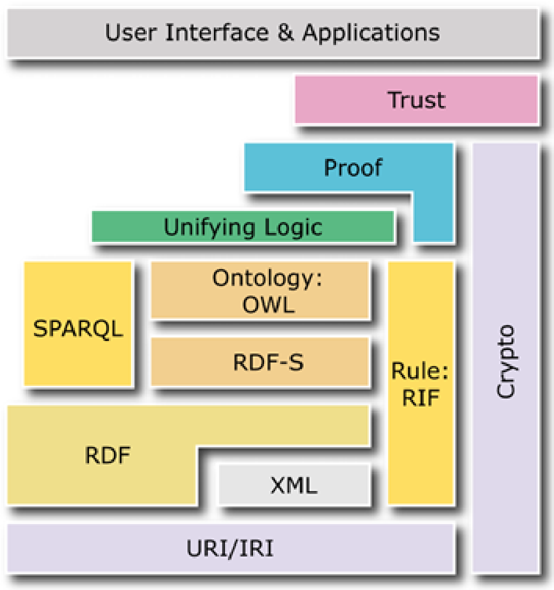


Figure 2.1: Semantic Web Stack

- Internationalized Resource Identifier (IRI), and URI are standards of the World Wide Web which provide means for uniquely identifying resources of the web.
- Unicode is a character encoding standard to represent and manipulate text in many languages.
- eXtensible Markup Language (XML) is a markup language that enables creation of documents composed of structured data and used for data exchange in the Web. XML used as a syntax layer in semantic web.
- XML Namespaces provides a way to use markups from more sources when it is needed to refer more sources in one document.

Middle layers contain technologies standardized by W3C which are the core of semantic web.

- Resource Description Framework (RDF) is a framework for representing information in the Web in the form of triples and graphs.

- RDF Schema (RDFS) provides basic vocabulary for RDF. Using RDFS we can define hierarchies of classes and properties of RDF resources.
- Web Ontology Language (OWL) extends RDF Schema by adding more advanced concepts to describe semantics of RDF statements. It allows expressing additional constraints and precise representation of concepts. It is based on description logic and so brings reasoning power to the semantic web.

Top layers contain technologies that are not yet standardized or implemented.

- RIF or SWRL will bring support of rules expressed in the Rule Interchange Format (RIF) and allow inferring knowledge from existing data.
- Cryptography is important to verify that semantic web statements are coming from trusted source.
- Trust which involves the reliability and trustworthiness of the data.
- User interface is the final layer that will enable humans to use Semantic Web applications.

In the following sections we represent the middle layer technologies which are the basic concepts of Semantic Web: RDF, OWL, SPARQL.

2.2 Resource Description Framework (RDF)

The Resource Description Framework (RDF) is a standard framework proposed by W3C which is used for data interchange on the web. It is used as a general method for conceptual description of data related to web resources. It is based on creating statement about web resources in the form of subject-predicate-object expression which is known as a triple. A set of triples create a directed, labeled graph, where the edges represent the predicates, and nodes represent the web resources which are subjects or objects. RDF is a conceptual model for describing information and used different syntax and serialization formats. XML is one of the common serialization formats which is used in W3C specifications to express an RDF graph as an XML document. In addition, the W3C introduced Notation 3 (or N3) which is easier to use because of its flat notation. The underlying triples encoded in the N3 format are more easily recognizable compared to the XML serialization.

In order to define the triple in a formal way we need to introduce some key concepts of RDF

- A URI reference within an RDF graph (an RDF URI reference) is a Unicode string that produces a valid URI to identify each web resources.
- A Literal is used to identify values such as numbers and dates by means of a lexical representation.
- A blank node is a node that is not a URI reference or a literal. A blank node is anonymous resources that can be used in RDF statements to create a relation between some nodes.

A triple statement consists of subject, predicate, and object. The formal definition of RDF triple and RDF graph are defined in the [Perez et al., 2006] as (Figure 2.2):
Definition 1: Let I, B and L be three pairwise disjoint sets, defined as IRIs, Blank Nodes and Literals, respectively. A triple

$$(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$$

is called an RDF triple, while a set of RDF triples is called an RDF graph.

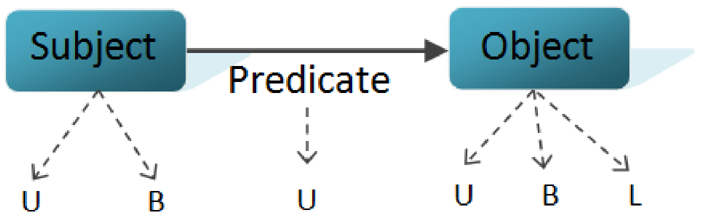


Figure 2.2: RDF Graph

The subject of the RDF triple is either a URI reference, or a blank node. The predicate must be a URI reference. The object can be a URI reference, blank node, or a literal. In the following lines we can see an example of a simple RDF/XML document that describes the resource Agent007.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlsn:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:foaf="http://XMLns.com/foaf/0.1/">
<rdf:Description rdf:ID="Agent007">
<foaf:name>James Bond</foaf:name>
<rdf:type>
<rdfs:Class rdf:about="#Spy"/>
</rdf:type>
</rdf:Description>
</rdf:RDF>
```

Listing 2.1: Caption example.

Each RDF/XML document starts with an unique rdf:RDF element and the definition of the useful namespaces (lines 1-3). The rdf:Description tag (lines 4-9) represents an RDF Statement and it includes the subject of the statement, in our case Agent007. Inside the rdf:Description element we define two predicates, foaf:name (line 5) and rdf:type (lines 6-8). The predicate foaf:name contains an object literal, that is a string with the name of the resource. The predicate rdf:type is a special predicate, which relates the individual resource to its class. In this case Agent007 is an individual of the class Spy (line 7).

2.3 Web Ontology Language

The OWL is a W3C recommendation for representing complex ontologies. Ontology is a set of Description Logic axioms which place constraints on sets of individuals called "classes" and the types of relationships permitted between them. OWL extends the RDF Schema Language with additional language constructs, while constraining the use of RDF(S) constructs. These constraints are required in order to restrain the language to a subset of First Order Logic. The W3C has developed three different sub-languages, with different level of expressiveness and complexity called Profiles [Pascal Hitzler, 2012]:

- OWL 2 EL is designed for ontologies include complex structural descriptions, with huge numbers of classes to manage the terminology. OWL 2 EL has an expressive class and property expression language. OWL 2 EL is domain independent, but It works better for domain an applications which has structurally complex objects. OWL 2 EL disallows negation, disjunction, and universal quantification on properties.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22−rdf−syntax−ns#"
xmlsn:rdfs="http://www.w3.org/2000/01/rdf−schema#"
xmlns:foaf="http://XMLns.com/foaf/0.1/">
<owl:Class rdf:ID="DoubleAgent">
<rdfs:subClassOf rdf:resource="#Spy"/>
<rdfs:subClassOf>
<owl:Restriction>
<owl:onProperty rdf:resource="#hasAgency"/>
<owl:minCardinality>2</owl:minCardinality>
</owl:Restriction>
</rdfs:subClassOf>
</owl:Class>
```

- OWL 2 QL can be realized using standard relational database technology (e.g., SQL) simply by expanding queries in the light of class axioms. This means it can be tightly integrated with RDBMSs and benefit from their robust implementations and multi-user features. It also includes many of the main features of conceptual models such as UML class diagrams and ER diagrams. Expressively, OWL 2 QL can represent key features of Entity-relationship and UML diagrams. So, it can be used as a high level database schema language and it is suitable both for representing database schemas and for integrating them via query rewriting. OWL 2 QL disallows existential quantification of roles to a class expression, property chain axioms and equality.

- The OWL 2 RL profile is aimed at applications that require scalable reasoning without sacrificing too much expressive power. It is designed to be used both in OWL 2 applications, and RDF(S) applications that need some added expressivity from OWL 2, which is achieved by implementation using rule-based technologies. Suitable rule-based implementations of OWL 2 RL under RDF-Based Semantics can be used with arbitrary RDF graphs. As a consequence, OWL 2 RL is ideal for enriching RDF data, especially when the data must be massaged by additional rules. OWL 2 RL disallows statements where the existence of an individual enforces the existence of another individual: for instance, the statement "every person has a parent" is not expressible in OWL RL.

OWL can be serialized in the RDF/XML syntax. In the following example, we use OWL EL to describe the Class of DoubleAgents, which are a SubClass of Spies that work for at least two Agencies.

The owl:Class tag (lines 4-12) is a shortcut to represent an RDF Statement about a resource with a rdf:type of owl:Class. The subject of the statement is the resource DoubleAgent, which represents the class of individual Double Agents. This statement has two predicates, both of the type rdfs:subClassOf. The first predicate (line 5) relates the OWL class DoubleAgent to its super-class, the resource Spy. The second predicate (lines 6-11) states that DoubleAgent is a subclass of the OWL restriction class, which contains all the individuals that are in relationship hasAgency (line 8) with at least 2 resources (line 9).

2.4 SPARQL Protocol and RDF Query Language (SPARQL)

The SPARQL Protocol and RDF Query Language (SPARQL) is the standard query language for RDF, able to retrieve and manipulate data stored in Resource Description Framework format. In January 2008 it became an official W3C Recommendation [Prudhommeaux and Seaborne, 2008]. The SPARQL query language was initially designed to meet the requirements identified in the RDF Data Access Use Cases and Requirements. Only successively it has been formalized through the definition of an official algebra. The new version of SPARQL, SPARQL 1.1, has been released which is completely compatible with the SPARQL specification, but has some new additional features, like aggregates, sub queries, negation and project expressions, i.e. the possibility to use expressions in the SELECT clause. Since RDF is a directed labelled graph data format, SPARQL is defined as a graph-matching query language. The SPARQL language specifies four different query variations for different purposes.

- SELECT query: used to extract a set of variables and their matching values, called set of mappings in the table format.
- CONSTRUCT query: used to provide an RDF graph created directly from the results of the query.
- ASK query: used to provide a simple Boolean value expressing if there are any matches in the graph.
- DESCRIBE query: used to extract an RDF graph based on the information related to the retrieved resources.

Each of these query forms takes a WHERE block to describe and restrict the searched graph pattern using keywords like FILTER, OPTIONAL, AND, or UNION. In the following we describe SELECT query in more details, as we work with this type of query in this thesis.

2.4.1 SELECT Query

The structure of SPARQL SELECT query consists of five clauses (Figure 2.3):

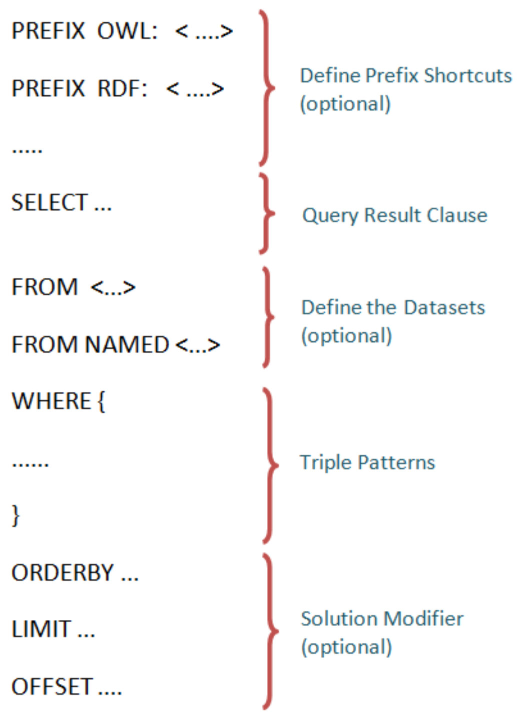


Figure 2.3: Select Query

- PREFIX: The PREFIX keyword associates a prefix label with an IRI. A qualified name is a prefix label and a local part, separated by a colon, which is mapped to an IRI.
- FROM: The FROM allows a query to specify an RDF dataset by reference.
- SELECT: The SELECT keyword specify the form of returning variables and their combinations by introducing new variable.
- WHERE: The WHERE clause provides the graph pattern to match against the data graph. The graph pattern can be in different form

```
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT ?person ?name
WHERE {
?person a foaf:Person.
?person foaf:name ?name
}
ORDER BY ?name
LIMIT 3
```

of simple graph pattern, group graph pattern, optional graph pattern, alternative graph pattern. Different keywords can be used in this clause such as OPTIONAL (for optional patterns), UNION (for unions of patterns), FILTER (for filtering patterns).

- Solution modifiers: this clause use different keywords like ORDER BY, LIMIT,. . . in order to modify the result of the query.

In the following lines we can see an example of a SPARQL query, that retrieves the top 3 person names, ordered alphabetically from dataset shows in table 2.1. Variables are expressed in the form ?variablename.

?person	?name
Agent007	"James Bond"
Agent008	"Johnny Lee Outlaw"
Agent009	"Peter Goodguy"
Agent010	"Alice"
Agent011	"Bob"
Agent012	"Smith"

Table 2.1: RDF Dataset Example

This minimal query contains all the described elements. The WHERE clause matches all triples that have as predicate foaf:name, binding the subjects of each triple to the variable ?person and the object to the variable ?name. Once the WHERE clause has returned its results, the solution modifiers ORDER BY and LIMIT order them based on the ?name and limit the number of solutions in the result set to 3. In the end, the SELECT clause returns to the issuer the ?person and ?name values. If we run this query against the RDF dataset in table 2.1, the results will be the ones in the table 2.2.

In this case the variable ?person is bound to the only resource that is in a relationship foaf:name, that is for example Agent007, and the variable

?person	?name
Agent010	"Alice"
Agent011	"Bob"
Agent007	"James Bond"

Table 2.2: SPQRQL result

?name is bound to the resource with which for example Agent007 is in this relationship.

2.4.2 The SPARQL Algebra

In this section we will introduce two important SPARQL algebras. First we will describe a relational algebra for SPARQL and then we will introduce the algebra on which the official SPARQL specification is based.

[Cyganiak, 2005] proposed a relational algebra for SPARQL to reuse the classical results from query planning and optimization in the relational context.Through the use of this algebra it could be theoretically possible to translate a set of SPARQL queries into SQL, but this approach cannot be fully adopted due to a number of issues and limitations caused by important differences in the two languages.

The main idea in [Cyganiak, 2005] is that an RDF graph can be represented as a table with three columns, ?subject, ?predicate, ?object, where each row is an RDF triple. In a similar way, any SPARQL result set can be seen as a table, or RDF relation, thus SPARQL can be translated into an algebra consisting of a set of operators on RDF relations.

An RDF tuple defined as a partial function from variables to RDF terms. The variables in an RDF tuple are called its attributes and they can be bound or unbound. A set of RDF tuples forms an RDF relation, in which each column is an attribute. Due to the semi-structured nature of RDF, not all tuples will have bound values for all the attributes. [Cyganiak, 2005] also suggests an algorithm to translate SPARQL queries to the operators of the SPARQL relational algebra, which includes several operators from the traditional algebra.

[Perez et al., 2006] introduced a SPARQL algebra which became the foundation of the official SPARQL algebra. The formalization of the SPARQL language lead to a better definition of its specification [Prudhommeaux and Seaborne, 2008], avoiding ambiguities and reducing the gaps in the language. The foundation of this algebra is the concept of mapping, which represents a single solution of any SPARQL query.

In order to proposed an algebraic formalization of the SPARQL, there exist several definitions [P erez et al., 2006]:
Definition 2: Assuming three pairwise disjoint sets I (IRIs), L (literals) and V (variables), a graph pattern expression is defined recursively as:

1. A tuple from $(I \cup L \cup V) \times (I \cup V) \times (I \cup L \cup V)$ is a graph pattern (a triple pattern) .

2. If P1 and P2 are graph patterns, then expressions (P1 AND P2), (P1 OPT P2), and (P1 UNION P2) are graph patterns (conjunction graph pattern, optional graph pattern, and union graph pattern, respectively).

3. If P is a graph pattern and R is a SPARQL built-in condition, then the expression (P FILTER R) is a graph pattern (a filter graph pattern).

A SPARQL built-in condition is composed by elements of the set $(I \cup L \cup V)$ and constants, logical connectives $(\neg, \vee, \wedge)$, ordering symbols $(<, \leq, \geq, >)$, the equality symbol $(=)$, unary predicates like bound, isBlank, isIRI and other features.

An important special case of graph pattern expression is the Basic Graph Pattern, that can be defined as:
Definition 3: A Basic Graph Pattern (BGP) is a set of triple patterns that are connected by the AND (represent also as "·")operator.
Definition 4: Let P be a graph pattern, $var(P)$ denotes the set of variables occurring in P.
Definition 5: A *mappting* μ is a partial function $\mu : V \to (I \cup L \cup B)$. The domain of μ, denoted by $dom\,(\mu)$, is the subset of V where μ is defined. Given a triple graph pattern t and a mapping μ such that $var\,(t) \subseteq dom\,(\mu)$, $\mu\,(t)$ is the triple obtained by replacing the variables in t according to μ.

For example, let's consider the graph pattern P, which is a Basic Graph Pattern containing two triple patterns:

$$P = \{(?X, name, ?Name), (?X, email, ?Email)\}$$

It can be easily seen that $var\,(P) = \{?X,\ ?Name,\ ?Email\}$. A mapping is given, such that:

$$\mu = \{?X \to R1,\ ?Y \to R2,\ ?Name \to john,\ ?Email \to j@ed : exg\}$$

A set of mappings containing only this mapping can be represented also in a tabular form, in which each row is a mapping and each column a variable.

	?X	?Y	?Name	?Email
μ	R1	R2	john	j@ed.ex

Since $var(P) = \{?X, ?Name, ?Email\}$, we can evaluate $\mu(P)$as:

$$\mu\,(P) = \{(R1, name, john), (R1, email, j@ed : ex)\}$$

2.5 SPARQL Query Engines

Even if the RDF/XML form is useful in representing the RDF graph, it is not appropriate for storage, since RDF has its own data model that is very different from the XML tree structure and It is hard to query RDF model through XML query language. Due to the nature of the XML representation, RDF model can be coded in different ways and specific query can not guaranty to return all the answer in the model.

Different storage systems introduce for RDF model based on two approaches: Relational-based approaches which suffer from several inefficiencies due to the semi-structured nature of the RDF data, and native RDF systems called triple-store. Most of these triple-stores map the RDF graph to an indexing scheme, in some cases storing the data directly in the indexes.

Each index indicated by the acronym that represents the values that it index. For example the index SPO works on the values (Subject-Predicate-Object). The main native-storage approaches are:

- Storing all possible subsets of the triple keys in a B+ Tree structure. There are 15 possible subsets, in particular six subsets containing three values, (SPO, SOP, OSP, OPS, PSO, POS), other six containing two values (SP, PS, SO, OS, PO, OP) and three one-valued subsets (S,P,O).
- Reducing the indexes to six and maintaining a three level hierarchy
- Choosing a subset of three triple keys indexes and implement them as B+ trees.

2.5.1 Sesame

Sesame is an architecture that allows persistent storage for RDF data and schema information and subsequent querying of that information [Broekstra et al., 2002b]. It was developed as a part of the European IST project On-To-Knowledge. Sesame provides the first implementation of a query engine

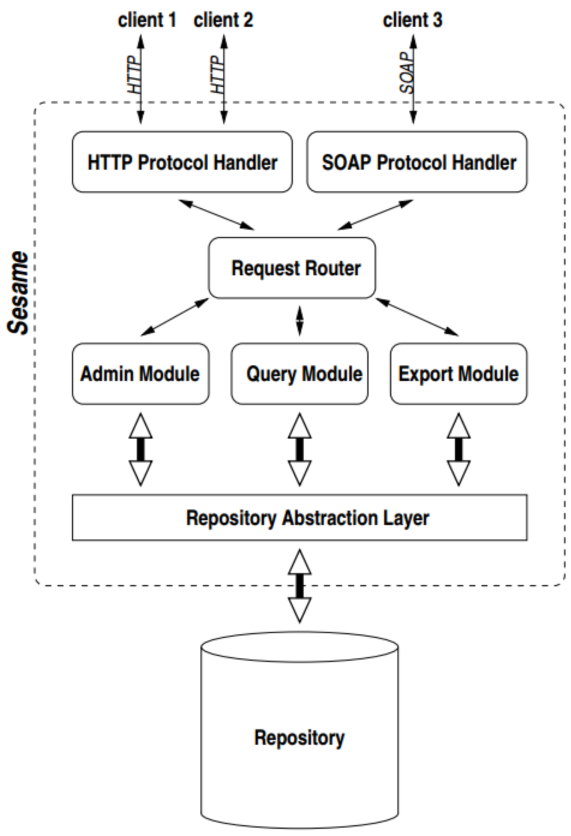


Figure 2.4: Sesame Architecture

for RQL query language which is the first declarative query language for RDF and RDF Schema.

Figure 2.4 shows an overview of Sesame's architecture which consist of the following components:

- Repository Abstraction Layer (RAL): It is an interface that offers RDF-specific methods to its clients and translate them for different DBMS. To keep the Sesame independent from DBMS, the RAL includes all DBMS-specific codes. Thus, it is possible to implement the Sesame on top of the variety of the repositories. Any kind of database such as relational databases (RDBMS), and object-relational databases (ORDBMS) can be used. It is also possible to use existing RDF stores as repository if the RAL knows how to communicate with the RDF store. Files containing RDF can be used as repositories too and for small volumes of data it is a good alternative. The other option is network service that offers basic functionality for storing, retrieving and deleting RDF data.

- RQL query module: It is one of the functional modules which evaluate the RQL queries posed by the user. The Query Module follows the path shows in figure 2.5 when handling a query. First of all, the module parts the query and build the query tree model for it. The optimizer transforms the query tree model into an equivalent model with more efficiency. Since the sesame is DBMS-independent, it needs several optimization techniques in the engine.

- RDF administration module: Another functional module that allows incrementally adding RDF data/schema information, and clearing a repository. In all cases, the admin module checks the consistency of the inferred information with the current contents of the repository and if so, the inferred information is added to the repository.

- RDF export module: It is a very simple module that is able to export the contents of a repository formatted in XML-serialized RDF. Using this module the combination with other RDF tools becomes possible. The RDF Export Module is able to selectively export the schema, the data, or both.

- Protocol Handler: It is used for different ways of communication protocol such as HTTP, RMI (Remote Method Invocation), or SOAP (Simple Object Access Protocol)

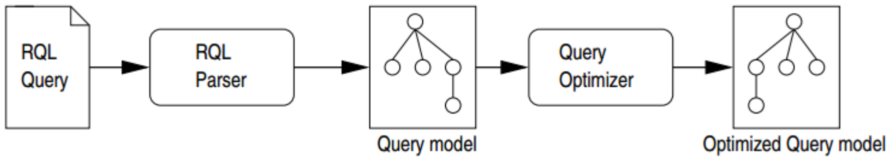


Figure 2.5: Query Module Path

Through the Sesame console we can create different types of repositories such as in-memory, native, and remote. Native repository is a repository that uses on-disk data structure. The native store uses on-disk indexes to speed up querying. It uses B-Trees for indexing statements, where the index key consists of four fields: subject (s), predicate (p), object (o) and context (c). By default, the native repository only uses two indexes, one with a subject-predicate-object-context (spoc) key pattern and one with a predicate-object-subject-context (posc) key pattern. It is possible to define

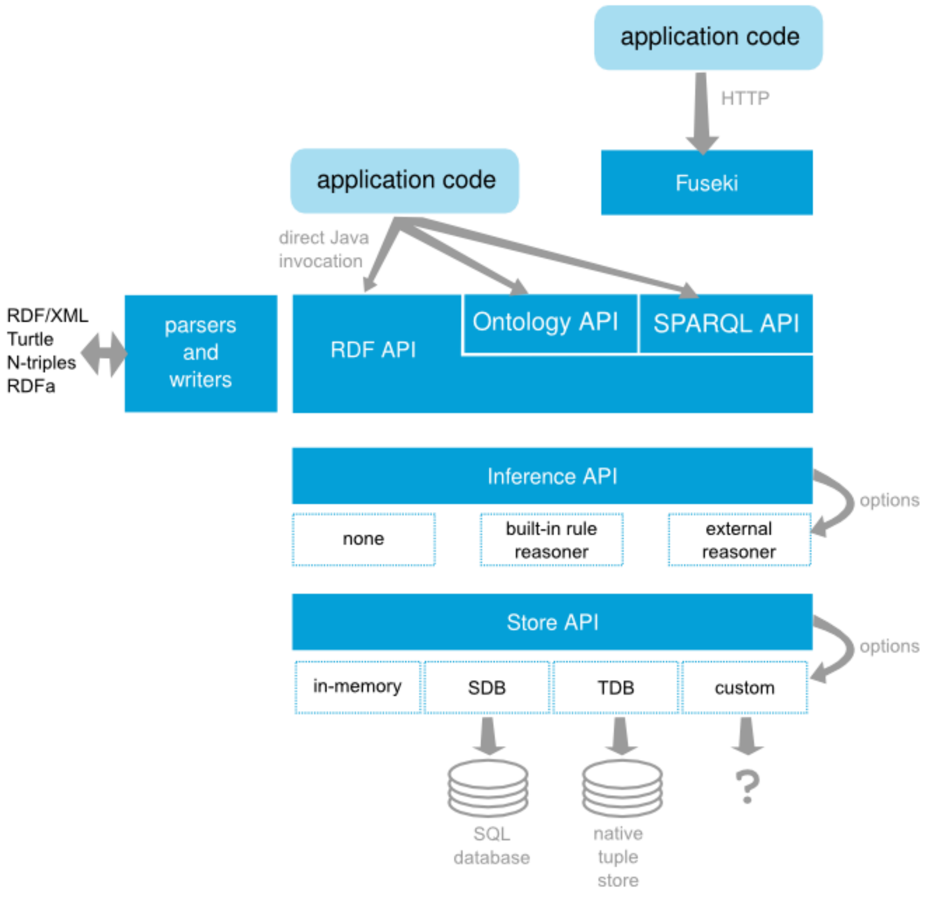


Figure 2.6: Jena Framework

more or other indexes for the native repository through configuration. In this thesis we will use Sesame native for evaluation experiments.

2.5.2 Jena TDB

The Jena framework is an open source Java framework for developing semantic Web and linked-data applications. In Figure 2.6 we can see the overall architecture of the Jena framework. The RDF API is used for adding and removing triples to graphs and finding triples that match particular patterns. Jena can store a graph as an in-memory store, in an SQL database, or as a persistent store using a custom disk-based tuple index.

The inference API provides a number of rule engines to perform inference, either using the built-in rule sets for OWL and RDFS, or using application custom rules. Alternatively, the inference API can be connected up to an external reasoner, such as description logic (DL) engine, to perform the same job with different, specialized, reasoning algorithms.

The SPARQL API is used for query and tracking the under-development areas of the standard. The Ontology API supported two ontology languages for RDF: RDFS, and OWL which is much more expressive. The other component of the Jena package is Fuseki, a SPARQL Server which offering a SPARQL endpoint on the top of the mentioned systems. It can present, and update RDF models over the web using the SPARQL protocol over HTTP.

Jena also provides two storage systems: the TDB triplestore [1], which has been developed for large- scale RDF data storage and retrieval, and the SDB system [2], which allows the storage of RDF data on a number of different RDBMSs

SDB is a Jena component that provides storage for RDF datasets using conventional SQL databases like PostgreSQL, Apache Derby and MySQL. Most of the implementation use a triple table and a separate table for RDF terms as their database layout.The Triples table has three columns, subject, predicate and object. For each triple the values of the columns are identifiers of the related RDF terms, which are called nodes. The Nodes table allows the translation from a node identifier to its effective value.

SDB implements a specific query engine, which reuses the ARQ functionalities for the standard SPARQL algebra translation, while introducing a new algebra operator that is implemented in SQL, called OpSql. SDB creates the SQL syntax based on the modified algebra expression. SDB implements some of the operators like basic graph patterns, joins and left-joins to SQL and leaves to ARQ the other operators. An important detail of the SDB implementation is the fact that the ORDER BY and LIMIT clauses are not forwarded to the relational database, but are performed on the final results by the SDB component. Thus the system cannot take advantage of the relational top-k optimizations.

TDB is the native storage system provided by the Jena framework, which can be used as a high performance RDF store on a single machine. The foundation of the TDB architecture consists in three composite indexes, implemented by B+ trees, in the form Subject-Predicate-Object (SPO), Predicate-Object-Subject (POS) and Object-Subject- Predicate (OSP). B+ Trees are traditional data structures that are usually used in order to implement indexes in relational databases.

A TDB dataset is contained in a single directory of the file system and

[1]http://jena.apache.org/documentation/tdb/index.html
[2]http://jena.apache.org/documentation/sdb/index.html

is composed by:

- The node table: Same as SDB, the node identifiers are stored in the indexes instead of the complete data.The Node table, which works as a dictionary, enabling the translation between Node Ids and the real data.

- Triple and Quad indexes: Quads are used for named graphs, and triples for the default graph. The triple indexes contain triples of 64 bit node identifiers, also called NodeIds. Each NodeId is a unique entry in the node table, which is created during the load process. The assignment of a NodeId to a generic RDF term is not arbitrary, but it is mostly performed by assigning the disk address of the RDF term serialization. This design choice brings several optimizations to the RDF term retrieval, as well as eliminating the need for an index over the node table. The performances are further improved by the use of large caches for the node table.

- The prefixes table: The prefixes table uses a node table and a index for Graph-Prefix-URI (GPU). It provides support for Jena's Prefix Mappings used mainly for presentation of data.

2.5.3 Virtuoso

Virtuoso is a multi-protocol server providing ODBC-JDBC access to relational data stored either within Virtuoso itself or any combination of external relational databases. Besides catering for SQL clients, Virtuoso has a built-in HTTP server providing a DAV repository, SOAP and WS protocol end points and dynamic web pages in a variety of scripting languages [Erling and Mikhailov, 2007].

Virtuoso offers an object-relational database, virtual federated database, a powerful procedure language, Java and .NET run time hosting, and web services access to all of these. These features plus Virtuoso's strong XML support (built-in XQuery, XPath, XSLT, XML Schema) make it an attractive platform for the new crop of Web-2.0 and Semantic Web applications. For example, a single Virtuoso server can provide storage, dynamic web pages, and various XML feed formats for any of these application profiles [Orlink, 2010].

In Virtuoso, an RDF mapping schema consists of declarations of one or more quad storages. In the default quad storage, the system table RDF QUAD consists of four columns G for graph, P for predicate, S for subject and O for object. In culumns P, G, and S there exist IRI ID's and the data type distinguish at run time. The type of the column O is SQL type ANY, which means that any serializable SQL object is acceptable. Given the S or the value of culumn O, the triple should be locatable, so there exist two indices, G, S, P, O and O, G, P, S for the table.

To manage RDF data it is require to have a data type compatible with XML Schema. A distinct type for URIs and having string type with language tags is necessary. Virtuoso have a SQL type called IRI_ID, which is an unsigned 32-bit integer representing the ID of an IRI. The mapping between an IRI_ID and the IRI is represented in two tables, one for the namespace prefixes and one for the local part of the name.

RDF literal with non-default type or language have no exact matches among standard SQL types. Virtuoso introduces a special data type called RDF_BOX in order to handle that cases. Instance of RDF_BOX consists of data type, language, the content (or beginning characters of a long content) and a possible reference to DB.DBA.RDF_OBJ table if the object is too long to be held in-line in some table or should be outlined for free-text indexing.

The object of a triple can be an arbitrarily long value, So it is impossible to have the full value as part of an index key. The value with short string or non-string scalar store in-line. For the longer string, The mapping between ID's of long O values and their full text is kept in a separate table, with the full text or its MD5 checksum as one key and the ID as primary key.

The default RDF representation consists of the tables show in the following. The RDF_URL table contains a mapping between internal IRI ids and their external form.

Since Virtuoso possesses its own SQL-200n implementation; It is possible to process SPARQL queries using the existing query processor, Virtuoso embed SPARQL inside SQL, so SPARQL inherits all the aggregation, grouping, build-in, and user defined functions.

When a SPARQL fragment is found, it is parsed with a SPARQL parser and converted to SQL. The SQL processor picks up from this point on. The result is a SQL-only parse tree that is further processed by the SQL compiler [Orlink, 2010]. If all triples are in one table, the translation is straightforward, with union becoming a SQL union and optional becoming a left outer join.

2.5.4 ARQ Rank

ARQ Rank is a prototype implementation of a SPARQL-RANK query

```
Create table DB.DBA.RDF_ QUAD
(
  G IRI_ID,
  S IRI_ID,
  P IRI_ID,
  O ANY,
  PRIMARY KEY (G, S, P, O)
);
Create index RDF_QUAD_PGOS on RDF_QUAD ( P, G, O, S);

Create table DB.DBA.RDF_URL
(
  RU_IID IRI_ID,
  RU_QNAME VARCHAR,
  PRIMARY KEY (RU_IID)
);

Create table DB.DBA.RDF_OBJ
(
  RO_ID INTEGER,
  RO_VAL VARCHAR,
  RO_LONG LONG VARCHAR,
  PRIMARY KEY (RO_ID)
);
```

engine that extends Jena ARQ 2.8.9. The SPARQL-RANK algebra introduced in [Magliacane, 2011] which is a a rank-aware extension of the SPARQL algebra in order to support ranking as a first-class SPARQL construct. In Section 3.3 we will introduce the SPARQL-RANK algebra in details.

In most of the algebraic representation of SPARQL, the algebraic operators that evaluate the ORDER BY and LIMIT clauses change the sequence of the result after the full evaluation of the graph pattern in the WHERE clause.

This materialize-then-sort scheme can have negative effects on the performance of SPARQL top-k queries, for example a query can have thousand of result but in the LIMIT clause only a limited number of them were requested. Moreover, the scoring function can be complex and expensive to compute, so should be evaluated only when needed.

SPARQL-RANK has been designed to address such a need, evaluating of the scoring function by splitting and delegating to rank-aware operator that are interleaved with other operators and incrementally order the mappings extracted from the data store [Magliacane et al., 2012]. Figure 2.7 shows the proposed extensions of ARQ Rank framework inside the ARQ query engine.

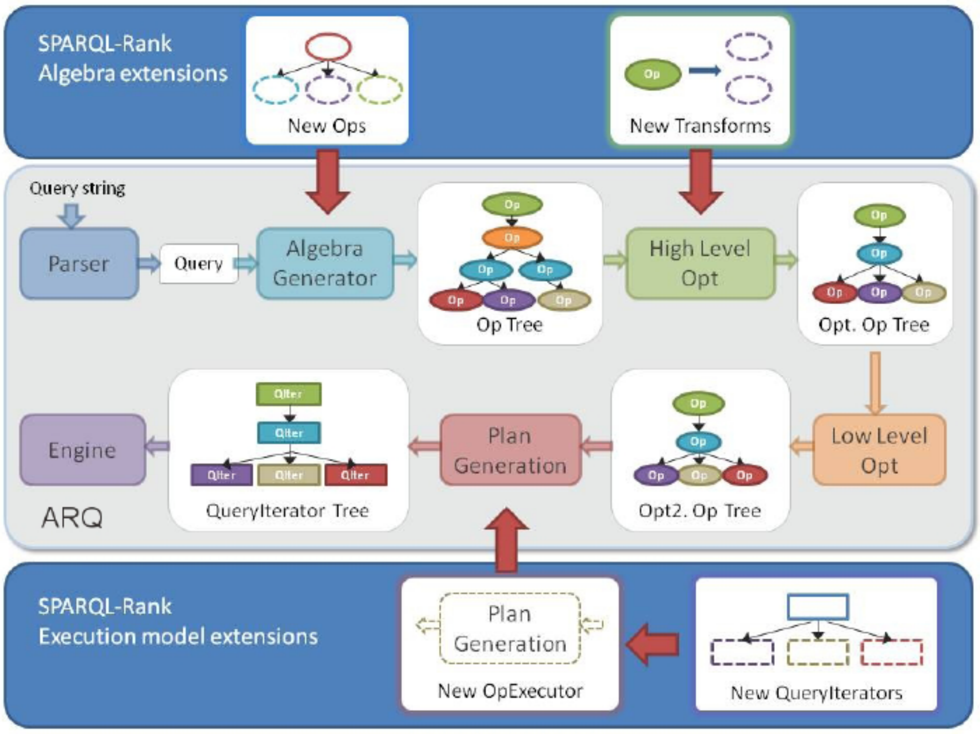


Figure 2.7: The structure of ARQ Rank framework

By defining the new Rank operator ρ and extend the original semantics of existing SPARQL operators such as Selection, Join, Union, Difference,and Left Join, with rank-awareness, interaction between Rank operator and traditional SPARQL operations become possible. In the framework, they convert the extended algebra operators into new Op subclasses, that are able to deal with ranked sets of mappings

The SPARQL-RANK execution model is a pipelined and incremental execution model for top-k queries based on the SPARQL-RANK algebra, that extends the common SPARQL execution model in order to handle ranking query plans. This model follows the RankSQL approach. Following the RankSQL approach, SPARQL-RANK execution model creates trees of physical operators that incrementally output ranked sets of mappings according to their upper-bounds. The execution stops as soon as the requested number of mappings has been drawn from the root operator. A rank-aware execution model calls for rank-aware physical operators.

Query optimization relies on algebraic equivalences to produce several equivalent formulations of a query. The SPARQL-RANK algebra introduced a set of algebraic equivalences that consider the order property. The rank operator ρ can be pushed-down and exploited to limit the number of mappings. For example in the following, we present the equivalences that can apply to the ρ and Join operators:

- Rank splitting: allows splitting the criteria of a scoring function into a series of rank operations, so enabling the individual processing of the ranking criteria.
- Rank commutative law: allows the commutativity of the ρ operand with itself, thus enabling query planning strategies that exploit optimal ordering of rank operators.
- Pushing ρ over Join operator: this law handles swapping Join operator with ρ, thus allowing to push the rank operator only on the operands whose variables also appear in b.

The SPARQL-RANK algebra enables an execution model in which the blocking ordering operator can be split in several non-blocking rank operators. It is possible to push rank operators inside the execution tree and evaluate the order for each ranking criterion incrementally by using the algebraic equivalences.

Chapter 3

Top-k Query

Top-k query is a query that return only a limited top ordered result according to a specified ranking function. Usually the ranking is a function of a set of criteria that are calculated on the tuple values. In SQL, top-k queries are characterized by a combination of an ORDER BY and a LIMIT clause. In this chapter we will introduce the top-k query in relational databases (Section 3.1) and represent the RankSQL framework for efficient evaluation of top-k queries in relational databases. Then we will introduce SPARQL Top-k query (Section 3.2) an SPARQL algebra for top-k queries to overcome limitations in SPARQL(Section 3.3).

3.1 Relational Top-k Query

In traditional relational databases, top-k queries are performed as ordinary queries. In order to evaluate a Top-k query, all inputs are elaborated and the predicates are computed on each of the produced results. After all the results have been materialized, they are sorted according to the ranking function. Finally, only top k results are returned to the query issuer. The materialized-then-sort approach is inefficient, because it requires elaborating all records, creating large intermediate results and evaluating the ranking function on all tuples.

In order to avoid the evaluation of a ranking function on the whole result set produced by a query, top-k queries are usually subject of specific optimization procedures, especially in relational databases. Such optimizations aim at ranking results incrementally during query execution, thus allowing the retrieval and evaluation of only the necessary solutions, instead of computing and ordering all the possible results.

Research on the optimization of Top-k queries has a long history. RankSQL [Li et al., 2005] is one of these optimization approaches which provides a framework to support efficient Top-k queries in relational database systems (RDBMS), by extending relational algebra and query optimization.

3.1.1 RankSQL

RankSQL provide a complete framework for efficient evaluation of Top-k SPJ (Select-Project-Join) queries in relational databases. RankSQL proposed an extended relational algebra called Rank-Relational algebra. Furthermore the authors present a pipe-lined and incremental execution model with new physical operators, and rank-aware query optimization. In this framework ranking consider as a first-class database construct like Boolean Filtering.

A rank-relational query Q, is a traditional SPJ query augmented with ranking predicates. In terms of relational algebra. Such queries can be presented in the canonical form:

$$Q = \pi_* \lambda_k \tau_{\mathcal{F}(p_1,...,P_n)} \sigma_{\mathcal{B}(c_1,...,c_n)}(R_1 \times ... \times R_h)$$

That is, upon the product of the base relations $(R_1 \times ... \times R_h)$, the following two types of operations are performed, before the top k tuples (which we denote by λ_k) with projected attributes (as π_* indicates) are returned as the results.

- Filtering: the selection operator σ_β filters the results by a Boolean function $\mathcal{B}(c_1, ..., c_n)$, where c_i are Boolean predicates (e.g., $\beta = (c_1 \wedge ... \wedge c_n\ )$, and

- Ranking: the sorting operator $\tau_\mathcal{F}$ ranks the results by a monotonic scoring functionn $\mathcal{F}(p_1, ..., P_n)$, where pi are rank predicates (e.g., $\mathcal{F} = p_1 + p_2 + p_3$)

Formally, Q returns a sorted list K of k top tuples, ranked by $\mathcal{F}$, from the qualified tuples $R_\beta = \sigma_{\mathcal{B}(c_1,...,c_n)}(R_1 \times ... \times R_h)$.

Each tuple u has a predicate score $p_i[u]$ for every p_i and its overall query score is$\mathcal{F}(p_1, ..., p_n)[u] = \mathcal{F}(p_1[u], ..., p_n[u])$.

In this formalization the ranking and selection criteria are called rank and Boolean predicates respectively. Selection restricts tuple membership by applying a function $\mathcal{B}$ of Boolean selection or join predicates, and ranking restrict order by applying a function $\mathcal{F}$ of corresponding ranking predicates.

The final ordering of the results is based on the scoring function, which combines the rank predicates. As a standard assumption, the scoring function F is monotonic, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}(x_1, \ldots, x_n) \geq \mathcal{F}(y_1, \ldots, y_n)$ when $\forall i : x_i \geq y_i$. Another assumption is that each rank predicate can assume only real values in the interval [0,1]. This limitation can be easily overcome in real-life datasets by normalization.

Evaluating rank-relational queries efficiently can be achieved by fulfilling the two following requirements:

- Splitting: Ranking should be evaluated in stages, predicate by predicate instead of monolithic.
- Interleaving: Ranking should be interleaved with other operators instead of always after filtering.

In order to enable rank-aware query processing and optimization, the first step is to extend the relational algebra to handle ranking. In the extended model, they define rank-relation as a relation with its tuples scored and ordered according to some ranking function.

In this model ranking predicates will be processed in several steps. At any intermediate steps some predicates will be evaluated and It is not possible to have the final ranking. The partial ranking of the tuples by their incomplete scores is defined which is consistent with the final ranking.

Definition 6: With respect to a scoring function $\mathcal{F}(p_1, ..., P_n)$, and a set of evaluated predicates $P = \mathcal{F}p_1, ..., P_j$, the maximal-possible score (or upperbound) of a tuple t, denoted $\overline{\mathcal{F}}_p[t]$, is defined as

$$\overline{\mathcal{F}}_{\mathcal{P}}(p_1, ..., P_n)\,[t] = \mathcal{F}\begin{pmatrix} p_i = p_i\,[t] \quad if \quad p_i \in \mathcal{P} \quad \forall i \\ p_i = 1 \quad otherwise \end{pmatrix}$$

In the RankSQL framework queries are evaluated incrementally, thus the output tuples will be passed to subsequent operators partially ordered in accordance with the following ranking principle.

Given two tuples t_1 and t_2, if $\mathcal{F}_{\mathcal{P}}[t_1] > \mathcal{F}_{\mathcal{P}}[t_2]$, then t_1 must be further processed if we necessarily further process t_2 for query answering.

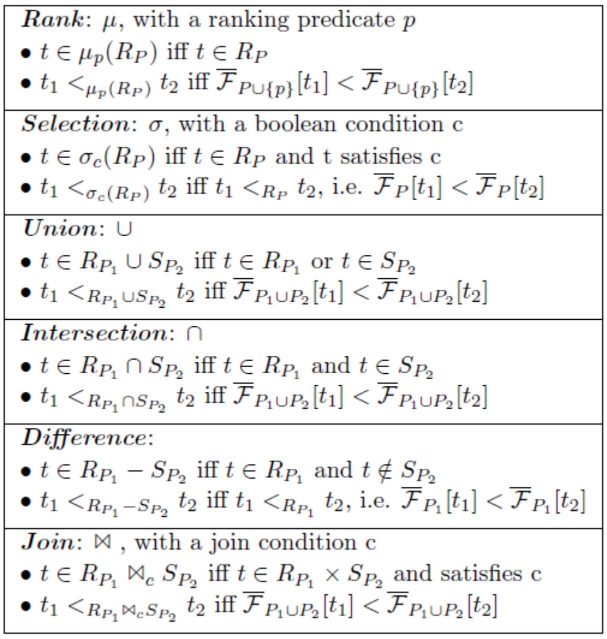
By this ranking principle, the authors formalized the Rank-Relation concept as:

A rank-relation R_P, with respect to relation R and monotonic scoring function $\mathcal{F}(p_1, ..., P_n)$, for $P \subseteq p_1, ..., p_j$, is the relation R augmented with the following ranking induced by P.

- (Scores) The score for a tuple t is the maximal-possible score of t under $\mathcal{F}$, when the predicates in P have been evaluated, i.e., $\mathcal{F}_P[t]$. It is an implicit attribute of the rank-relation.
- (Order) An order relationship $<_{R_P}$ is defined over the tuples in R_P by ranking their scores, i.e., $\forall t_1, t_2; t_1 <_{R_P} t_2 \Leftrightarrow \mathcal{F}_{\mathcal{P}}[t_1] < \mathcal{F}_{\mathcal{P}}[t_2]$.

Rank-relation possesses two logical properties: 1) membership as defined by the relation R, and 2) order induced by evaluated predicates P. In order to effectively split and interleave ranking with other operations, the authors extend relational algebra by adding a new rank operator μ. The rank operator (μ) evaluates an additional predicate p upon a rank-relation, which is already ordered by a predicate set P, producing a new rank-relation ordered by $P \cup \{p\}$.

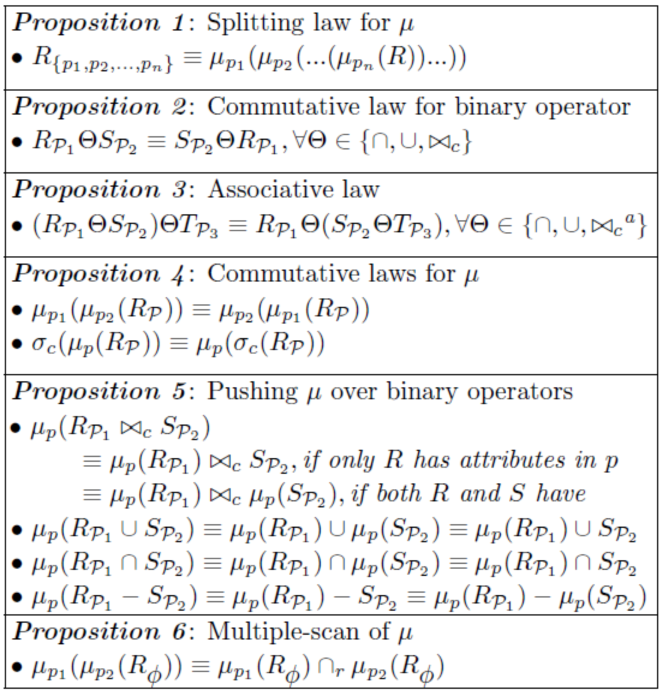
Figure 3.1 shows how the authors extend the original semantics of existing relational operators such as join, selection, Intersection, with rankawareness, thus enabling the interaction between the new rank operator μ and traditional Boolean operations.



Rank: μ, with a ranking predicate p • $t \in \mu_p(R_P)$ iff $t \in R_P$ • $t_1 <_{\mu_p(R_P)} t_2$ iff $\overline{\mathcal{F}}_{P\cup\{p\}}[t_1] < \overline{\mathcal{F}}_{P\cup\{p\}}[t_2]$
Selection: σ, with a boolean condition c • $t \in \sigma_c(R_P)$ iff $t \in R_P$ and t satisfies c • $t_1 <_{\sigma_c(R_P)} t_2$ iff $t_1 <_{R_P} t_2$, i.e. $\overline{\mathcal{F}}_P[t_1] < \overline{\mathcal{F}}_P[t_2]$
Union: $\cup$ • $t \in R_{P_1} \cup S_{P_2}$ iff $t \in R_{P_1}$ or $t \in S_{P_2}$ • $t_1 <_{R_{P_1}\cup S_{P_2}} t_2$ iff $\overline{\mathcal{F}}_{P_1\cup P_2}[t_1] < \overline{\mathcal{F}}_{P_1\cup P_2}[t_2]$
Intersection: $\cap$ • $t \in R_{P_1} \cap S_{P_2}$ iff $t \in R_{P_1}$ and $t \in S_{P_2}$ • $t_1 <_{R_{P_1}\cap S_{P_2}} t_2$ iff $\overline{\mathcal{F}}_{P_1\cup P_2}[t_1] < \overline{\mathcal{F}}_{P_1\cup P_2}[t_2]$
Difference: • $t \in R_{P_1} - S_{P_2}$ iff $t \in R_{P_1}$ and $t \notin S_{P_2}$ • $t_1 <_{R_{P_1}-S_{P_2}} t_2$ iff $t_1 <_{R_{P_1}} t_2$, i.e. $\overline{\mathcal{F}}_{P_1}[t_1] < \overline{\mathcal{F}}_{P_1}[t_2]$
Join: $\bowtie$, with a join condition c • $t \in R_{P_1} \bowtie_c S_{P_2}$ iff $t \in R_{P_1} \times S_{P_2}$ and satisfies c • $t_1 <_{R_{P_1}\bowtie_c S_{P_2}} t_2$ iff $\overline{\mathcal{F}}_{P_1\cup P_2}[t_1] < \overline{\mathcal{F}}_{P_1\cup P_2}[t_2]$

Figure 3.1: Rank-relational Algebra Operators

Query optimizers essentially rely on algebraic equivalences to generate efficient query plans. The extension of the rank-relational model and algebra implies a reformulation of the traditional algebraic equivalences. The new algebraic equivalence laws are presented in figure 3.2 and they lay the foundation for query optimization. Based on these laws, many equivalent query plans can be created by splitting and interleaving operators.



Proposition 1: Splitting law for μ
- $R_{\{p_1,p_2,\ldots,p_n\}} \equiv \mu_{p_1}(\mu_{p_2}(\ldots(\mu_{p_n}(R))\ldots))$

Proposition 2: Commutative law for binary operator
- $R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2} \equiv S_{\mathcal{P}_2} \Theta R_{\mathcal{P}_1}, \forall \Theta \in \{\cap, \cup, \bowtie_c\}$

Proposition 3: Associative law
- $(R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2}) \Theta T_{\mathcal{P}_3} \equiv R_{\mathcal{P}_1} \Theta (S_{\mathcal{P}_2} \Theta T_{\mathcal{P}_3}), \forall \Theta \in \{\cap, \cup, \bowtie_c{}^a\}$

Proposition 4: Commutative laws for μ
- $\mu_{p_1}(\mu_{p_2}(R_{\mathcal{P}})) \equiv \mu_{p_2}(\mu_{p_1}(R_{\mathcal{P}}))$
- $\sigma_c(\mu_p(R_{\mathcal{P}})) \equiv \mu_p(\sigma_c(R_{\mathcal{P}}))$

Proposition 5: Pushing μ over binary operators
- $\mu_p(R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2})$
  $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c S_{\mathcal{P}_2}$, *if only R has attributes in p*
  $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c \mu_p(S_{\mathcal{P}_2})$, *if both R and S have*
- $\mu_p(R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup S_{\mathcal{P}_2}$
- $\mu_p(R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap S_{\mathcal{P}_2}$
- $\mu_p(R_{\mathcal{P}_1} - S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) - S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) - \mu_p(S_{\mathcal{P}_2})$

Proposition 6: Multiple-scan of μ
- $\mu_{p_1}(\mu_{p_2}(R_{\phi})) \equiv \mu_{p_1}(R_{\phi}) \cap_r \mu_{p_2}(R_{\phi})$

Figure 3.2: Algebraic Equivalence Laws

In the next step, the authors introduce the RankSQL execution model which is an extension of the common execution model that is able to handle ranking query plans with the following properties: 1) operators incrementally output rank-relations (tuple streams pass through operators in the order of maximal-possible scores according to the ranking principle); 2) the execution stops when the requested number of results are reported or no more results are available. In order to output tuples ordered by their current maximal-possible score, each operator can pass a tuple to its successor only if it is guaranteed to have a higher score than that of all future output tuples. This decision is implemented with a different strategy for each kind of operator.

The implementation of the new rank operator μ adopts the MPro algorithm, while the rank-join operator is based on the HRJN and the NRJN physical operators which are two non-blocking join operators based on the rank-join algorithm. Furthermore the rank-scan is performed by exploiting the IndexScan operation in PostgresSQL, which accesses the base table using a B+ tree index on a ranking predicate p. The purpose of the rank-join is to take advantage of individual orders of its inputs to produce join results ordered on a user-specified scoring function.

3.2 SPARQL Top-k Query

Top-k queries can be easily defined also in the SPARQL language as SELECT queries with an ORDER BY and LIMIT clause. Moreover, by using projection expression is it possible to explicit the scores of the ranked mappings, even if this is a feature that is officially not part of SPARQL 1.0, but of SPARQL 1.1.

The standard execution of a SPARQL top-k query is very inefficient. [Magliacane, 2011] proposed two optimization technique for SPARQL top-k queries.in order to optimized SPARQL top-k queries, two different types of query introduced:

- Class A: Top-k queries that do not allow variables involved in the ranking to appear inside an OPTIONAL or UNION clause.
- Class B: Top-k queries that allow variables involved in the ranking to appear inside an OPTIONAL or UNION clause.

For both classes, there exist two standard assumptions, one regarding the monotonicity of the combining ranking function and the other assuming that the values of the single predicates are in the interval [0,1]. They introduce two different approaches based on the two different classes of queries:

- Supporting Class A Queries: They propose to configure a virtual RDF engine in order to work with the RankSQL database. Additionally, it is required to modify the systems in order to push down LIMIT and ORDER BY clauses to the database. This solution is the easiest both from a theoretical and a practical point of view, because it allows to reuse the rank-relational algebra formalization and the RankSQL implementation. Moreover, there is no need to change the SPARQL algebra, because we consider only a fragment that is equivalent to the rank-relational algebra.
- Supporting Class B Queries: Since it is not possible to rely on the rank-relational algebra for this class of queries, they proposed an extension of the SPARQL algebra into a new rank-aware SPARQL-Rank algebra. Furthermore, they introduce the SPARQL-Rank execution model

based on the RankSQL execution model, which provides a set of algorithms for new physical operators. By extending the query engine in order to fit the SPARQL-Rank algebra and execution model, we are able to support an efficient execution for this class of queries.

3.3 SPARQL Rank algebra

SPARQL Rank algebra [Magliacane et al., 2012] proposed in order to support top-k SPARQL queries that have an ORDER BY clause that can be formulated as a scoring function combining several ranking criteria. In order to provide the SPARQL Rank algebra, some basic concept should be defied.
Definition 7: Given a graph pattern P, a **ranking criterion** $b : \mathbb{R}^m \to \mathbb{R}$ is a function defined over a set of variables $?x_j \in var(P)$. The evaluation of a ranking criterion on a mapping μ, that is, the substitution of all of the variables $?x_j$ with the corresponding values from the mapping, is indicated by $b\,[\mu]$. A criterion b can be the result of the evaluation of any built-in function (having an arbitrary cost) of query variables.
Definition 8: A **scoring function** on P is an expression of the form $\mathcal{F}$ defined over the set B of ranking criteria. As typical in ranked queries, the scoring function $\mathcal{F}$ is assumed to be monotonic, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}\,(x_1, \ldots, x_n) \geq \mathcal{F}\,(y_1, \ldots, y_n)$ when $\forall i : x_i \geq y_i$.

In order to evaluate a scoring function , the variables in $var(P)$ that contribute in the evaluation of $\mathcal{F}$ must be bound. Since OPTIONAL and UNION clauses can introduce unbound variables, we assume all the variables in $var(P)$ to be certainly bound, i.e. variables that are certainly bound for every mapping produced by P.

The following query provides an example of the scoring function $\mathcal{F}$ calculated over the ranking criteria $g1(?avgRat1)$, $g2(?avgRat2)$, and $g3(?price1)$. We note that $?avgRat1$, $?avgRat2$, and $?price1$ are certain bound variables, as the query contains no OPTIONAL or UNION clauses. The result of the evaluation is stored in the $?score$ variable, which is later used in the ORDER BY clause.

Overall, a key property if SPARQL Rank is the ability to retrieve the first k results of a top-k query before scanning the complete set of mappings resulting from the evaluation of the WHERE clause. To enable such a property, the mappings progressively produced by each operator should flow in an order consistent with the final order, i.e., the order imposed by $\mathcal{F}$. When the evaluation of a SPARQL top-k query starts on the Basic Graph Patterns the resulting mappings are unordered. As soon as some $\mathcal{B} = \{b_1, \ldots, b_j\}\,(\,with\;\; j < |B|)$ of the ranking criteria can be computed

```
SELECT ?product ?offer (F(g1(?avgRat1),g2(?avgRat2),g3(?price1)) AS ?score)
WHERE {
?product hasAvgRating1 ?avgRat1 .
?product hasAvgRating2 ?avgRat2 .
?product hasName?name .
?product hasOffers ?offer .
?offer hasPrice ?price1 .
}
ORDER BY DESC(?score) LIMIT 10
```

Listing 3.1: A top-k SPARQL query that retrieves the best ten overs of products ordered by a function of the product user ratings and the over price.

(i.e., when $var(b_j) \subseteq dom(\mu)$), an order can be imposed to a set of mappings Ω by evaluating for each $\mu \in \Omega$ the upper bound of $\mathcal{F}[\mu]$ as:

$$\overline{\mathcal{F}}_{\mathcal{B}}[\mu] = \mathcal{F}\begin{pmatrix} b_i = b_i[\mu] & if \quad b_i \in \mathcal{B} \quad \forall i \\ b_i = max(b_i) & otherwise \end{pmatrix}$$

where $max(b_i)$ is the application-specific maximal possible value for the ranking criterion b_i. $\overline{\mathcal{F}}_{\mathcal{B}}[\mu]$ is the upper bound of the score that μ can obtain, when $\mathcal{F}_{\mathcal{B}}[\mu]$ is completely evaluated, by assuming that all the ranking criteria still to evaluate will return their maximal possible value. We can now formalize the notion of ranked set of mappings.

A ranked set of mappings Ω_B, with respect to a scoring function $\mathcal{F}$, and a set $\mathcal{B}$ of ranking criteria, is the set of mappings Ω augmented with an order relation $<_{\Omega_B}$ defined over Ω. which orders mappings by their upper bound scores, i.e., $\forall\ \mu_1,\ \mu_2\ \in \Omega : \mu_1\ <_{\Omega_B}\ \mu_2\ \Leftrightarrow \overline{\mathcal{F}}_{\mathcal{B}}[\mu_1] < \overline{\mathcal{F}}_{\mathcal{B}}[\mu_2]$.

The monotonicity of $\mathcal{F}$ implies that $\mathcal{F}_{\mathcal{B}}$ is always an upper bound of $\mathcal{F}$, i.e. $\overline{\mathcal{F}}_{\mathcal{B}}[\mu_2]\ \geq\ \mathcal{F}_{\mathcal{B}}[\mu_2]$ for any mapping $\mu \in \Omega_B$, thus guaranteeing that the order imposed by $\overline{\mathcal{F}}_{\mathcal{B}}$ is consistent with the order imposed by $\mathcal{F}$.

Note that a set of mappings on which no ranking criteria is evaluated ($\mathcal{B} = 0$) is consistently denoted as Ω_0 or simply Ω.

Starting from the notion of ranked set of mappings, SPARQL Rank introduces a new rank operator ρ, representing the evaluation of a single ranking criterion, and redefines the Selection, Join, Union, Difference,and Left Join operators, enabling them to process and output ranked sets of mappings.The rank operator ρ_b evaluates the ranking criterion $b \in B$ upon a ranked set of mappings $\Omega_{\mathcal{B}}$ and returns $\Omega_{\mathcal{B}\cup\{b\}}$, i.e. the same set ordered by $\overline{\mathcal{F}}_{\mathcal{B}\cup\{b\}}$.

Chapter 4

SPARQL Benchmarking

In this chapter we will introduce the SPARQL Benchmarking. In the first section (Section 4.1) we compare some existing SPARQL benchmarks. In the following (Section 4.3) we will introduce the Berlin benchmark. Finally we will represent the DBpedia Benchmark which we choose it as a foundation for our work, in more detail (Section 4.4)

4.1 Database Benchmarking

Benchmarking a given database is a process of performing well defined tests on that particular database management system for the purpose of evaluating its performance. The Response time and the throughput are the two main criteria on which the performance of a database can be measured. To be useful, a domain-specific benchmark must meet four important criteria. It must be[Gray, 1993]:

- Relevant: It must measure the peak performance and price/performance of systems when performing typical operations within that problem domain.
- Portable: It should be easy to implement the benchmark on many different systems and architectures.
- Scaleable: The benchmark should apply to small and large computer systems. It should be possible to scale the benchmark up to larger systems, and to parallel computer systems as computer performance and architecture evolve.

- Simple: The benchmark must be understandable, otherwise it will lack credibility.

Wisconsin benchmark [DeWitt, 1993] is one the most widely used benchmark to test the performance of relational query systems. It is a portable and scaleable benchmark which measure the performance of parallel databases with simple relational operators. The Wisconsin benchmark has some weaknesses such as its single-user nature, the absence of bulk updates, database load and unload tests, utility function tests, lack of outer join tests, and the relative simplicity of the various complex join queries.

AS3AP benchmark [Turbyfill et al., 1993] provides a more complete metric and include some of the missing feature of Wisconsin benchmark. It gives a more realistic evaluation of the overall performance of a relational system used for database queries.

The Set Query Benchmark [O'Neil, 1993] is used to evaluate the ability of systems to process complex queries typically found in decision-support applications and data-mining applications. Wisconsin and AS3AP are used simple relational queries, however most decision support applications need to test more complex query set.

4.2 SPARQL Benchmarks

The RDF quickly become an standard for representing information in web, search engines, companies, and organizations. The main reason of this acceptance is the flexibility of the RDF in representing data ranging from structured to unstructured data. However, this flexibility has some cost. RDF data management become a challenges, as no assumption can be made about the type and structure of the data. There is a growing need for benchmarks to compare the performance of storage systems such as Virtuoso [Erling and Mikhailov, 2007], Sesame [Broekstra et al., 2002a], and Jena-TDB [Owens et al., 2008], which use SPARQL endpoints through the SPARQL protocol. To test the performance of these RDF stores different RDF benchmark have been developed such as

One of the first RDF benchmarks is The Lehigh University Benchmark (LUBM) [Guo et al., 2005], which focused on performance, completeness and soundness of OWL reasoning engines. It generates synthetic data for universities, their departments, their professors, employees, courses and publications. The small number of classes are used in the benchmark, limits the variability of data and creates repetition in the structure of LUMB, which causes query caching, and has affects the overall average query times.

SP2Bench [Schmidt et al., 2009] is another more recent benchmark for RDF store. Its RDF data is based on the DBLP Computer Science Bibliography which includes information about publications and their authors. Using SP2Bench Generator, SP2Bench generates its synthetic data which suffers from lake of heterogeneity. The benchmark queries vary in common characteristics like selectivity, query and output size, and different types of joins. Comparing to LUMB, the main advantage of SP2Bench is that its query contain SPARQL features such as FILTER, and OPTIONAL. Different engines are used in SP2Bench such as the in-memory engines like ARQ, Sesame memory and engines with physical back-end like Redland, SBD, Sesame DB, and Virtuoso.

The Berlin SPARQL Benchmark (BSBM) [Bizer and Schultz, 2009] is a benchmark for RDF stores, which is applied to various RDF stores, such as Sesame, Virtuoso, and Jena-TDB and two SPARQL-to-SQL rewriters (D2R Server and Virtuoso RDF Views). It is based on an e-commerce use case in which there exist a set of products provided by a set of vendors and consumers are able to post reviews for products. The queries contain various SPARQL features. It simulates the real user operation by considering a sequence of queries which execute during a use case. Although the data and queries are artificial, they choose an effective testing strategy.

DBpedia SPARQL Benchmark (DBPSB)[Morsey et al., 2011] applies a novel methodology to create heterogeneous datasets of different sizes derived from the DBpedia knowledge base. In DBPSB, in order to create a set of template queries, they perform query analysis and clustering on query logs. They used these templates by parametrization, to generate the actual benchmark queries DBPSB was applied to different RDF stores such as Virtuoso, Sesame, Jena-TDB, and BigOWLIM [Bishop et al., 2011], to evaluate the performance and scalability of them. In the following sections we will represent Berlin SPARQL Benchmark, and DBpedia SPARQL Benchmark in more details.

4.3 Berlin SPARQL Benchmark

Berlin SPARQL Benchmark (BSBM) is proposed to compare the performance of four popular RDF stores: Sesame [Broekstra et al., 2002a], Virtuoso [Erling and Mikhailov, 2007], Jena TDB [Owens et al., 2008], and Jena SDB with the performance of two SPARQL-to-SQL rewriters: D2R Server, and Virtuoso RDF Views, as well as the performance of two relational database management systems: MySQL, and Virtuoso RDBMS. The benchmark is built based an e-commerce use case, where a set of products is offered by different vendors and consumers have posted reviews about products. The benchmark query mix simulate the search and navigation pattern of a consumer looking for a product.

The goals of the Berlin SPARQL Benchmark consist of comparison of different storage systems, simulation the use case with concurrent clients, and measuring performance against large amounts of RDF data. The benchmark can be useful both for application developers and RDF stores developers. Application developers can choose an appropriate storage system according to their requirement with the help of the benchmark. RDF stores developers also able to find the weaknesses and improve the current systems.

The implementation of the benchmark consists of a data generator and a test driver. The data generator create randomly large datasets using the number of products as scale factor. The data generator has two representations of the benchmark data: An RDF representation and a relational representation.

The test driver executes sequences of SPARQL queries over the SPARQL protocol against the system under test. In order to mimic a realistic workload, the test driver can simulate multiple clients that concurrently execute query mixes against the system under test. The queries are parametrized with random values from the benchmark dataset, to reduce the amount of caching. In order to measure the performance of the systems under normal working conditions, the test drivers executes a series of warm-up query mixes. The SQL representations of the query mix are defined to execute against relational databases and compare the SPARQL results with the performance of traditional RDBMS.

In the following subsection we describe the dataset generation and the query mix in more details.

4.3.1 The benchmark dataset

The BSBM benchmark is established in an e-commerce use case where different vendors offer set of products and consumers have posted reviews about products. The benchmark defines an abstract model for this use case. The size of the datasets can be different according to the number of the product as a scale factor.

The benchmark defines two representation of data model: RDF representation, and relational representation. The data model contains the following classes: Product, ProductType, ProductFeature, Producer, Vendor, Offer, Review, and Person. Figure 4.1 shows the data model in details. The most important data production rules which are used to generate the RDF or relational dataset are in the following:

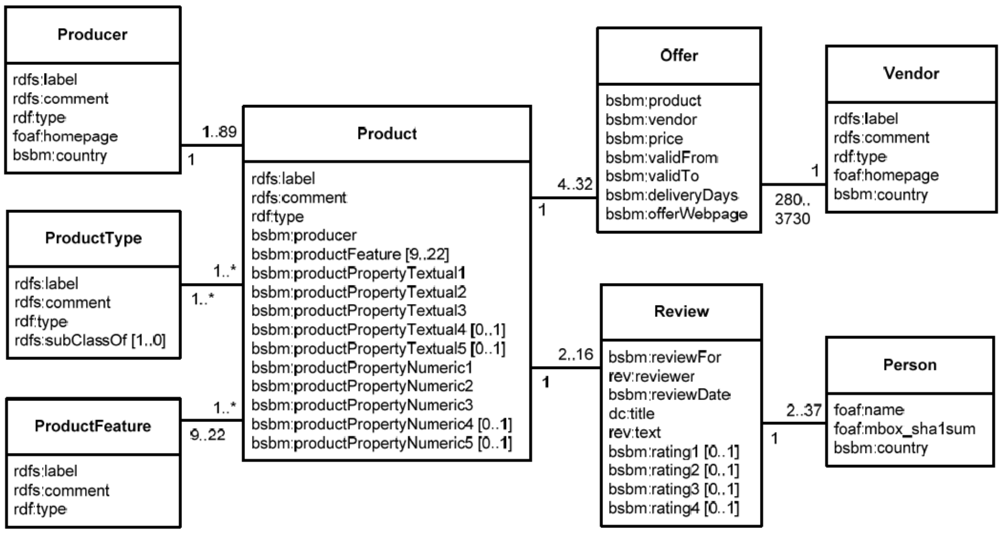


Figure 4.1: Overview of the abstract data model.

- The data generator creates n product instances which are described by a label and a comment. products have 3 to 5 textual properties and between 3 and 5 numeric properties with random value.
- Products have a type which is part of a type hierarchy. The depth of hierarchy and the branching factor of root are calculated by the specified formula.
- Products have a variable number of product features. Each product type in the type hierarchy is assigned with a random number of product features.
- Producers produce products. Products are distributed over producer using a normal distribution.
- Vendors offer products and described by a label, a comment, a home-page URL and a country URI.
- Countries have the following distribution: US 40%, UK 10%, JP 10%, CN 10%, 5% DE, 5% FR, 5% ES, 5% RU, 5% KR, 5% AT.
- Offers are valid for a specific period and contain a price between \$5 and \$10000. The number of days it takes to deliver the product is

between 1 and 21. Number of Offers per products follows a normal distribution.

- Reviews consist of a title and a review text between 50 and 300 words. Reviews have up to four ratings with a random integer value between 1 and 10.

- Reviewers are described by their name, mailbox checksum and the country the reviewer lives in.

Table 4.1 summarizes the number of instances of each class in BSMB datasets of different sizes.

Total number of triples	250K	1M	25M	100M
Number of products	666	2,785	70,812	284,826
Number of product features	2,860	4,745	23,833	47,884
Number of product types	55	151	731	2011
Number of producers	14	60	1422	5,618
Number of vendors	8	34	722	2,854
Number of offers	13,320	55,700	1,416,240	5,696,520
Number of reviewers	339	1432	36,249	146,054
Number of reviews	6,660	27,850	708,120	2,848,260
Total number of instances	23,922	92,757	2,258,129	9,034,027

Table 4.1: Number of instances in BSBM datasets of different sizes

4.3.2 The Query Mix

There are two principle options for the design of benchmark query mixes [Gray, 1993]:

1. Design the queries to test specific features of the query language or to test specific data management approaches.

2. Base the query mix on the specific requirements of a real world use case.

The Berlin Benchmark follow the second principle and designed the query mix as a sequence of queries which are used in a use case. The query mix simulate the procedure of navigating and searching for a product. This sequence of queries may be executed by a shopping portal in which consumer

```
Query 1: Find products for a given set of generic features.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 3: Find products having some specific features and not having one feature.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 4: Find products matching two different sets of features.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 2: Retrieve basic information about a specific product for display purposes.
Query 5: Find products that are similar to a given product.
Query 7: Retrieve in−depth information about a product including offers and reviews.
Query 7: Retrieve in−depth information about a product including offers and reviews.
Query 6: Find products having a label that contains a specific string.
Query 7: Retrieve in−depth information about a product including offers and reviews.
Query 7: Retrieve in−depth information about a product including offers and reviews.
Query 8: Give me recent English language reviews for a specific product.
Query 9: Get information about a reviewer.
Query 9: Get information about a reviewer.
Query 8: Give me recent English language reviews for a specific product.
Query 9: Get information about a reviewer.
Query 9: Get information about a reviewer.
Query 10: Get cheap offers which fulfil the consumer's delivery requirements.
Query 10: Get cheap offers which fulfil the consumer's delivery requirements.
Query 11: Get all information about an offer.
Query 12: Export information about an offer into another schema.
```

Listing 4.1: The BSBM query mix

find products and offers. Consumer search for products and after reviewing the result they may be search with more specific features and reducing the size of the set of potential candidates. They also can check the reviews for each product and also check the profile of the reviewer. After selecting the product, the consumer try to find the best vendor located in his country with a specific delivery time. After choosing the specific offer, he retrieves all the information to save it in local. Listing 4.1 shows the query mix that can be used in this search.

The BSBM benchmark has two representations of mix query: SPARQL representation and SQL representation. In SPARQL representation queries contain parameters which are enclosed with % chars. During a test run, these parameters are replaced with random values from the benchmark dataset.

Variety of SPARQL features, but not all of them were used in BSBM queries.Table 4.2 gives an overview of the characteristics of the BSBM benchmark queries and shows specific SPARQL features that are used by the queries

Characteristic	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Q11	Q12
Simple filters	√		√	√			√	√	√	√		
Complex filters					√	√						
More than 9 patterns		√		√			√	√				
Unbound predicates											√	
Negation			√									
OPTIONAL operator		√	√				√	√				
LIMIT modifier	√		√	√	√			√		√		
ORDER BY modifier	√		√	√	√			√		√		
DISTINCT modifier	√				√					√		
REGEX operator						√						
UNION operator				√							√	
DESCRIBE operator									√			
CONSTRUCT operator			√									√

Table 4.2: Characteristics of the BSBM benchmark queries

The BSBM benchmark uses the following measure in the experiments:

- **Query Mixes per Hour (QMpH)**: it measures the number of complete BSBM query mixes that are answered by system under test within one hour.
- **Queries per Second (QpS)**: measures the number of queries of a specific type that were answered by the system within a second.
- **Load time (LT)**: Cumulative time to load an RDF or relational benchmark dataset from the source file into the system under test.

BSBM run against four popular RDF stores (Sesame, Virtuoso, Jena TDB, and Jena SDB) and two SPARQL-to- SQL rewriters (D2R Server and Virtuoso RDF Views) for three dataset sizes: One million triples, 25 million triples, and 100 million triples. They applied the following test procedure to each store for each dataset size:

- Load the benchmark dataset. The load performance of the systems was measured by loading the two type of representation to the related data stores.
- Shutdown store, clear caches, restart store.
- Execute ramp-up until steady-state is reached to benchmark the systems under normal working conditions.
- Execute single-client test run and measure the query performance of the systems for a single client.

- Execute multiple-client test runs with 2, 4, 8, and 64 clients concurrently working against the system.
- Execute test run with reduced query mix.

The result of these experiments can be found in [Bizer and Schultz, 2009].

4.4 DBpedia SPARQL Benchmark

DBpedia SPARQL Benchmark (DBPSB) is a benchmark which created for heterogeneous data. They apply a novel methodology to datasets of different sizes extracted from the DBpedia knowledge base. DBpedia (version 3.6) for example contains 289,016 classes of which 275 classes belong to the DBpedia ontology. Moreover, it contains 42,016 properties, of which 1335 are DBpedia-specific. Also, various datatypes and object references of different types are used in property values [Morsey et al., 2011].

The DBPSB methodology follows the four key requirements for domain specific benchmarks [Gray, 1993], it is (1) relevant, thus testing typical operations within the specific domain, (2) portable, i.e. executable on different platforms, (3) scalable, e.g. it is possible to run the benchmark on both small and very large data sets, and (4) it is understandable [Morsey et al., 2011].

In DBPSB, in order to create a set of prototypical queries, they perform query analysis and clustering on query logs. after applying clustering, they derive a set of 25 SPARQL query templates. These templates cover most commonly used SPARQL features. To generate the actual benchmark queries, they used these template by parametrization. DBPSB was applied to different RDF stores such as Virtuoso [Erling and Mikhailov, 2007], Sesame [Broekstra et al., 2002a], Jena-TDB [Owens et al., 2008], and BigOWLIM [Bishop et al., 2011], to evaluate the performance of them.

The DBPSB benchmark generated through the two main steps:Dataset generation, and Query analysis and clustering. In the following section we introduce these steps in more details.

4.4.1 Dataset Generation

Generating suitable dataset is one of the critical steps in benchmark creation. DBPSB consider some requirements in order to generate dataset. First of all they require data which resemble the original data as much as possible. The characteristics of the network of data should be similar in different size of the dataset. The process of the data generation should support different size of dataset and should be easily repeatable with new versions of the dataset.

The process of dataset creation need an input dataset. In this work they use DBpedia Datasets. To generate a bigger datasets comparing to original data, they use triples duplication in different scale factor, by changing the namespaces. For generating smaller datasets, they consider two method, first called rand which select an appropriate fraction of all triples of the original dataset randomly, and the second called seed which is based on the assumption that a representative set of resources can be obtained by sampling across classes in the dataset. They performe statistical analysis to select one of these methods for small dataset generation. Comparing the characteristic of data network such as average in-degree, out-degree, and number of nodes between the generated dataset through these methods and the original datasets, they select seed method for dataset generation.

4.4.2 Query Analysis and Clustring

The goal of the query analysis and clustering step is to create prototypical queries which be used to generate actual benchmark queries. The step consists of the following steps:

- **Query Selection**: using the DBpedia SPARQL query-log which contains all queries posed to the official DBpedia SPARQL endpoint for a three-month period from April to July 2010. Around 31.5 million queries were posed to the endpoint during the period. In order to reduce the size of the queries they applied two methods: First, they found the similar queries by renaming the variables by using i.e., var0, var1, var2, and so on. Second, omitting the query with low frequency (below 10), as they do not have affect on the overall query performance. these methods reduce the size of queries to just 35,965.

- **String Stripping**: SPARQL query contains substrings such as PREFIX, SELECT, FROM and WHERE which divide query into different clauses. These substrings generate noise during the computation of query similarity, so by removing these substrings and also common prefixes more accurate result achieve during the next step.

- **Similarity Computation**: In the third step they compute the similarity between segmented queries. Considering Cartesian product of the queries, enormous amount of runtime needed for computing similarity, So they use the LIMES framework [Ngomo and Auer, 2011] to reduce the runtime of the benchmark compilation.

```
SELECT ∗
WHERE {
 { ?v2 a dbp−owl:Settlement ;
 rdfs:label \%\%v\%\% .
 ?v6 a dbp−owl:Airport . }
 { ?v6 dbp−owl:city ?v2 . }
UNION
 { ?v6 dbp−owl:location ?v2 . }
 { ?v6 dbp−prop:iata ?v5 . }
UNION
 { ?v6 dbp−owl:iataLocationIdentifier ?v5 . }
OPTIONAL { ?v6 foaf:homepage ?v7 . }
OPTIONAL { ?v6 dbp−prop:nativename ?v8 . }
}
```

Listing 4.2: Sample query with placeholder

- **Clustering**: The final step is applying graph clustering to the query similarity graph computed in previous step to find very similar groups of queries and generate prototypical queries.

4.4.3 SPARQL Feature Selection and Query Variability

In the next step, a subset of frequently executed queries which cover most SPARQL features was selected. The following SPARQL features considered in selecting queries:

- The overall number of triple patterns contained in the query ($|GP|$),
- The graph pattern constructors UNION (UON), OPTIONAL (OPT),
- The solution sequences and modifiers DISTINCT (DST),
- As well as the filter conditions and operators FILTER (FLT), LANG (LNG), REGEX (REG) and STR (STR).

First of all, they rank the cluster by aggregating the frequency of all queries in the cluster. Then, for each of the SPARQL features, the highest ranked cluster that contain queries having the feature, was selected. Then the query selected from the particular cluster. In this way they choose 25 queries. In order to convert the selected queries in to the query template, a part of the query selected to be varied. Listing 4.2 shows an example of template query which contains place holder indicated by %%v%% or in the case of multiple varying parts %%vn%%. In order to generate various

```
SELECT DISTINCT ?v
WHERE {
 { ?v2 a dbp−owl:Settlement ;
 rdfs:label ?v .
 ?v6 a dbp−owl:Airport . }
 { ?v6 dbp−owl:city ?v2 . }
UNION
 { ?v6 dbp−owl:location ?v2 . }
 { ?v6 dbp−prop:iata ?v5 . }
UNION
 { ?v6 dbp−owl:iataLocationIdentifier ?v5 . }
OPTIONAL { ?v6 foaf:homepage ?v7 . }
OPTIONAL { ?v6 dbp−prop:nativename ?v8 . }
} LIMIT 1000
```

Listing 4.3: Sample auxiliary query returning potential values a placeholder can assume

queries from the query template, a list of all value that can replace the %%v%% place holders can be achieved by executing the query in the listing 4.3. The list of retrieved concert values are used to replace the %%v%% place-holders within the query template.

DBpedia SPARQL Benchmark applied the experiments on the RDF stores Virtuoso, Sesame, Jena-TDB, and BigOWLIM. The configuration and the version of each RDF store can be found in [Morsey et al., 2011]. In order to execute the benchmark, DBpedia datasets with different scale factors, i.e. 10%, 50%, 100%, and 200% load in different RDF stores. The execution consist of the following steps:

- **System Restart**: The RDF stores restart in order to clear memory caches.
- **Warm-up Phase**: Warm-up phase was used to measure the performance of the system in normal operational conditions
- **Hot-run Phase**: The benchmark query mixes were sent to the tested store. The average execution time of each query, and the number of query mixes per hour (QMpH) are computed.

The overall performance of each RDF store was measured by computing its query mixes per hour (QMpH). For query-based performance evaluation the measure Queries per Second (QpS) is used which is computed by summing up the runtime of each query in each iteration,and dividing it by the QMpH value and scaling it to seconds. The detailed result can be found in [Morsey et al., 2011].

Chapter 5

Problem Setting

In this chapter we introduce the work that done during this thesis. First we will present our research question and motivations, followed by success criteria (Section 5.1). In the next section we present the challenges that we have during the work (Section 5.2). finally, we will introduce the basic formalization (Section 6.1).

5.1 Overview

Top-k queries retrieve only the top k query results ordered by a given ranking function which is a combination of the ranking criteria. Search engine is one the example of using Top-k queries. The search engine returns thousands of results, but the user is interested only in the top k most related results. Top-k queries can be easily defined also in the SPARQL language as SELECT queries with an ORDER BY and LIMIT clause. Moreover, It is also possible to use projection in order to define the ranking function.

Top-k queries are gaining more and more attention in the Database and Semantic Web communities. Recent works [Magliacane et al., 2012; Wagner et al., 2012; Cheng et al., 2010; Cedeno, 2010] tried to improve the performance of the Top-k queries by different optimization procedures, but a consistent comparison of those works is not possible. Each of these works relied on their custom data set and query mix for the experimental evaluation. The lack of a SPARQL benchmark covering top-k SPARQL queries is the main reason of this fragmentation. We believe that it is right time for the Semantic Web community to define a top-k SPARQL benchmark. Thus, we decide to introduce a method to automatically generate a Top-k query benchmark for SPARQL Engines to compare their performances.

One of the important requirements that a meaningful benchmark should hold is resembling the reality. Using DBpedia SPARQL Benchmark which used DBpedia knowledge base, and query logs, as a foundation let us to keep close to the approach of resembling reality. The other important requirement is that the benchmark should stress the distinguish features of Top-k queries. From the syntactic perspective, we should use Top-k queries which contain ORDERBY and LIMIT clauses. From the performance perspective, the query mix should include queries with different characteristic, such as different value of k, ranging from low to high selectivity, controlling the correlation of ranking produced by multiple ranking predicates.

Recent works on SPARQL query benchmarks satisfied the first requirement; for instance, the DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] introduces a procedure for benchmark creation based on query-log mining, clustering and SPARQL feature analysis. They also used different various datasets of different sizes extracted from the DBpedia knowledge base. To fulfill the second requirement, we decided to extend DBpedia SPARQL Benchmark with the capabilities required to compare SPARQL engines on Top-k queries.

DBpedia SPARQL Benchmark used extracted data from DBpedia knowledge base to create different size of the datasets. We also use DBpedia as our dataset which has heterogeneity of data. DBpedia SPARQL Benchmark consider dataset with 10, 50, 100, and 200 percentage of the DBpedia dataset. In evaluation due to lack of resources we just consider the 10 percentage of the BDpedia dataset which consist of 15,267,418 triples.

DBpedia SPARQL Benchmark generate its query mix based on query-log mining, and clustering. They consider different SPARQL features such as overall number of triple patterns, graph pattern constructors like UNION, and OPTIONAL, solution sequences and modifiers like DISTINCT, and filter conditions and operators like FILTER, LANG, and REGEX.

We generate the Top-k queries by using the Auxiliary queries used in the DBPSB. In order to create a top-k SPARQL query benchmark, it is necessary to define a process to create scoring function for each query. A scoring function is a combination of ranking criteria. The simplest ranking criterion is a ranking predicate which has an object with rankable data type. It is also possible to define complex ranking criteria. Aggregated ranking criterion is one of the complex ranking criteria. In this type of criteria an aggregation function such as average, count, sum, maximum, and so on is used to compute the value of specific ranking criterion. The other type of ranking criterion can be achieved by some specified calculation between two related objects which have rankable predicate. In addition, it is also possible to consider rankable criteria which produced by inference. In this thesis we consider only the simplest ranking criterion which is ranking predicate.

According to the selectivity of each rankable criterion, different combinations of scoring criterion from different scoring subjects can be selected. It is possible that all the scoring criteria related to one subject or they be selected from different. In order to generate the Top-k queries, various combinations of scoring criteria will be selected, and related ORDERBY and LIMIT clause are added to the Auxiliary queries of DBpedia SPARQL Benchmark. For evaluating the performance of the SPARQL engines we use the same metrics of DBPSB. We define some hypotheses and evaluate them through the experimental results.

5.2 Issues and Limitations

We confront different issues and limitations during the implementation of automated top-k query benchmark. As we mentioned before, we decided to extend the DBpedia SPARQL Benchmark in order to evaluate the Top-k queries. Top-k query return the top k best results according to the predefined scoring function, so, to generate the Top-k query we have to define a scoring function which is a combination of ranking criteria. In this thesis we try to generate Top-k queries from DBPSB Auxiliary queries. In the first step we checked all the Auxiliary queries to find ranking criteria, but most of the queries used in the DBPSB do not have any ranking criteria. In order to overcome this problem we decided to extend the queries and consider all the ranking criteria related to the variables that be used in the Auxiliary queries.

To find the rankable criteria, we changed Auxiliary queries and run them against the Virtuoso SPARQL endpoints [1]. Most of the queries have runtime execution problem and the default maximum execution time is not enough for running the queries. To solve this problem we decided to install the RDF stores locally and load the dataset in each of them. In order to create the Top-k queries we ran some extended query against the Virtouso which installed locally. From 25 Auxiliary queries, 4 queries have the result number equals to zero due to some changes in the structure of the DBpedia. We set the execution time limit equal to 400000 sec and in 6 queries the estimated execution time is exceed the execution time limit and we omit them from the list of the queries.

In this work we just consider the simplest ranking criteria which is rank-

[1]http://demo.openlinksw.com/sparql/

ing predicate. Predicate which has datatype such as Integer, Decimal, Double, Float, Duration, and Time can be used as ranking predicate. We forced to omit the predicate related to the time, because for calculating the scoring function we need normalization and the SPARQL Language does not have any feature that can calculate the difference between two time values. Thus, we just consider the ranking criteria that can be calculated in the scoring function.

5.3 Basic Formalization

In order to introduce our automated process of top-k SPARQL query generation, we need to present some definitions. First of all, we need some basic concept of SPARQL like IRIs (I), Literals (L), Variables (V), Triple Patter, a Graph Pattern (P), a mapping (μ), and the variables occurring in P ($var(P)$). In section 3.3 they are defied in details. In addition we need some definition from SPARQL Rank which represent in section 3.3 in details.

Top-k queries in SPARQL 1.1 can be formulated using a selection expression [2] , and a pair of ORDER BY and LIMIT clauses. In particular, a scoring function is an expression defined over a set on n ranking criteria. A ranking criterion $b(?x_1, \ldots, ?x_m)$ is a function over a set of m variables $?x_j \in var(P)$. A criterion b can be the result of the evaluation of any built-in function of query variables which must have a numerical value. We define as max_b and min_b the application-specific maximal and minimal possible value for the ranking criterion b.

A scoring function on P is an expression $\mathcal{F}(b_1, \ldots, b_n)$ defined over the set B of n ranking criteria. The evaluation of a scoring function $\mathcal{F}$ on a mapping μ, indicated by $\mathcal{F}[\mu]$, is the value of the function when all of the $b_i[\mu]$, where $\forall i = 1, ..., n$, are evaluated. As we mentioned in section 3.3 , it is typical in ranking queries that the scoring function $\mathcal{F}$ is assumed to be monotonic, i.e., a $\mathcal{F}$ for which holds $\mathcal{F}(b_1[\mu_1], \ldots, b_n[\mu_1]) \geq \mathcal{F}(b_1[\mu_2], \ldots, b_n[\mu_2])$ when $\forall i : b_i[\mu_1] \geq b_i[\mu_2]$.

For the scope of this work we defined further constrains on the definition of the SPARQL Rank. First of all, we consider the ranking criterion consisting of a single variable. Second, the scoring function is defined as a weighted sum of normalized ranking criteria each. For this definition, we indicate $w_i \in [0..1]$ a weight and norm as a normalization function in the form of $\frac{b_i - min_{b_i}}{max_{b_i} - min_{b_i}}$. The scoring function is defined in the following form:

[2]For more information on projection functions consult http://www.w3.org/TR/sparql11-query/#selectExpressions

$$\sum_{i=0}^{n} w_i \cdot norme(b_i)$$

In the following, there are some relevant definition to the top-k SPARQL query generation process describe later. Let I_o defined as subset of IRIs representing Object Properties, and I_D defined as subset of IRIs representing Data Properties

Definition 8: A Data Triple Pattern (t_D) is a triple pattern in which the predicate is a member of Data Properties I_D as the following:

$$(s, p, o) \in (I \cup V) \times I_D \times (V \cup L)$$

A set of Data Variable (V_D) of a Data Triple Pattern in a Graph Pattern P is the set of variables in the object position of all Data Triple Patterns in P.

Definition 9: A data property I_D is rankable if its range is either in xsd:int, xsd:long, xsd:float, xsd:integer, xsd:decimal, and xsd:double (which can be all casted in xsd:double) or xsd:dateTime and xsd:date (which can be casted in xsd:dateTime) or xsd:time or xsd:duration.

Definition 10: a set of Rankable Variables (V_R) is the set of V_D that appears in triple patterns where I_D is rankable

Definition 11: a Rankable Triple pattern (t_R) is a Data Triple Pattern whose object variable is a member of Rankable Variables and the subject is a variable as the following:

$$t_R \in V \times I_D \times V_R$$

Definition 12: a Rankable Subject is the variable in the subject position of a Rankable Triple pattern t_R.

Definition13: a set of scoring variables (V_S) is the set of $?x_j \in var(P)$ appearing at least one of ranking criteria b_i in $\mathcal{F}(b_1, \dots, b_n)$.so we have $V_S \subseteq V_R$.

Definition 14: a Scoring Triple Pattern(t_S) is a Data Triple Pattern whose object variable is a member of Scoring Variables and the subject is a variable as the following:

$$t_S \in V \times I_D \times V_S$$

Definition 15: a Scoring Subject is the variable in the subject position of a Scoring Triple pattern t_S.

Chapter 6

Top-k Query Generation

In this chapter we introduce the solution to solve the problem described in previous chapter. This chapter present the process of generating top-k queries from DBpedia Query Templates. First we will introduce the process of top-k query generation consist of finding rankable variable, statistic computation, generating scoring function, and generating new queries(Section 6.1.3). Then, we will describe the work formally including a set of definitions and psuedo code (Section 6.2).

6.1 Top-k Query Generation

In order to automatically generate Top-k query benchmark, we extend the DBpedia SPARQL Benchmark. We use the same dataset, performance metrics, and test driver for our benchmark and create the Top-k queries from the Auxiliary queries of the DBPSB. To generate top-k queries we need to have rankable variable that can be used in the scoring function part of the Top-k queries. Using DBPSB Auxiliary Queries, we generate new queries to find all the rankable predicate that have relation with the variables of the queries. In section 6.1.1 the method will be introduced in details. As we defied the scoring function in section as a weighted sum of normalized ranking criteria, for each ranking variable, we need to find the maximum and minimum possible value to formed the normalization function. So, for each DBPSB auxiliary query and each rankable triple pattern, we generate a query to find the maximum and minimum possible value for each rankable variable. we also compute the selectivity for each rankable variable which will be used in evaluation. Section 6.1.2 will present these computations. Having all the requirement to generate top-k queries, for each DBPSB Auxiliary queries, we randomly generate the scoring function and add ORDERBY and LIMIT cluase to complete the Top-k queries. Figure 6.1 shows an overview of the Top-k query benchmark.

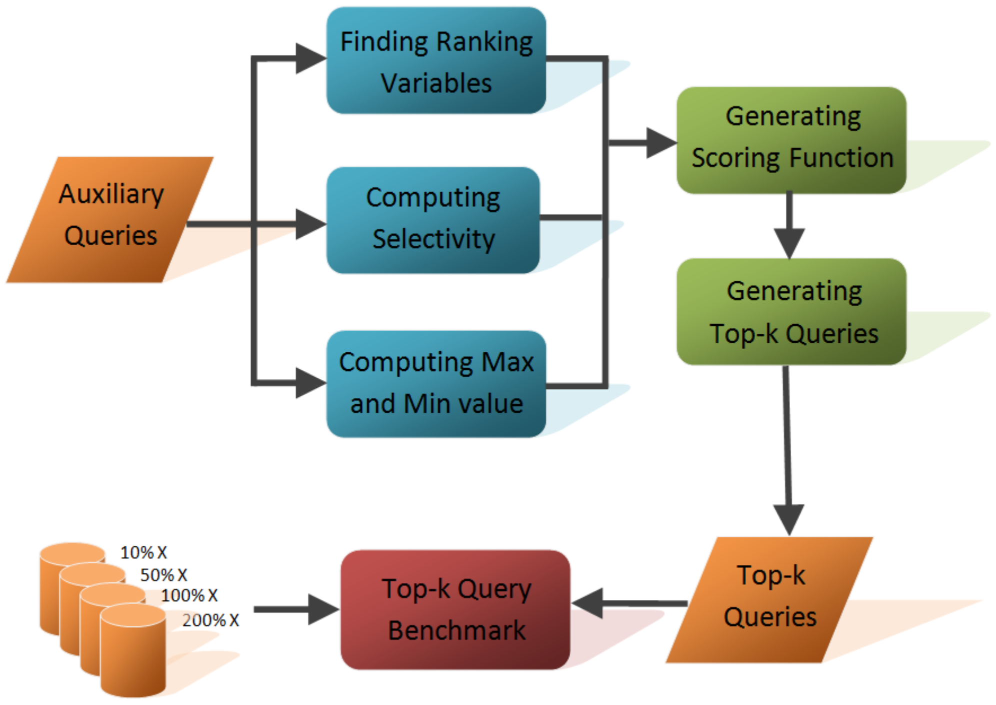


Figure 6.1: Conceptual Model

We describe in Section 6.1.1 how we find rankable variables in DBpedia Auxiliary Queries. In Section 6.1.2 we introduce the statistic computations performed on the set of the DBpedia Auxiliary Queries, and Finally in Section 6.1.3 we represent the generating of Top-k queries in details.

6.1.1 Finding Rankable Triple Patterns

In order to create SPARQL top-k query templates from DBPSB ones, we look for all rankable variables, in each auxiliary query of each BDPSB query template, which could be used as ranking criterion in scoring function of the SPARQL top-k query templates. This also requires collecting some statistics about each rankable variable.

For each DBPSB auxiliary query, to find rankable variables, we first check if the variables in the query are rankable. Notably, out of the 25 DBPSB auxiliary queries, only 4 queries contain rankable variables. For instance, the set of variables in the DBPSB auxiliary query in Listing 6.1 is

```
SELECT DISTINCT ?v
WHERE {
 { ?v2 a dbp−owl:Settlement ;
 rdfs:label ?v .
 ?v6 a dbp−owl:Airport . }
 { ?v6 dbp−owl:city ?v2 . }
UNION
 { ?v6 dbp−owl:location ?v2 . }
 { ?v6 dbp−prop:iata ?v5 . }
UNION
 { ?v6 dbp−owl:iataLocationIdentifier ?v5 . }
OPTIONAL { ?v6 foaf:homepage ?v7 . }
OPTIONAL { ?v6 dbp−prop:nativename ?v8 . }
} LIMIT 1000
```

Listing 6.1: Sample auxiliary query

equal to $V_q = \{?var, ?var0, ?var1, ?var2, ?var3, ?var6\}$. The only rankable variable in V_q is $?var1$ that matches the number of pages. DBpedia dataset contains books which use page number is written as a string, thus, formally speaking, also $?var1$ is not rankable, but this query is the best suited for presentation purposes.

To overcome the problem of existing not sufficient rankable variables in DBPSB auxiliary queries, we extend our search approach. In order to find additional rankable variables, we consider all variables in the query and try to find which of them is the rankable subject of a rankable triple pattern. Finding these rankable triple patterns, we can indicate the variable in the object position as a new rankable variable. For instance, $?var6$ and $?var3$ are two good candidate rankable subjects of the query in Listing 6.1, because they are IRIs.

To find all the rankable variable for each DBPSB auxiliary queries, we extend each query and create a new query for each variable in the auxiliary query. For example the query in Listing 6.2 shows the extension of auxiliary query in Listing 6.1 for variable $?var6$. This query checks if $?var6$ is a rankable subject by looking for rankable predicates and computing the number of results for each rankable predicate. In this query we consider predicates that can be casted to xsd:double.

Figure 6.2 shows that $?var6$ is a rankable variable because there are a tens of rankable predicates for it. We can discover that $?var3$ is also rankable in a similar way.

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT count(?p) ?p
WHERE { ?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
 ?var6 dbpprop:pages ?var1.
 ?var6 dbpprop:isbn ?var2.
 ?var6 dbpprop:author ?var3.
?var6 ?p ?o
FILTER ( isLiteral(?o))
FILTER (datatype (?o) = xsd:int
||datatype (?o) = xsd:float
|| datatype (?o) = xsd:decimal
|| datatype (?o) = xsd:double
|| datatype (?o) = xsd:integer
|| datatype (?o) = xsd:long)
}
Group by ?p
Order by DESC( count(?p))
LIMIT 1000
```

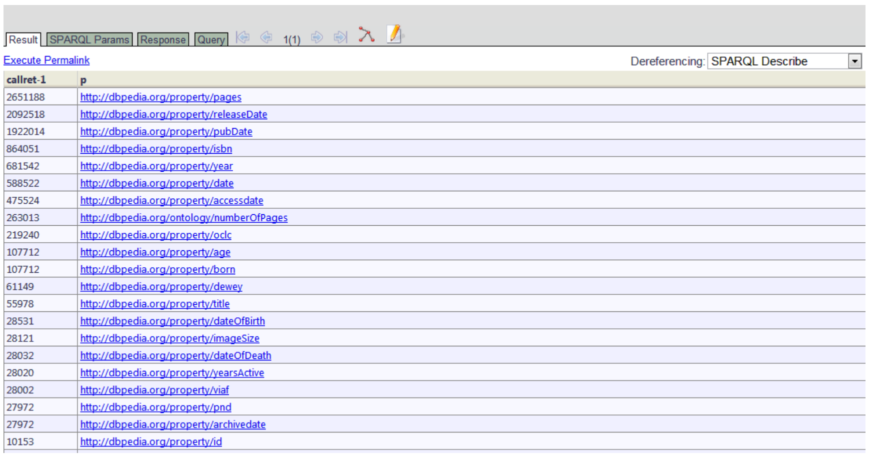
Listing 6.2: New query to find rankable variabe of DBpedia Auxiliary Query.

6.1.2 Statistical Computation

To generate the scoring function, for each rankable triple pattern, we need to compute its selectivity, and maximum and minimum value of the rankable variable. To compute the selectivity of each rankable triple pattern, first we need to change the DBPSB auxiliary queries to find the number of results for each of them. Listing 6.3 shows the modified query for the query in listing 6.1. The execution of this query shows that the Number of the results is equal to 592221.

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT count (?var) AS ?count
WHERE {
?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
?var6 dbpprop:pages ?var1.
?var6 dbpprop:isbn ?var2.
?var6 dbpprop:author ?var3.
}
```

Listing 6.3: Modified DBpedia Auxiliary Query to find number of result.



Dereferencing: SPARQL Describe

callret-1	p
2651188	http://dbpedia.org/property/pages
2092518	http://dbpedia.org/property/releaseDate
1922014	http://dbpedia.org/property/pubDate
864051	http://dbpedia.org/property/isbn
681542	http://dbpedia.org/property/year
588522	http://dbpedia.org/property/date
475524	http://dbpedia.org/property/accessdate
263013	http://dbpedia.org/ontology/numberOfPages
219240	http://dbpedia.org/property/oclc
107712	http://dbpedia.org/property/age
107712	http://dbpedia.org/property/born
61149	http://dbpedia.org/property/dewey
55978	http://dbpedia.org/property/title
28531	http://dbpedia.org/property/dateOfBirth
28121	http://dbpedia.org/property/imageSize
28032	http://dbpedia.org/property/dateOfDeath
28020	http://dbpedia.org/property/yearsActive
28002	http://dbpedia.org/property/viaf
27972	http://dbpedia.org/property/pnd
27972	http://dbpedia.org/property/archivedate
10153	http://dbpedia.org/property/id

Figure 6.2: Rankable variables

To compute the selectivity, two numbers are required. First, the number of the results of the DBPSB auxiliary query, and second, the number of the result of each DBPSB auxiliary query after adding the rankable triple pattern. By dividing these two numbers, you can compute the selectivity of each rankable triple pattern.

For example for rankable triple pattern $(?var6, < http : //dbpedia.org/ontology/publicationDate >, ?o)$, the listing 6.4 shows the query that gives us the second number which is equal to 125625. At this point, we can compute the selectivity of the rankable triple pattern by dividing the number of results of the new query by the number of result of the original auxiliary query. The number of results of the original query is 592221, thus the selectivity of this triple pattern is 21%.

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT count (?var) AS ?count
WHERE {
?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
?var6 dbpprop:pages ?var1.
?var6 dbpprop:isbn ?var2.
?var6 dbpprop:author ?var3.
?var6 <http://dbpedia.org/ontology/publicationDate> ?o
}
```

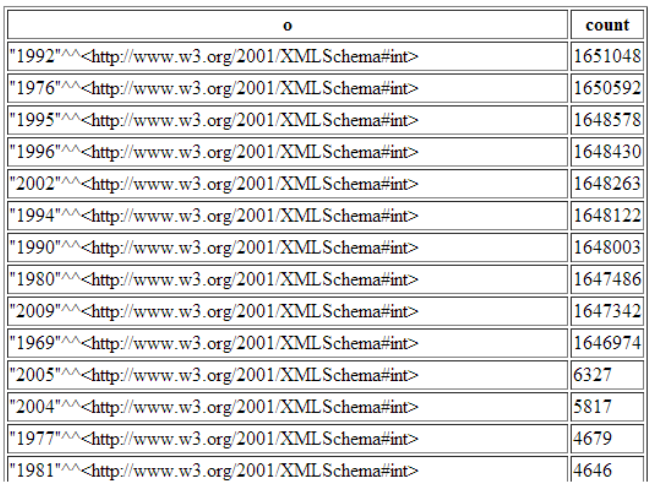
Listing 6.4: DBpedia Auxiliary Query with rankable triple pattern.

As we defied the scoring function as a weighted sum of normalized ranking criteria, for each ranking variable, we need to find the maximum and minimum possible value to formed the normalization function. So, for each DBPSB auxiliary query and each rankable triple pattern, we generate a query to find the maximum and minimum possible value for each rankable variable. Listing 6.5 shows an example query which find the maximum and minimum possible value for the auxiliary query of listing 6.1 and triple pattern $(?var6, dbpprop : releaseDate, ?o)$. The maximum value of $?o$ is 2020-02-01 and the minimum is 1986-03-12.

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT (max(?o) AS ?max) (min(?o) AS ?min)
WHERE {
?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
?var6 dbpprop:pages ?var1.
?var6 dbpprop:isbn ?var2.
?var6 dbpprop:author ?var3.
?var6 dbpprop:releaseDate ?o
FILTER ( isLiteral(?o))
FILTER ( datatype (?o) = xsd:int
|| datatype (?o) = xsd:float
|| datatype (?o) = xsd:decimal
||datatype (?o) = xsd:double
|| datatype (?o) = xsd:integer
|| datatype (?o) = xsd:long)
}
Group by ?p
Order by DESC( count(?p))
LIMIT 1000
```

Listing 6.5: Modified DBpedia Auxiliary Query to find maximum and minimum possible value.

Finally, we compute the distribution of the values matched by each Rankable Variable and create a set on pairs consist of Value and number Values matched by Rankable Variable. Listing 6.6 shows the generated query to obtain the distribution. For each Rankable Variable we find all the value that can map to the variable and find the number of occurrence for each value. Figure 6.3 shows the result of execution of query in Listing 6.6 related to the rankable triple pattern $(?var6, dbpprop : releaseDate, ?o)$.



o	count
"1992"^^<http://www.w3.org/2001/XMLSchema#int>	1651048
"1976"^^<http://www.w3.org/2001/XMLSchema#int>	1650592
"1995"^^<http://www.w3.org/2001/XMLSchema#int>	1648578
"1996"^^<http://www.w3.org/2001/XMLSchema#int>	1648430
"2002"^^<http://www.w3.org/2001/XMLSchema#int>	1648263
"1994"^^<http://www.w3.org/2001/XMLSchema#int>	1648122
"1990"^^<http://www.w3.org/2001/XMLSchema#int>	1648003
"1980"^^<http://www.w3.org/2001/XMLSchema#int>	1647486
"2009"^^<http://www.w3.org/2001/XMLSchema#int>	1647342
"1969"^^<http://www.w3.org/2001/XMLSchema#int>	1646974
"2005"^^<http://www.w3.org/2001/XMLSchema#int>	6327
"2004"^^<http://www.w3.org/2001/XMLSchema#int>	5817
"1977"^^<http://www.w3.org/2001/XMLSchema#int>	4679
"1981"^^<http://www.w3.org/2001/XMLSchema#int>	4646

Figure 6.3: Distribution Result of DBpedia Auxiliary Query

6.1.3 Top-k Query Generation

After computing those statistics for each rankable triple pattern, we can create the new top-k query template. First, we have to generate the scoring function that is defined as a weighted sum of normalized ranking criteria. In order to choose the ranking criteria we should consider the following issues:

- Selectivity: we consider different selectivity level for the ranking criteria. The selectivity can be "very low"(0% - 20%) , "low"(21% - 40%) , "medium"(41% - 60%) , "high"(61% - 80%) , and "very high"(81% - 100%). We can select ranking criteria from a specific selectivity level or choose mix selectivity

- Number of scoring variables: the number of the selected scoring variables can be between 1 and 3.

- Number of scoring subjects: the scoring variables can be selected from 1 to 3 scoring subjects.

- List of weights: according to the number of scoring variables a list of weights are generated randomly which their sum is equal to 1.

The scoring function is generated by considering maximum and minimum possible value for each scoring variable $v_i \in V_S$ using a normalization function of the form $\frac{v_i - min_{v_i}}{max_{v_i} - min_{v_i}}$, and weights related to each scoring variable.

```
PREFIX rdf: <http://www.w3.org/1999/02/22−rdf−syntax−ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
SELECT ?o (count(?o) AS ?count)
WHERE {
?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
?var6 dbpprop:pages ?var1.
?var6 dbpprop:isbn ?var2.
?var6 dbpprop:author ?var3.
?var6 dbpprop:releaseDate ?o
FILTER ( isLiteral(?o))
FILTER ( datatype (?o) = xsd:int
|| datatype (?o) = xsd:float
|| datatype (?o) = xsd:decimal
||datatype (?o) = xsd:double
|| datatype (?o) = xsd:integer
|| datatype (?o) = xsd:long)
}
Group by ?o
Order by DESC( count(?o))
LIMIT 1000
```

Listing 6.6: Modified DBpedia Auxiliary Query to find distribution of values matched by Rankable Variable $?var6$.

For instance, for the query in listing 6.1 a valid scoring function can ask to rank pairs of books and authors by a weighted sum of the normalized date of birth of the author and the normalized date of publication of the book.

In order to obtain an executable query, for each scoring variable that appears in scoring function we should add the related scoring triple pattern to the body of the top-k query template. It is also needed to add the scoring subjects to the SELECT clause.The LIMIT clause randomly assume one of 10, 100, and 1000. For instance the query in listing 6.7 shows one of the queries that our method generates.

6.2 Formal Description of Top-k Query Generation

In this subsection we introduce the formal description of the process of top-k query generation. In order to present the pseudo code of the process, we need to explain some definitions and functions which are used in the pseudo code. Here is the list of the symbols that are used in the pseudo code:

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX umbelBus: <http://umbel.org/umbel/sc/Business>
PREFIX umbelCountry: <http://umbel.org/umbel/sc/IndependentCountry>

SELECT ?var4 ?var ( ( 0.21* ( ?o0 - xsd:double("1.10999e+10") )
/ ( xsd:double("1.10999e+10") - xsd:double("1.10999e+10") ) )
+ ( 0.79* ( ?o1 - xsd:date("1957-05-02") )
/ ( xsd:date("2005-02-07") - xsd:date("1957-05-02") ) )
AS ?score )

WHERE {
 ?var4 <http://dbpedia.org/ontology/areaMetro> ?o0 .
  ?var <http://dbpedia.org/ontology/formationDate> ?o1 .

  { ?var dbpedia2:subsid ?var3 OPTIONAL{ ?var2 ?var dbpedia2:parent }
 OPTIONAL{ ?var dbpedia2:divisions ?var4 }}
 UNION { ?var2 ?var dbpedia2:parent
 OPTIONAL{ ?var dbpedia2:subsid ?var3 }
 OPTIONAL{ ?var dbpedia2:divisions ?var4 }}
 UNION { ?var dbpedia2:divisions ?var4
 OPTIONAL{ ?var dbpedia2:subsid ?var3 }
 OPTIONAL{ ?var2 ?var dbpedia2:parent }} }

ORDER BY ?score
LIMIT 100
```

Listing 6.7: Generated Top-k query

- Q: Set of all the DBPSB query templates
- R_q: The result number of a query $q \in Q$
- V_q: All the variables of a query $q \in Q$
- RT_q: All the rankable triple patterns of a query $q \in Q$
- R_q^v: The result number of a query $q \in Q$ considering rankable variable v of a rankable triple pattern
- ST_q: Scoring Triple Patterns of a query $q \in Q$
- $D_{v_q^r}$: Distribution of values matched by ranking variable v_q^r of query $q \in Q$
- $ST = \{(i\ rankablevariable, j\ rankablesubject) : 1 \leq i, j \leq 3\ and\ i \leq j\}$: All the possible combination of the rankable variable and rankable subject for defining scoring function. For the scope of this work the number of scoring variables ranges from 1 to 3 and the number of objects ranges between from 1 to 3.
- W: set of the weights required for defining scoring function.
- $L = \{1, 10, 100, 1k, 10k\}$: Set of limits.

In the following there is a list of the functions and their definitions, using these functions and the previous definitions we wrote the pseudo code of the process:

- $RN(q)$: given a query $q \in Q$, returns the number of result of the query
- $RTP(v_i)$: given a rankable variable v_i of query $q \in Q$ returns the rankable triple pattern contains
- $RTPS(q, v_i)$: given a query $q \in Q$, and a rankable variable v_i from the query , creates a new query and returns one or more rankable triple patterns which have rankable variable v_i in their subject positions
- $SUB(rt)$: given a rankable triple pattern rt, return the rankable subject
- $PRE(rt)$: given a rankable triple pattern rt, return the predicate
- $OBJ(rt)$: given a rankable triple pattern rt, return the object

- $MIN(q, rt)$: given a query $q \in Q$, and a rankable triple pattern rt, create a new query to return maximum value matched by ranking variable which is in the object position of the rt.
- $MAX(q, rt)$: given a query $q \in Q$, and a rankable triple pattern rt, create a new query to return minimum value matched by ranking variable which is in the object position of the rt.
- $DIS(q, rt)$: given a query $q \in Q$, and a rankable triple pattern rt, create a new query to return the distribution of values matched by ranking variable which is in the object position of the rt.
- $SFW(st)$: given a possible combination of the rankable variable and rankable subject $st \in ST$, returns a set of Weights
- $SF(SV_q, st, W)$: given a set of scoring variables SV_q , given a possible combination of the rankable variable and rankable subject st, and set of weights W, returns the scoring function Formula
- $Limit(L)$: select randomly a limit from set L
- $TOPKQT(q, score, l)$: given a query $q \in Q$, the scoring function formula score, and the Limit l, returns a top-k query template

The pseudo code of the process of Top-k query generation is introduced in algorithm 1.

Algorithm 1 Top-k query generation pseudo code

```
1: procedure TopKQueryGeneration($Q$)
2:     for all $q \in Q$ do
3:         $R_q \leftarrow RN(q)$
4:         for all $v_i \in V_q$ do
5:             if $v_i$ is rankable variable then
6:                 $RT_q \leftarrow RT_q \cup RTP(v_i)$
7:             else
8:                 $RT_q \leftarrow RT_q \cup RTPS(q, v_i)$
9:             end if
10:        end for
11:        for all $rt_i \in RT_q$ do
12:            if $!sub(rt_i) \in V_q$ then
13:                $R_q^v \leftarrow NR(vq)$
14:                $Selectivity(rt_i) \leftarrow R_q^v / R_q$
15:            end if
16:            $min \leftarrow MIN(q,\ rt_i)$
17:            $max \leftarrow MAX(q,\ rt_i)$
18:            $D_{v_i q^r} \leftarrow DIS(q,\ rt_i)$
19:            $SV_q \leftarrow SV_q \cup \{(rt_i, min, max)\}$
20:        end for
21:        for all $st \in ST$ do
22:            $W \leftarrow SFW(st)$
23:            $score \leftarrow SF(SV_q, st, W_i)$
24:            $l \leftarrow Limit(L)$
25:            $qt \leftarrow TOPKQT(q, score, l)$
26:        end for
27:    end for
28: end procedure
```

Chapter 7

Experimental Evaluation

In this Chapter we will discuss an experimental evaluation of automatically generated Top-k query benchmark. First we will describe the environment setting for the experiment (Section 7.1). Second, we will represent the experimental strategy (Section 7.2), and finally, we will describe a series of experimental results that compare the performance of SPARQL engines including Virtouso, Jena TDB, Sesame, and ARQ-Rank considering Top-k query parameters such as Object Number, Predicate Number, Selectivity, and Limit value (Section 7.3).

7.1 Experimental Setting

In order to evaluate the Top-k Query Benchmark, we extend the DBpedia SPARQL Benchmark test driver to run Top-k queries against different SPARQL engines.

The testing environment configure on a Intel Core i7 @ 1.8 GHz with 4 GB Memory and 250 GB hard disk capacity. The operating system is Mac OS X Lion 10.7.5 and Java 1.6.0_51 is installed on the machine. We carry out our experiments by using the SPARQL engines Virtuoso, Sesame, Jena-TDB, and ARQ-Rank. The configuration and the version of each SPARQL engine are as follows:

- Virtuoso Open-Source Edition version 6.1.6: we set the following memory related parameters in file named "‘virtouso.ini"’. NumberOfBuffers = 340000, MaxDirtyBuffers = 250000 and set the maximum execution time equals to 400000 sec.

- Sesame Version 2.7.2 with Tomcat 7.0.41 as HTTP interface: We use the native storage layout and set the spoc, posc, opsc indices in the native storage configuration.

- Jena-TDB Version 2.10.1: We use the default configuration.
- ARQ-Rank: We set the Maximum Java heap size to 2048 MB.

7.2 Experimental Methodology

After preparing the experimental environment, in the next step we load the DBpedia dataset with the scale factors of 10% on all the mentioned SPARQL engines. In order to generate the Top-k queries we run our benchmark generator against the Virtuoso. As mentioned in the Section 6.1.3 we found the ranking variable for the 25 DBpedia Benchmark queries. The list of the DBpedia Benchmark queries are available in 8.1.

The query number 2, 5, 21, and 24 have the result number equal to zero which cause the divided by zero problem in generating of scoring function, thus we omit them from our work. On the other hand, during the computation of statistic information for some of the queries, we have execution time problem. The estimated execution time of the query number 4, 13, 18, 20, and 24 exceed the maximum execution time. Due to large execution time we do not have all the data related to these queries. Running DBpedia Top-k query generator, we generate Top-k queries with different selectivity value which discus in Section 6.1.3. Table 7.1 shows some information about the generated Top- queries. In the rest of this document we will use the new query ID which are between 1 to 21.

In order to evaluate the performance of SPARQL engines, we use the DBpedia SPARQL Benchmark test driver and modify it for Top-k queries.

We create a class named TopKQuery in order to store the information related to each Top-k query. The following code shows a part of the TopKQuery class which contain the properties.

```
public class TopKQuery {
 private String query ;
 private int dbQueryId ;
 private int objectNumber ;
 private int predicateNumber ;
 private int selectivityType ;
 private int limit ;
...
}
```

DBPSB Query ID	Top-k Query ID	Number of Objects	Number of predicates	Number of Selectivity Type	Number of Limits
1	1	1	3	2	4
2	-	-	-	-	-
3	2	1	3	1	4
4	3	1	2	5	4
5	-	-	-	-	-
6	4	1	3	1	4
7	5	3	3	1	4
8	6	1	3	5	4
9	7	3	3	5	4
10	8	3	3	4	4
11	9	3	3	1	4
12	10	3	3	4	4
13	11	1	3	5	4
14	12	1	3	1	4
15	13	1	3	2	4
16	14	1	3	2	4
17	15	1	3	1	4
18	16	1	3	5	4
19	17	1	3	2	4
20	18	1	3	5	4
21	-	-	-	-	-
22	19	1	3	2	4
23	20	1	3	1	4
24	-	-	-	-	-
25	21	3	3	1	4

Table 7.1: Top-k query statistics

The property query contains the text of the Top-k query which is generated automatically in the previous step. dbQueryId shows the ID of the Auxiliary query in DBpedia SPARQL Benchmark which has the value between 1 to 25. The two properties objectNumber, and predicateNumber are related to the scoring function of the Top-k query and they show the number of predicates and objects that these predicates are related to them. The selectivityType shows the type of selectivity of the predicates that appear in the scoring function of the Top-k query. As we mentioned in Section 6.1.3 the selectivity is computed for each predicate and has one of the following values:“very low”, “low”, “medium”, “high”, and “very high”. The predicates of the scoring function can be selected from each of these selectivity levels or can be chosen from different levels. In the second case the selectivity type is mixed. The last property shows the value of the LIMIT clause in the Top-k query.

DBpedia SPARQL Benchmark define SPARQLQuery to store the information of the Auxiliary query such as query text, variables, and the values of variables. As our work is the extension of the DBPSB to have Top-k queries, we change this class according to our needs which shows in the following Listing:

```
public class SPARQLQuery {
 private String query ;
 private int queryId ;
 private ArrayList<TopKQuery> topLQueries ;
...
}
```

Each SPARQLQuery contains a main query, Query ID, and a list of Top-k queries generated from the main query. During the execution phases Top-k query is selected randomly from this list to run against the SPARQL engines.

In order to execute the queries, DBpedia SPARQL Benchmark contains an interface named QueryExecutor. This interface has two functions, executeQuery, and executeWarmUpQuery which run the query in main and warmup phases. For each SPARQL engine there exists a class which implement executeQuery and has special configurations in order to connect to the SPARQL engine. We change the following implementations to make them compatible with our work:

- JenaQueryExecutor
- SesameQueryExecutor
- VirtuosoQueryExecutor

We also implement the class ARQRankQueryExecutor to execute the queries against the ARQ Rank. In DBpedia SPARQL Benchmark the function executeQuery returns the execution time of the query. As we need the parameters of each top-k query to evaluate our work we change this function to return the execution time with the top-k query parameters. The warmup phases takes 10 minutes and the main execution phases takes 30 minutes.

After the execution we have the parameters and the execution time of each executed Top-k query in a specific file. The information gains from the execution is used to evaluate the following hypotheses:

- The more number of objects in the scoring function, the more average execution time
- The more number of Predicates in the scoring function, the more average execution time

- The more selectivity of the predicates in the scoring function, the less average execution time
- The value of the Limit has not any significant effects on the average execution time due to the materialization-the-sort approach in SPARQL engines except ARQ Rank.

In the following Section we will compare the execution times of the queries and evaluate our hypotheses.

7.3 Experimental Result

As we mentioned before, we configure the DBpedia SPARQL Benchmark test driver to execute warmup phases for 10 minutes and the main execution phases for 30 minuts in which the queries are executed in sequence. Figure 7.1 shows the result of the experiment for different SPARQL engines. The horizontal axis shows the Query ID and the vertical axis shows the average of execution time. During the execution some of the queries exceed the maximum execution time and we omit them before calculating the average execution time. Our experiments shows that ARQ Rank and Jena have better performance in the execution time of the Top-k queries.

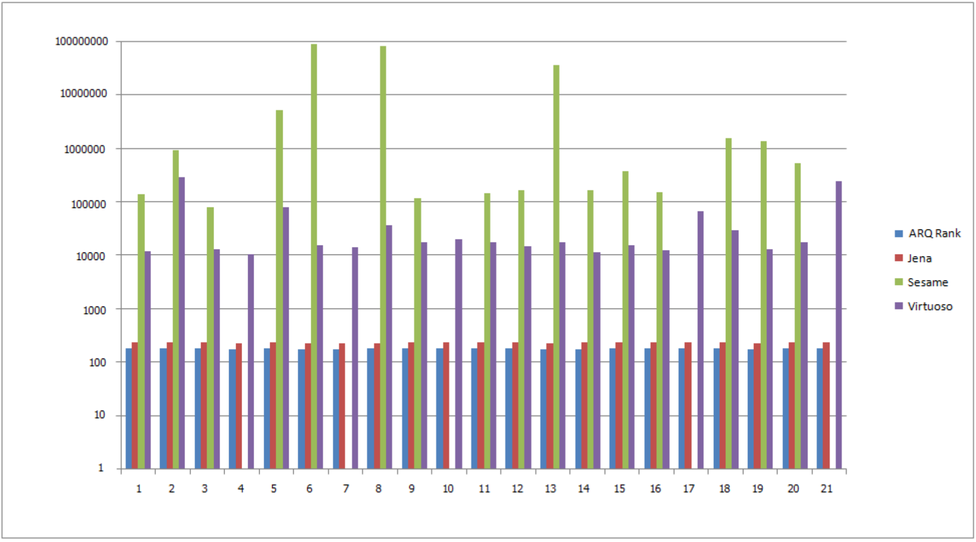


Figure 7.1: Comparing Overall Performance

We consider the following parameters of the Top-k query and compute the average value of the execution time for different parameters. In the following sections we will represent the result for these parameters.

- Number of objects
- Number of predicates
- Selectivity
- Limit

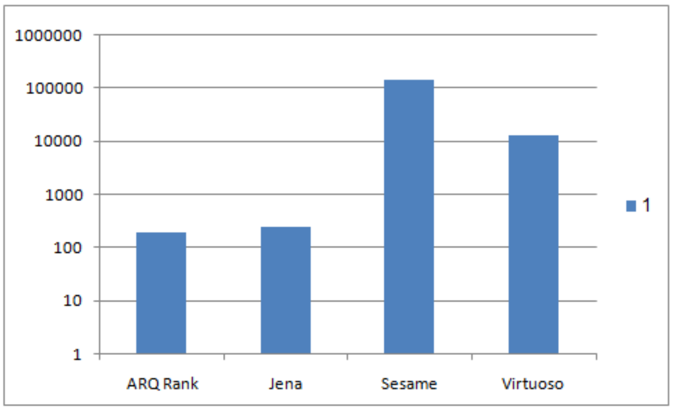
7.3.1 Number of Objects

The result of the testing with the test driver are shown in Figure 7.2. There exists a chart for each main query which indicate by Query ID. The names of the SPARQL engines come in the horizontal axis and the vertical axis shows the execution time in logarithmic scale. The object number shows with different color columns.

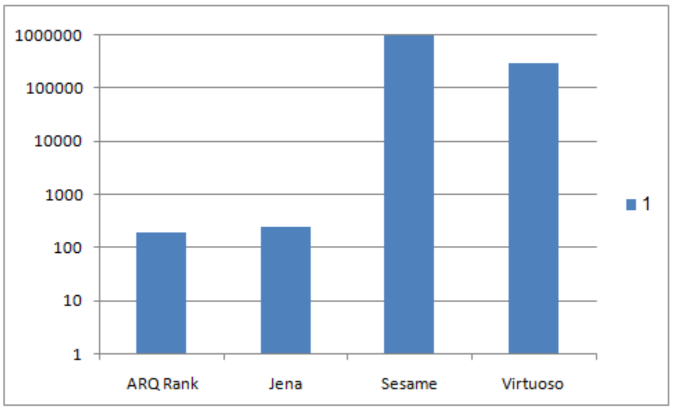
Reviewing the charts shows that in general most of the queries has only one object because all their rankable predicates are related to one rankable variable. Thus during the generating of the Top-k query all the predicates which be used in the scoring function are from one object.

The queries that can be evaluated are query numbers 5, 7, 8, 9, 10, and 21. According to our hypothesis, by increasing the number of objects we expect to have increasing in the execution time. Aggregating by query, the results indicate that query numbers 8, and 9 have the positive behavior and our hypothesis validate in this cases (Figure 7.2(h), 7.2(i)), but query numbers 5, 7, 10, and 21 have negative behavior (Figure 7.2(e), 7.2(g), 7.2(j), 7.2(u)). So we can not evaluate our hypothesis as true in general.

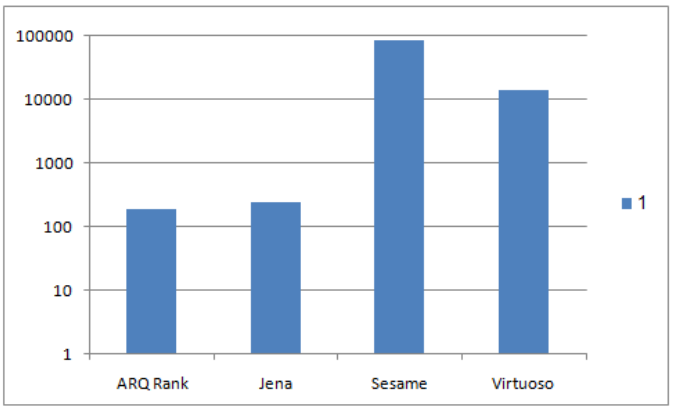
Aggregating by SPARQL engine, the results show that in ARQ Rank and Jena the behavior of the queries are compatible with our assumption. The only exception is query number 7 (Figure 7.2(g)) which need more analysis. The behavior of the Virtuoso in most of the cases are negative and only in query numbers 7, and 10 are positive. In Sesame, due to long execution time, we do not have enough executed query and in some cases we do not have any data (Figure 7.2(g)), or only have one column (Figure 7.2(e)), So we can not evaluate the assumption in Sesame.



(a) Query 1

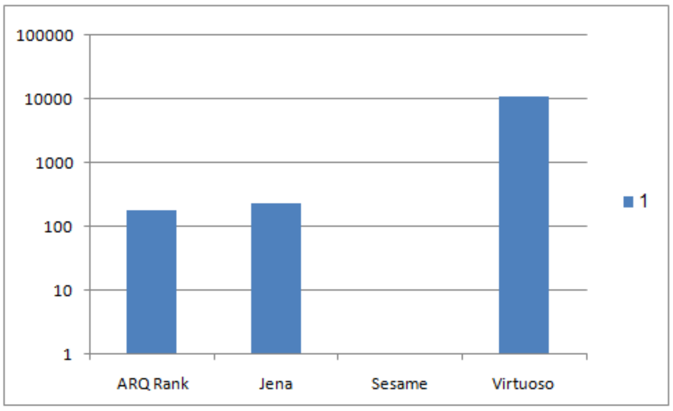


(b) Query 2

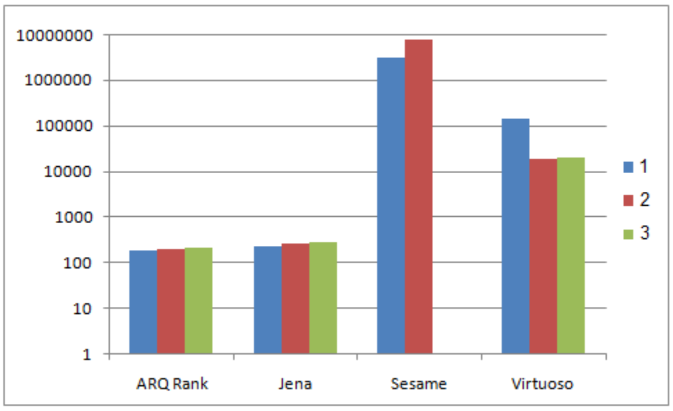


(c) Query 3

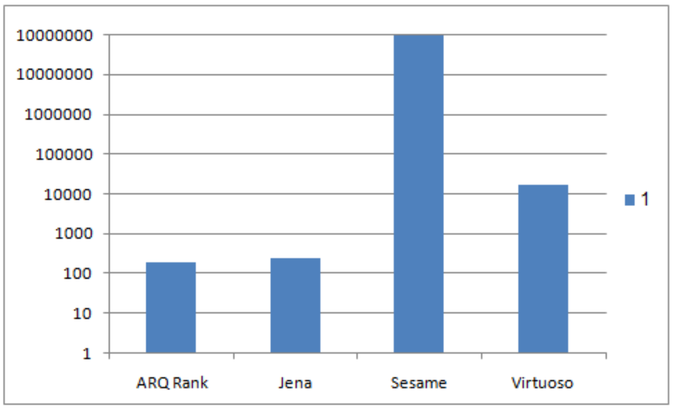
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(d) Query 4

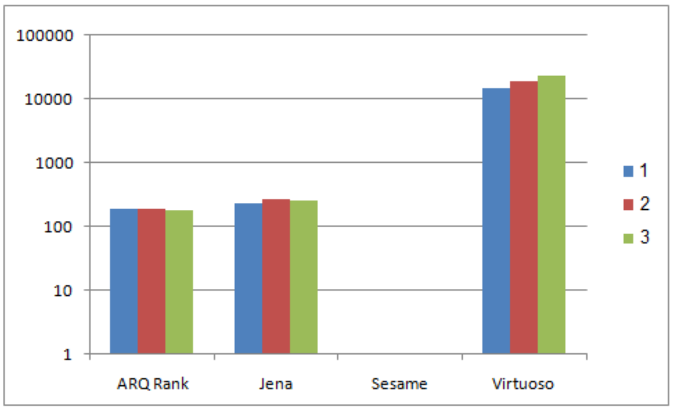


(e) Query 5

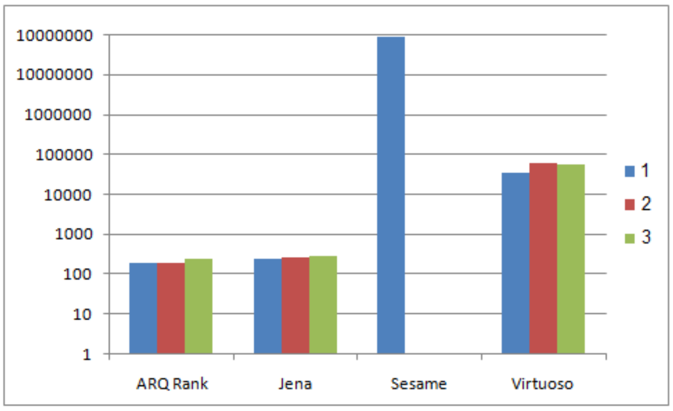


(f) Query 6

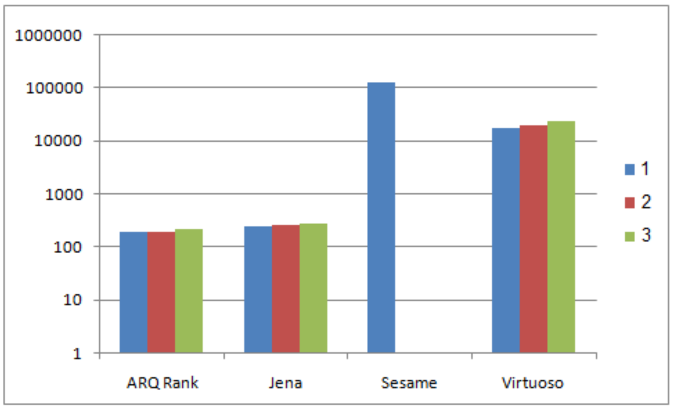
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(g) Query 7

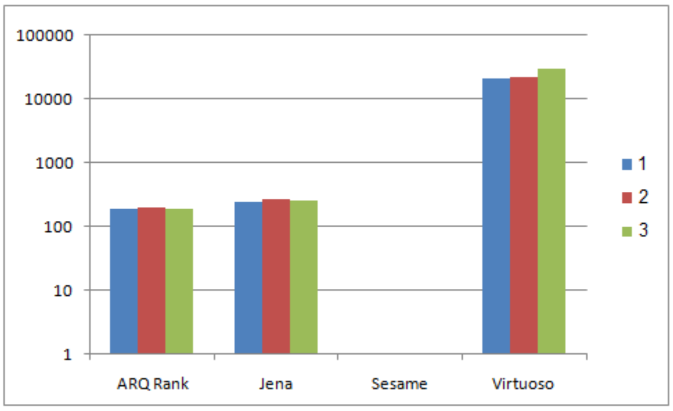


(h) Query 8

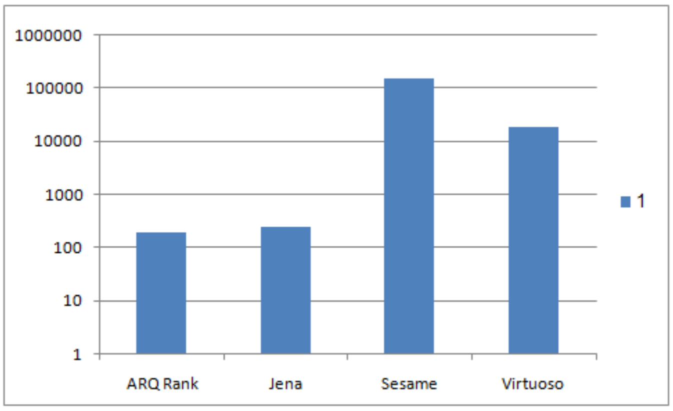


(i) Query 9

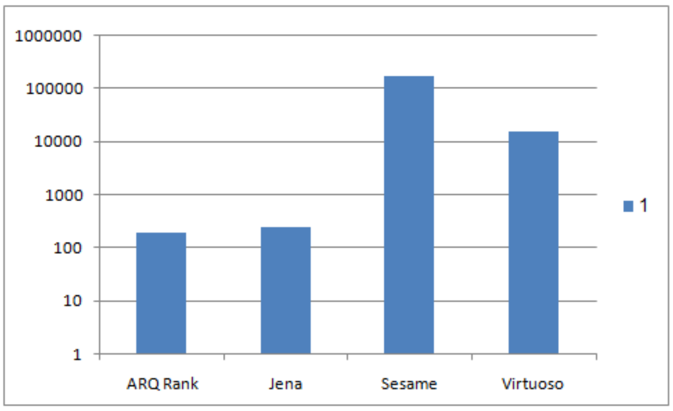
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(j) Query 10

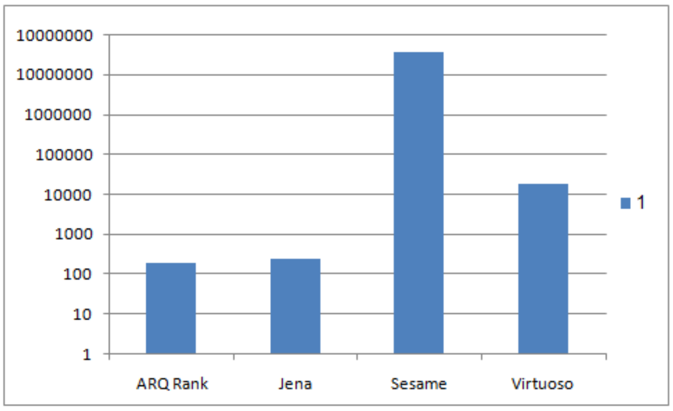


(k) Query 11

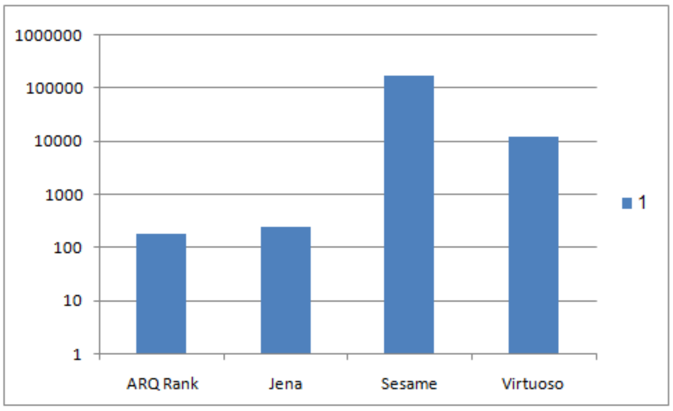


(l) Query 12

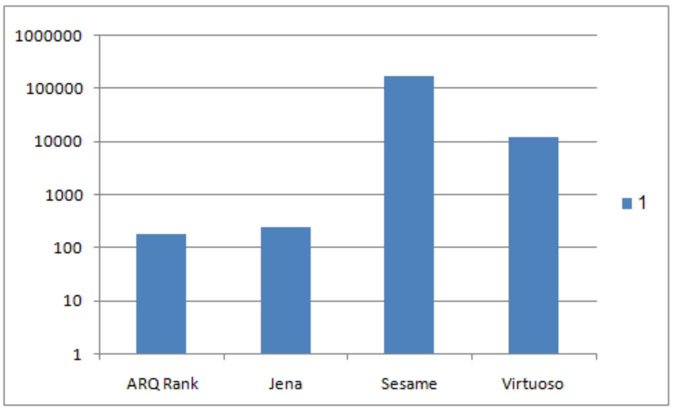
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(m) Query 13

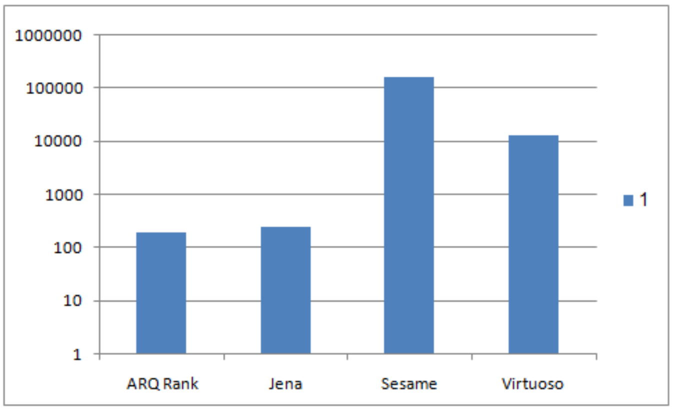


(n) Query 14

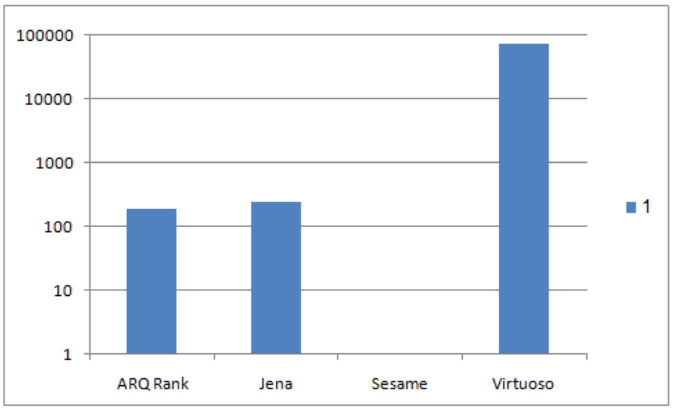


(o) Query 15

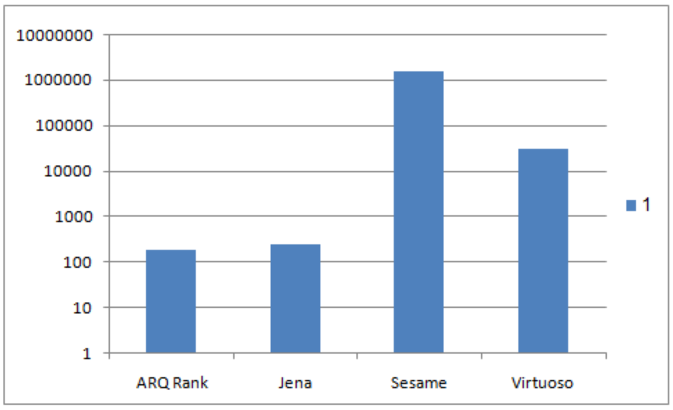
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(p) Query 16

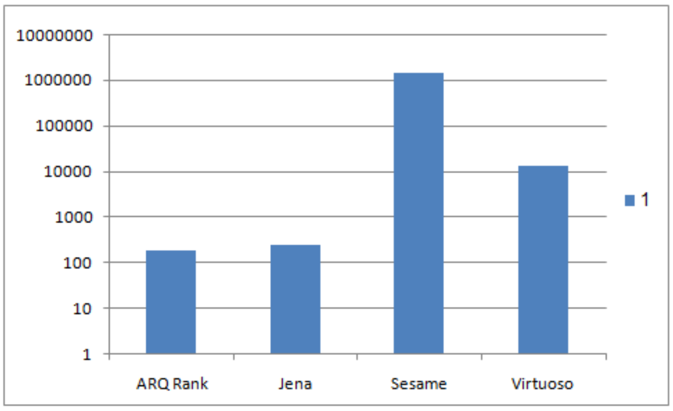


(q) Query 17

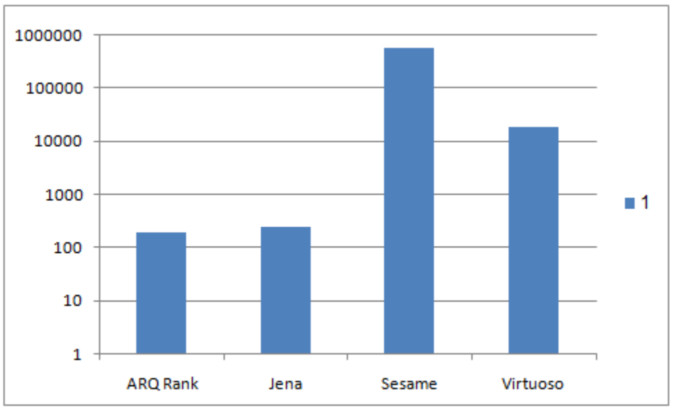


(r) Query 18

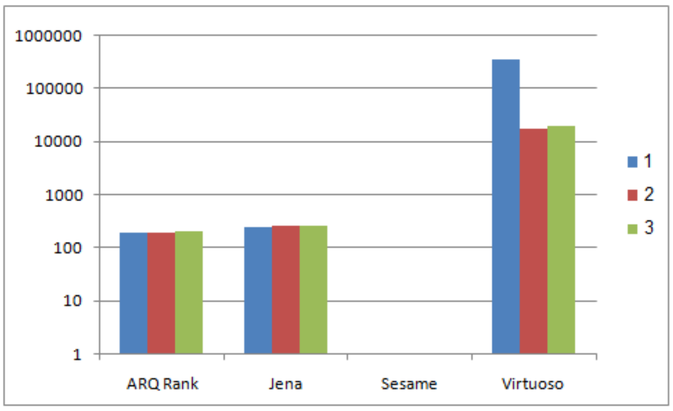
Figure 7.2: Execution Time of the Query according to different Object Number (cont.)



(s) Query 19



(t) Query 20



(u) Query 21

Figure 7.2: Execution Time of the Query according to different Object Number

7.3.2 Number of Predicates

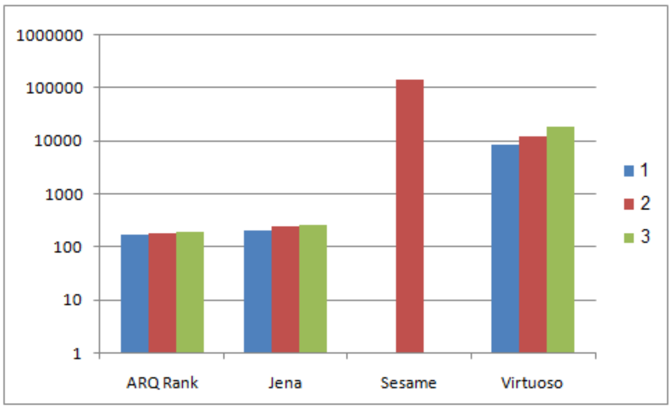
The results of the testing with the test driver are shown in Figure 7.3. For each main query there exists a chart which indicates by Query ID. The names of the SPARQL engines come in the horizontal axis and the vertical axis shows the execution time in logarithmic scale. The predicate number shows with different color columns.

Reviewing the charts shows that in general most of the queries' behavior are compatible with our hypothesis and the increasing the number of predicates cause increasing the execution time. However in some cases, this hypothesis is not correct.

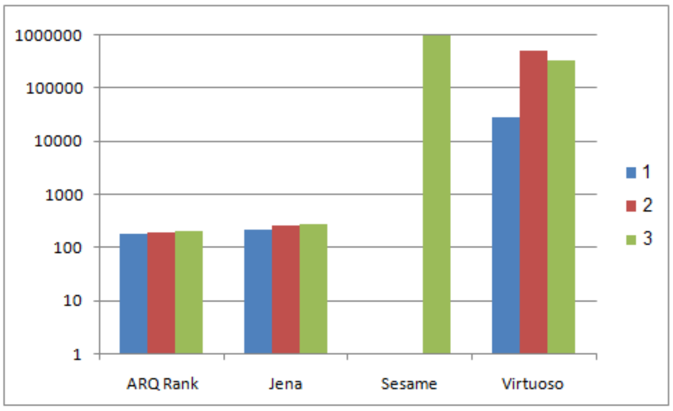
Aggregating by query, the results show that in 13 queries, the behaviors of the executed queries are compatible with our hypothesis in all the SPARQL engines. In some cases it is not possible to evaluate the result of the Sesame due to lack of enough data. The execution time of queries 2 (Figure 7.3(b)), 5 (Figure 7.3(e)), 17 (Figure 7.3(q)), 18 (Figure 7.3(r)), and 21 (Figure 7.3(u)) are against our hypothesis in Virtuoso. Queries 9 (Figure 7.3(i)), 14 (Figure 7.3(n)), and 16 (Figure 7.3(p)) also are against our hypothesis in Sesame. These queries need more investigation to find the reason of such behavior.

Aggregation by SPARQL engine, the results indicate that ARQ Rank, and Jena are compatible with our hypothesis in all cases and by increasing the predicate number, the execution time will increase. Due to long execution time in Sesame, we do not have enough executed query and in some cases we do not have any data (Figure 7.3(d)), or only have one column (Figure 7.3(a)). In the remaining cases from 8 queries, 5 queries are compatible with our hypothesis and 3 queries are against it. In Virtuoso five cases are against our hypothesis, but the remaining cases validate out hypothesis.

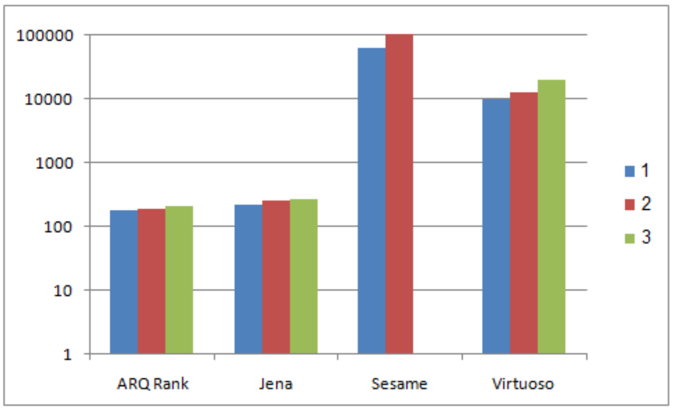
According to the result of the figure 7.3 we can say that our hypothesis is validated (87 %) and by adding more predicates in the scoring function the query needs more time for the execution.



(a) Query 1

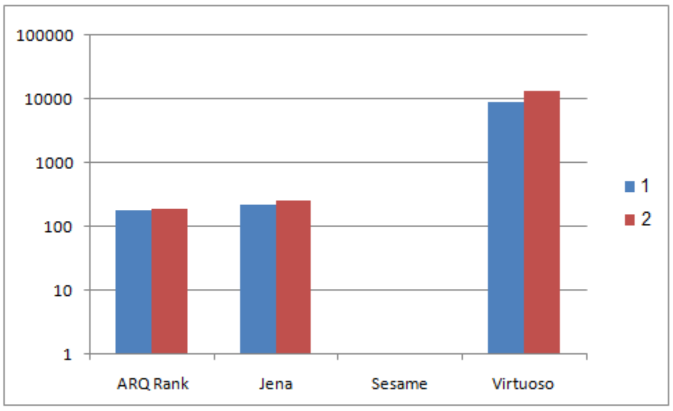


(b) Query 2

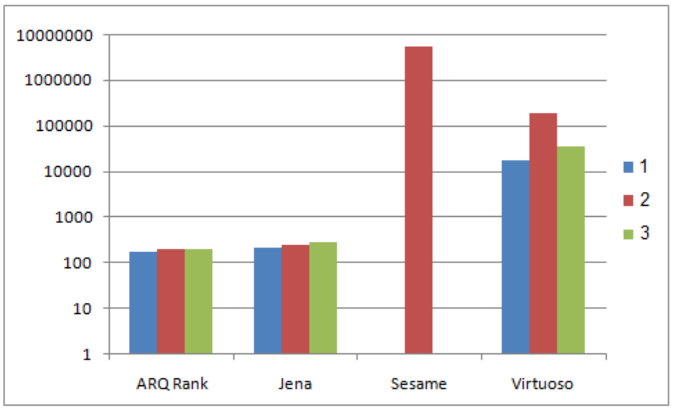


(c) Query 3

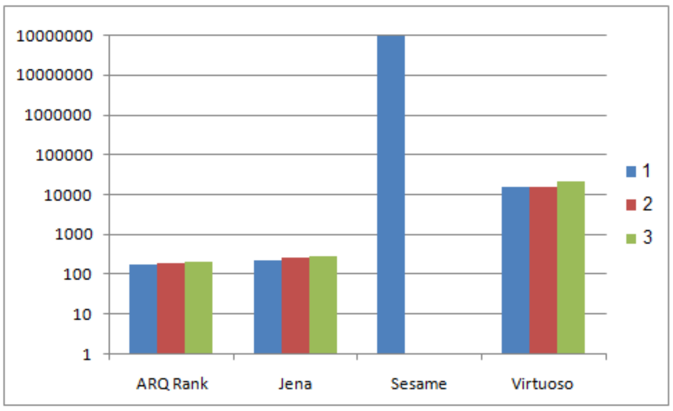
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(d) Query 4

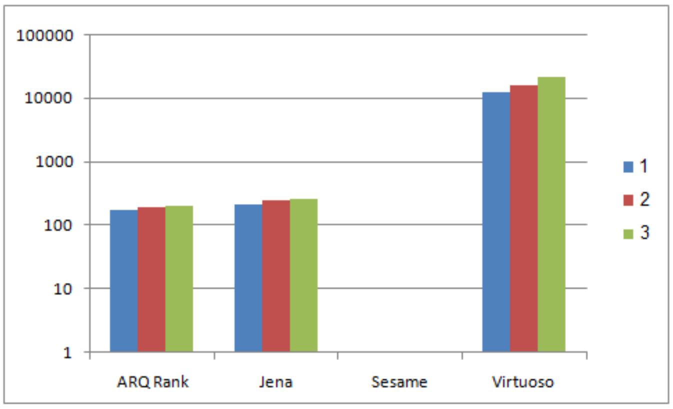


(e) Query 5

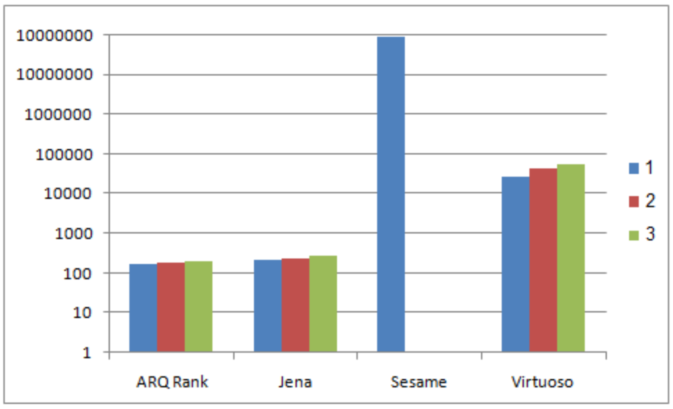


(f) Query 6

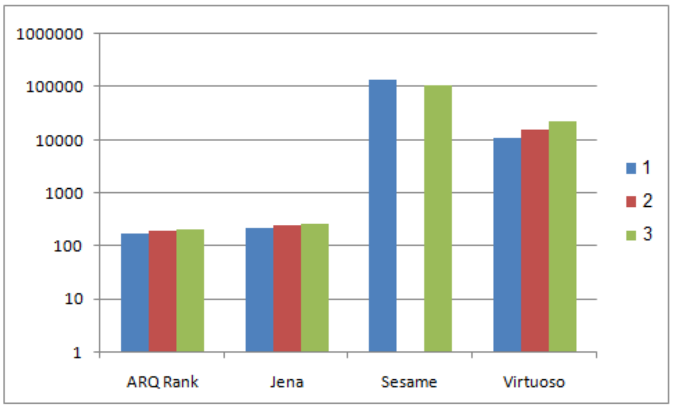
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(g) Query 7

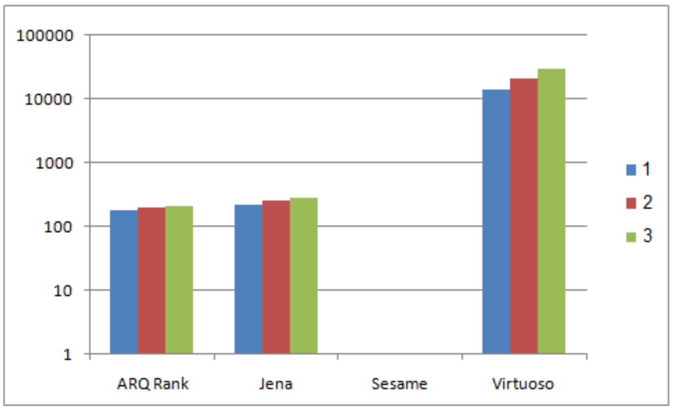


(h) Query 8

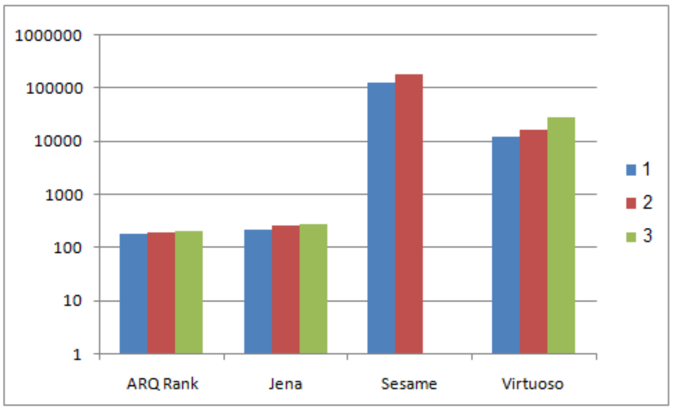


(i) Query 9

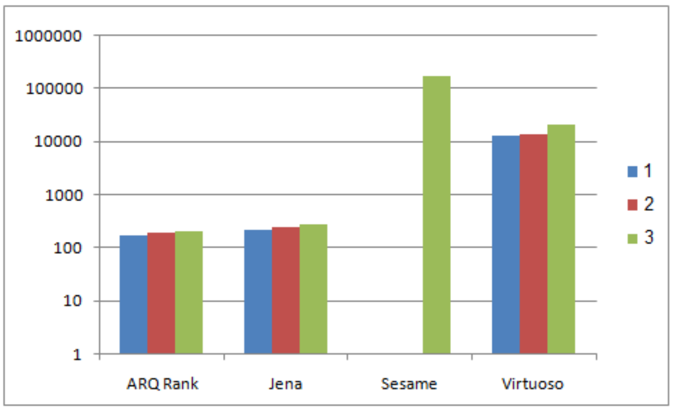
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(j) Query 10

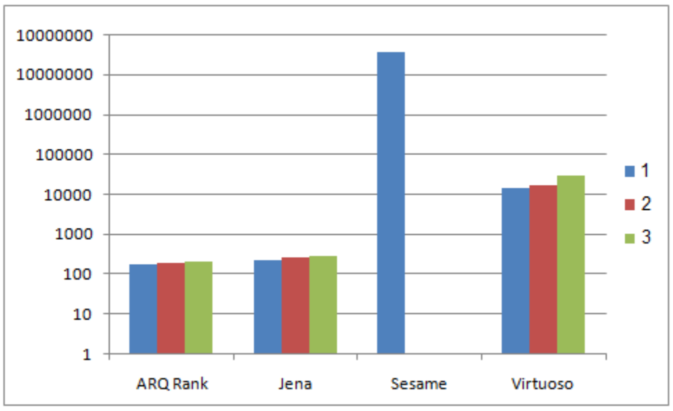


(k) Query 11

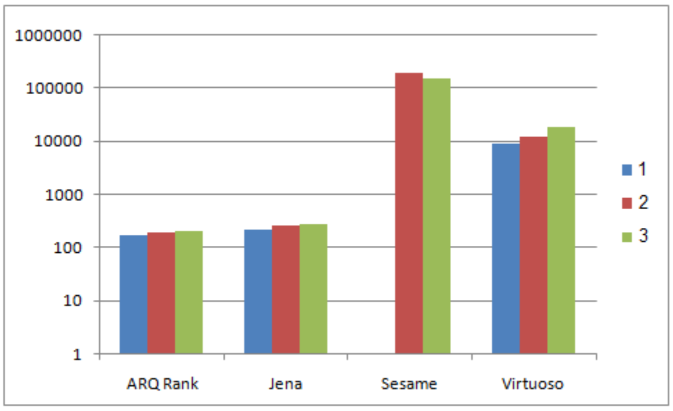


(l) Query 12

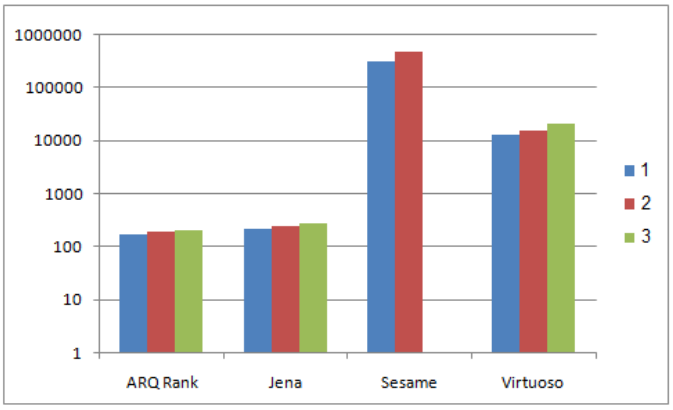
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(m) Query 13

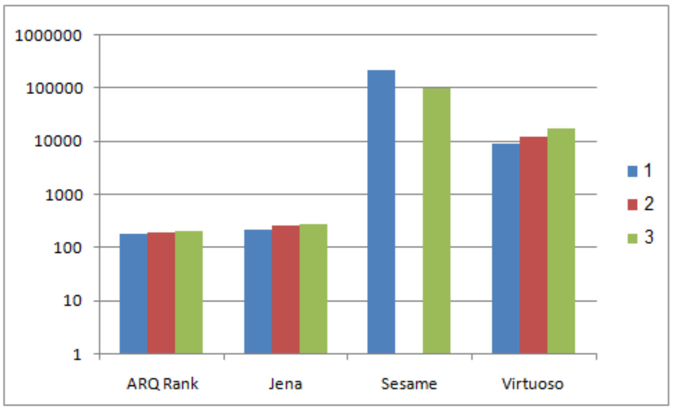


(n) Query 14

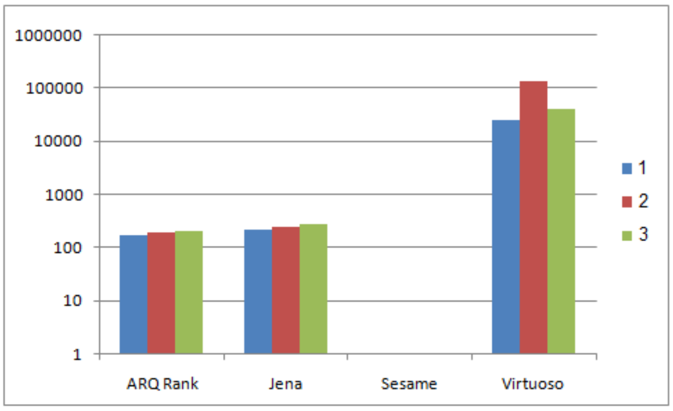


(o) Query 15

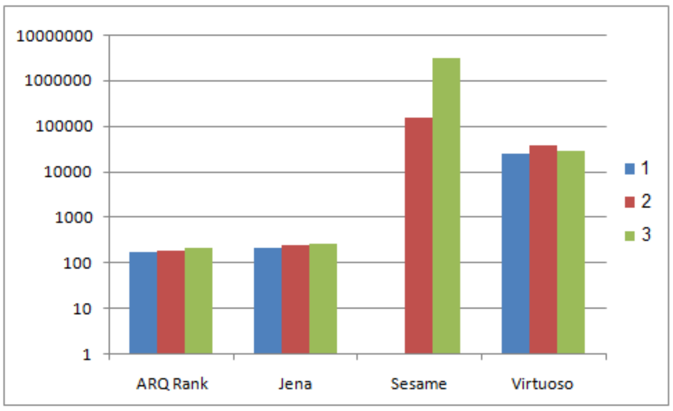
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(p) Query 16

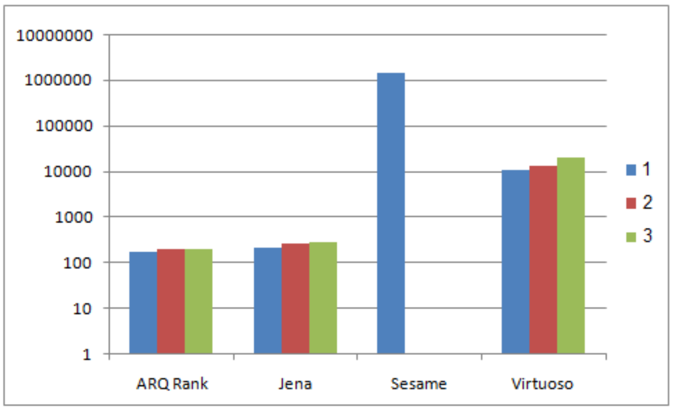


(q) Query 17

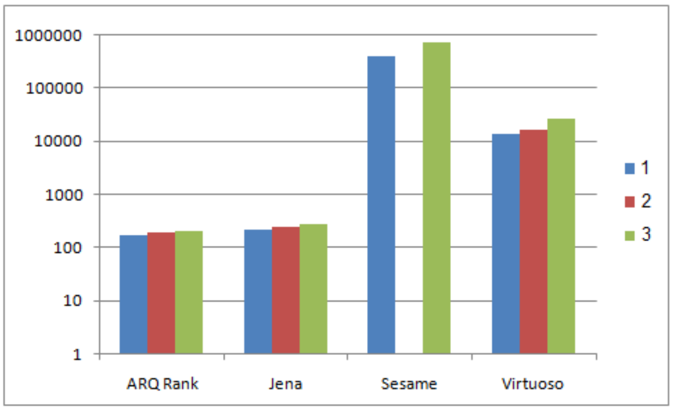


(r) Query 18

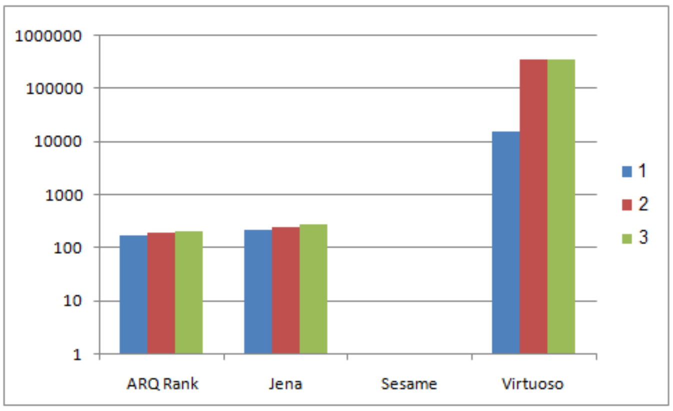
Figure 7.3: Execution Time of the Query according to different Predicate Number (cont.)



(s) Query 19



(t) Query 20



(u) Query 21

Figure 7.3: Execution Time of the Query according to different Predicate Number

7.3.3 Selectivity

The results of the testing with the test driver are shown in Figure 7.4. For each main query there exists a chart which indicate by Query ID. The names of the SPARQL engines come in the horizontal axis and the vertical axis shows the execution time in logarithmic scale. Different values of selectivity show with different color columns.

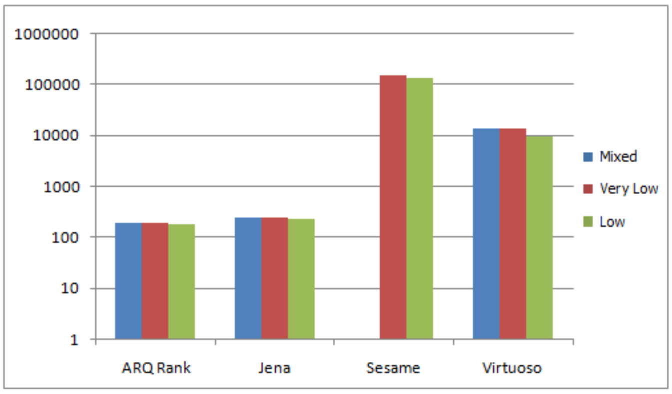
Reviewing the charts shows that in general most of the queries' behavior are against our hypothesis and the selectivity has not a linear relation with the execution time. This experiments show that our hypothesis is naive and it needs more investigation to find the relation between selectivity and execution time.

Evaluation for some of the cases is not possible due to having only one type of selectivity (queries 2, 4, 5, 9, 12, 15, 20, and 21). Considering the aggregation by query and SPARQL engine we will evaluate the execution time of the rest of the queries.

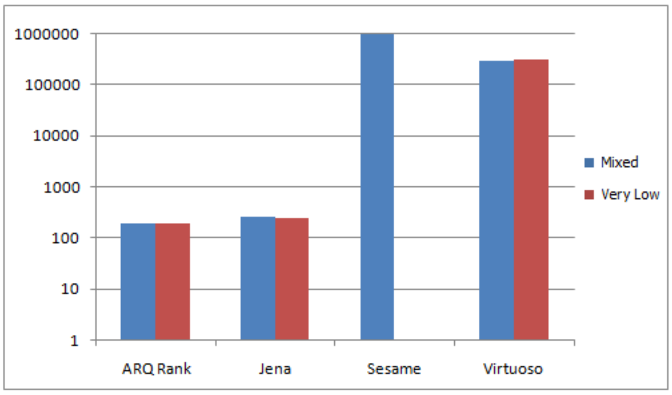
Aggregating by query, the results show that only a few cases are compatible with our hypothesis which are queries 1 (Figure 7.4(a)), 14 (Figure 7.4(n)), 17 (Figure 7.4(q)), 19 (Figure 7.4(s)).

Aggregating by SPARQL engine, the results indicate that in ARQ Rank and Jena Only 4, and 5 cases from 13 cases are compatible with the hypothesis. In Sesame it dose not exist enough data in most of the cases to evaluate the assumption. There exist two cases against and two cases compatible with our hypothesis. In Virtuoso 4 cases from 13 are compatible with our assumption.

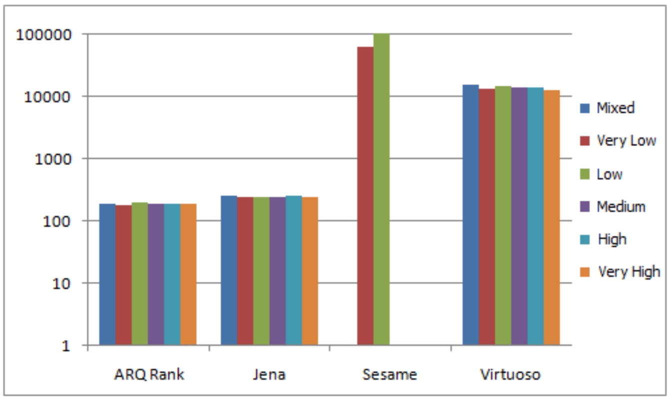
According to the results that are shown in the Figure 7.4, we can say that our assumption is not validated and it seems that the hypothesis, the more selectivity causes the less execution time, is a naive assumption and more investigation is needed to find a relationship between selectivity and execution time. However, in some cases we can find same trend and behavior for query in different SPARQL engines. For example the behavior of query 6 (Figure 7.4(f)) is the same in the ARQ Rank and Jena. The behavior of the queries 7 (Figure 7.4(g)), 8 (Figure 7.4(h)), and 10 (Figure 7.4(j)) are also the same as each other in ARQ Rank, Jena, and Virtuoso which need more analysis to find the reason.



(a) Query 1

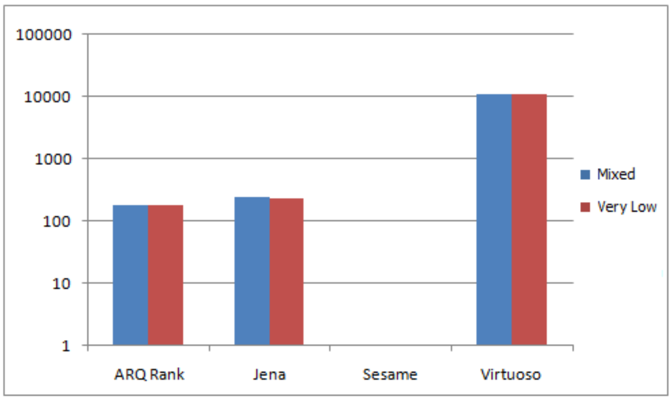


(b) Query 2

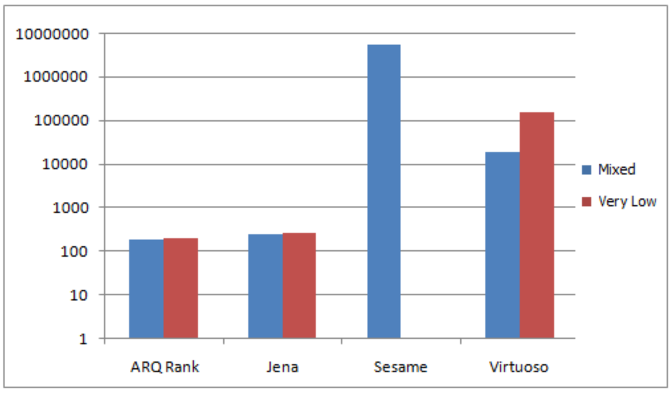


(c) Query 3

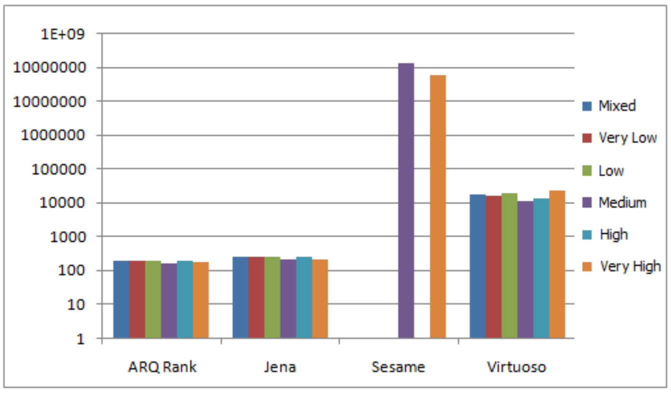
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(d) Query 4

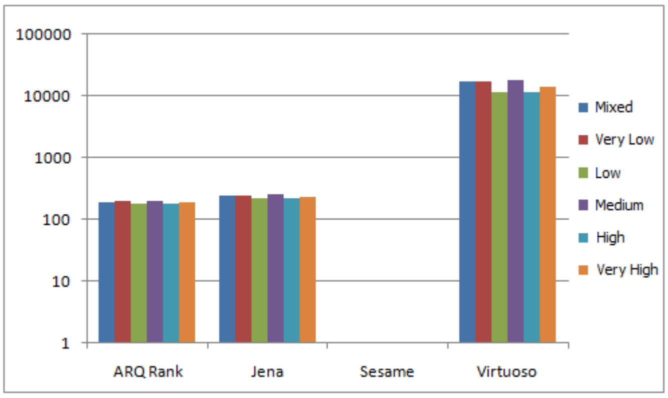


(e) Query 5

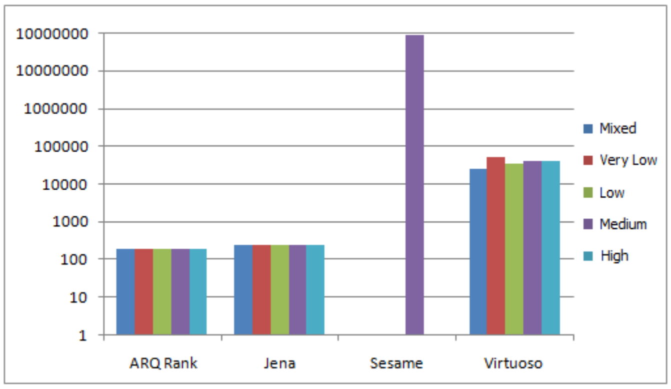


(f) Query 6

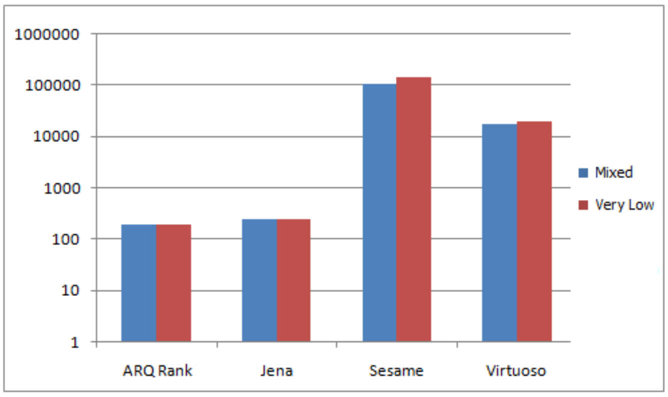
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(g) Query 7

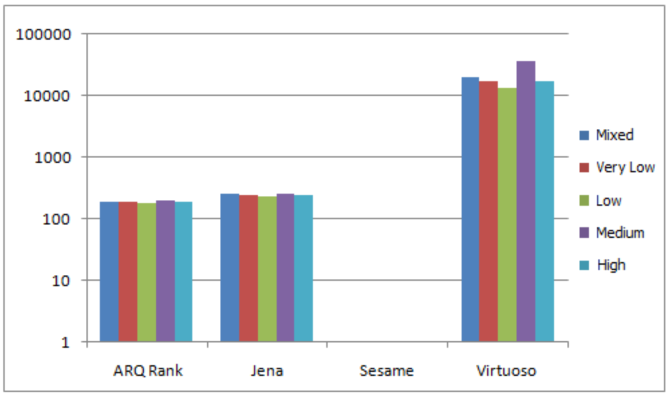


(h) Query 8

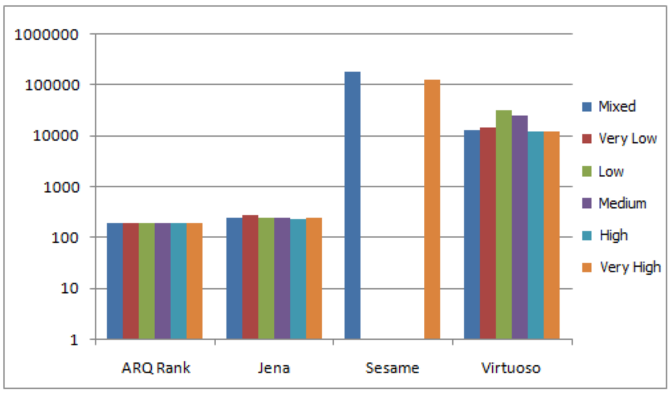


(i) Query 9

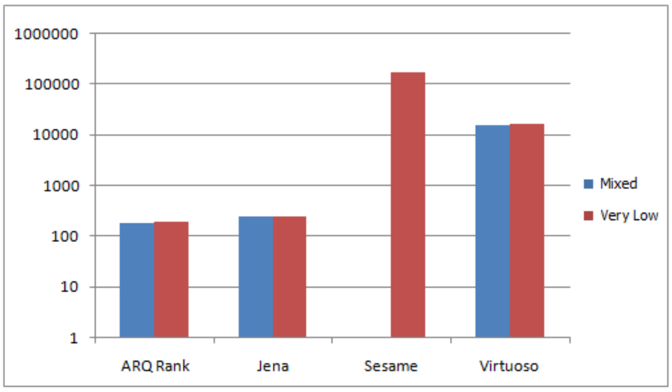
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(j) Query 10

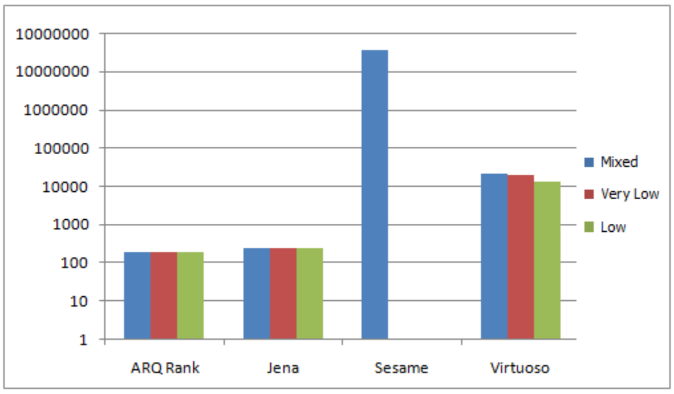


(k) Query 11

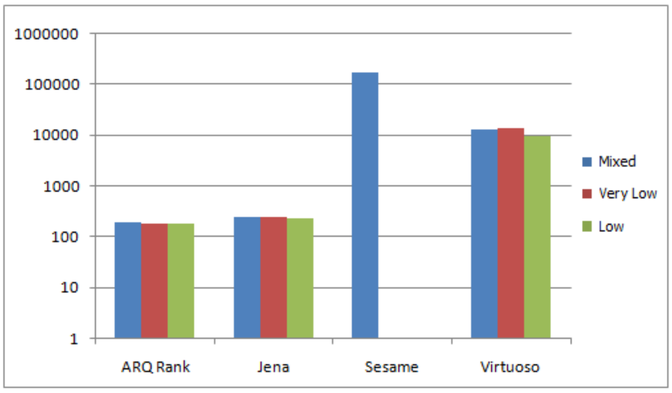


(l) Query 12

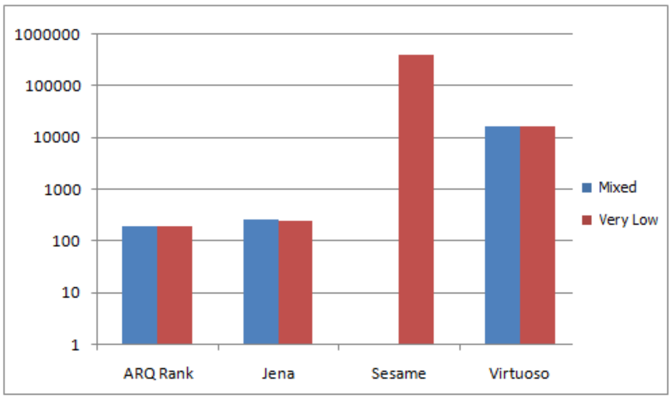
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(m) Query 13

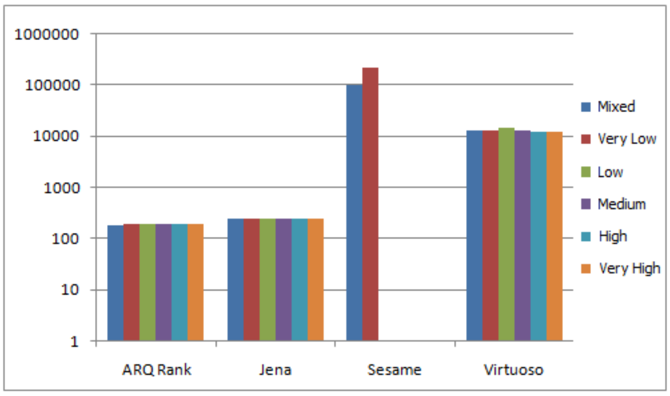


(n) Query 14

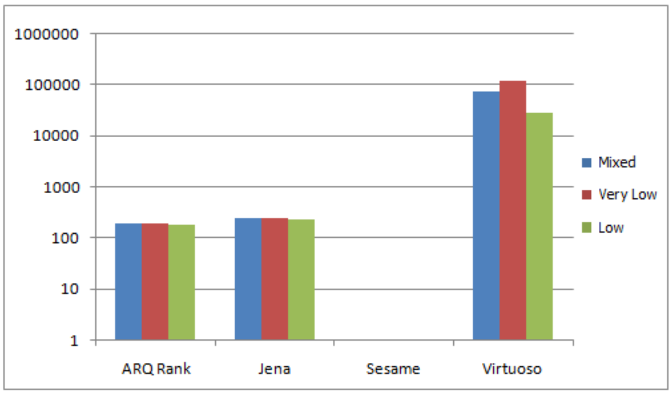


(o) Query 15

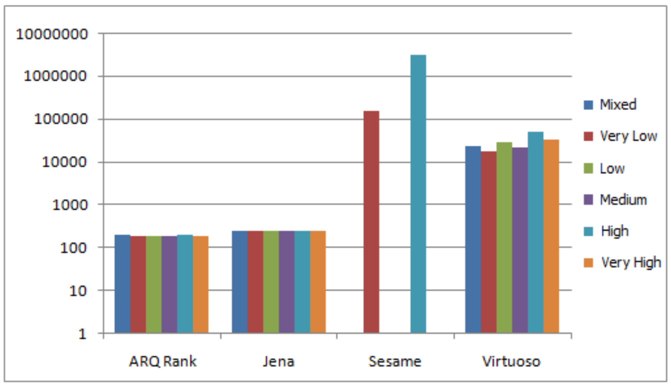
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(p) Query 16

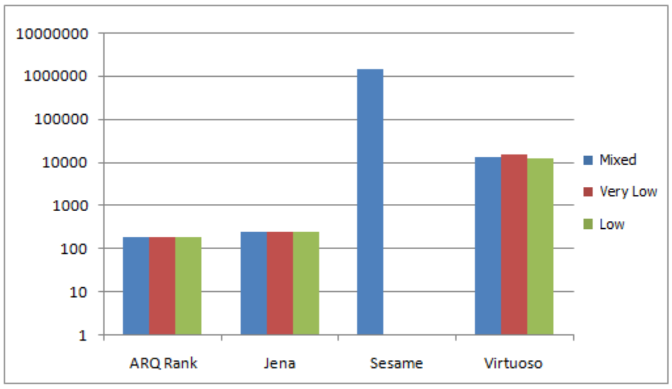


(q) Query 17

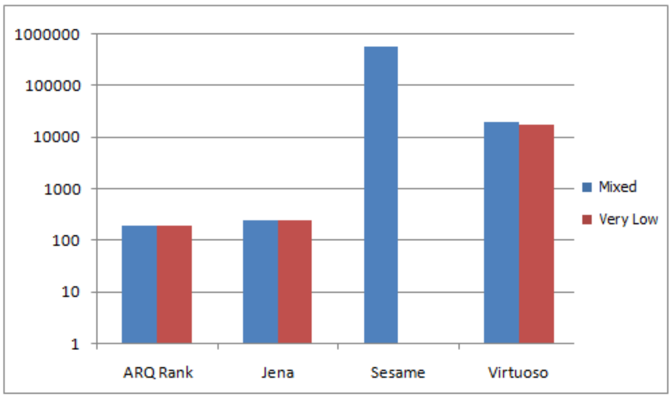


(r) Query 18

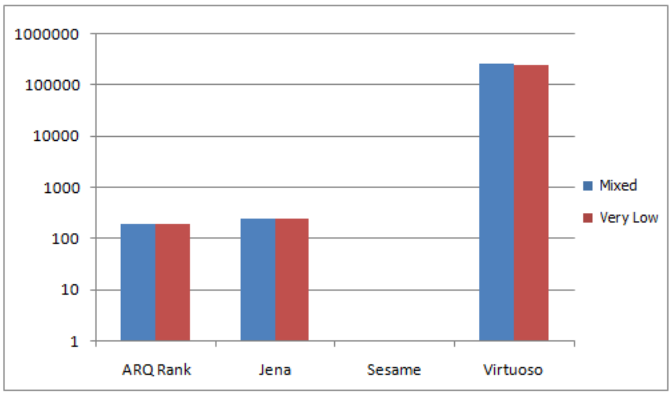
Figure 7.4: Execution Time of the Query according to different Selectivity (cont.)



(s) Query 19



(t) Query 20



(u) Query 21

Figure 7.4: Execution Time of the Query according to different Selectivity

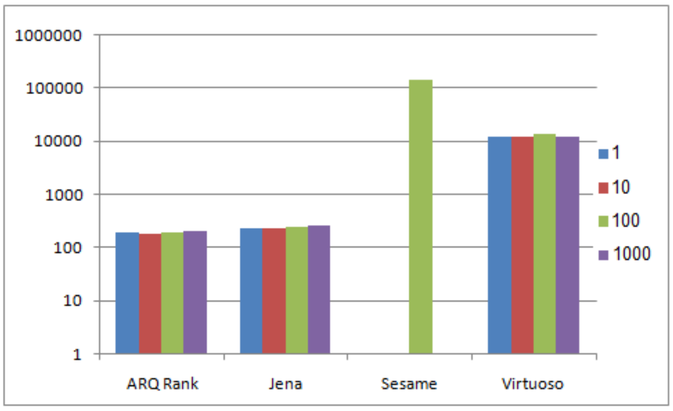
7.3.4 Limit

The results of the testing with the test driver are shown in Figure 7.5. For each main query there exists a chart which indicates by Query ID. The names of the SPARQL engines come in the horizontal axis and the vertical axis shows the execution time in logarithmic scale. Different values of Limit show with different color columns.

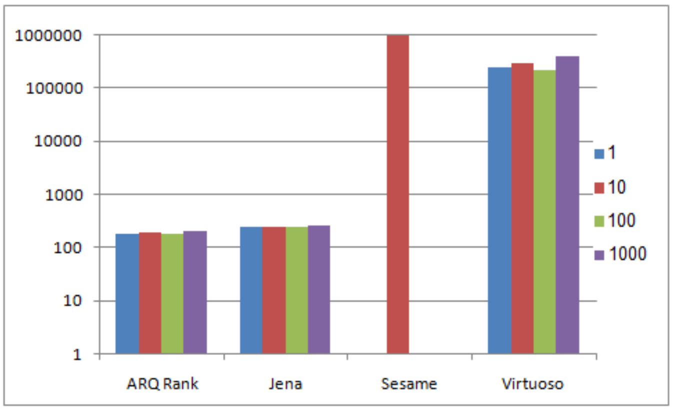
Reviewing the charts shows that in general most of the queries' behavior are compatible with our hypothesis. Aggregating by query, the results show that queris 2 (Figure 7.5(b)), 5 (Figure 7.5(e)), 8 (Figure 7.5(h)), 17 (Figure 7.5(q)), 18 (Figure 7.5(r)), and 21 (Figure 7.5(u)) are against the assumption.

Aggregating by SPARQL engine, the results indicate that in ARQ Rank and Jena our hypothesis is true in all cases. In Sesame the result of some cases (6 queries) are against our assumption but due to long execution time and lack ofenough data, we can not reach any result. In Virtuoso queries 2 (Figure 7.5(b)), 5 (Figure 7.5(e)), 8 (Figure 7.5(h)), 17 (Figure 7.5(q)), 18 (Figure 7.5(r)), and 21 (Figure 7.5(u)) can not validate our assumption.

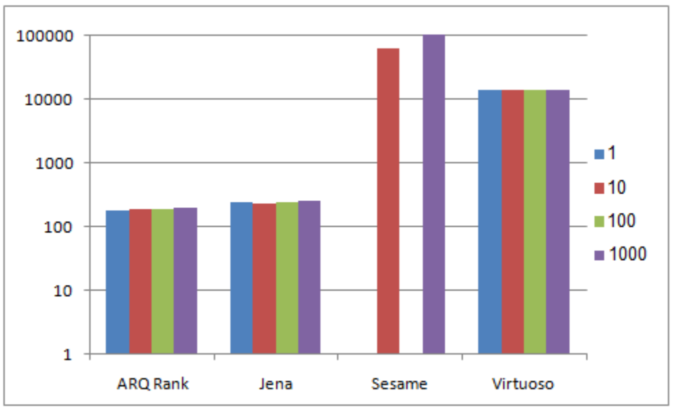
According to the result of Figure 7.5 our hypothesis is validate in most cases except the Sesame.



(a) Query 1

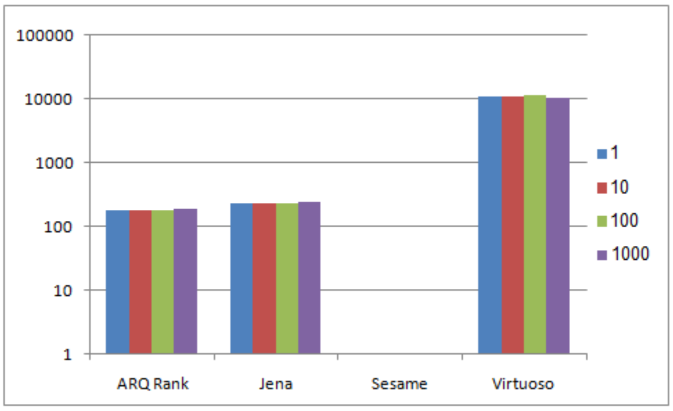


(b) Query 2

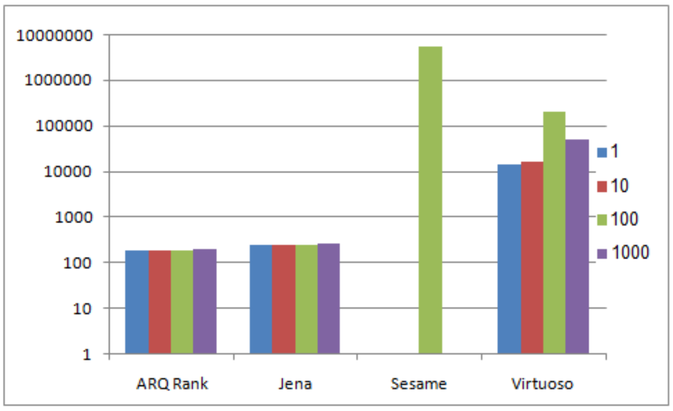


(c) Query 3

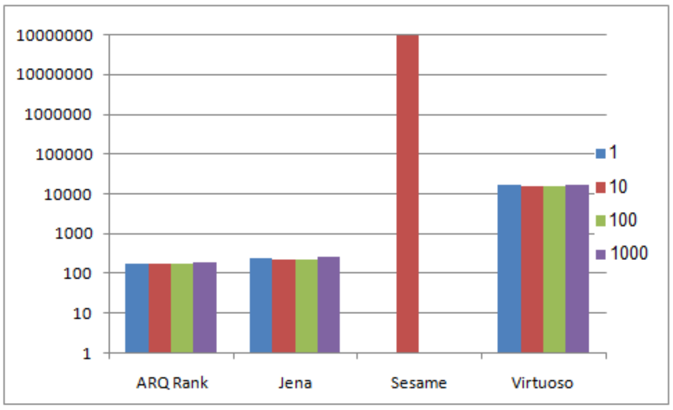
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(d) Query 4

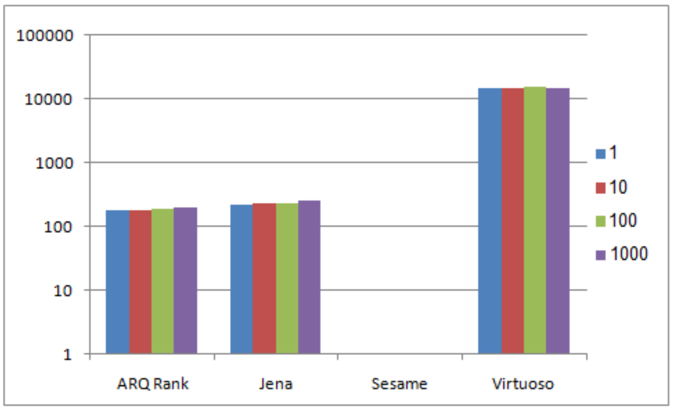


(e) Query 5

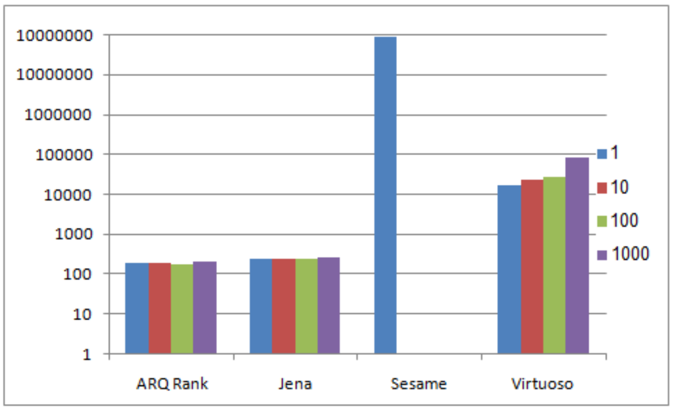


(f) Query 6

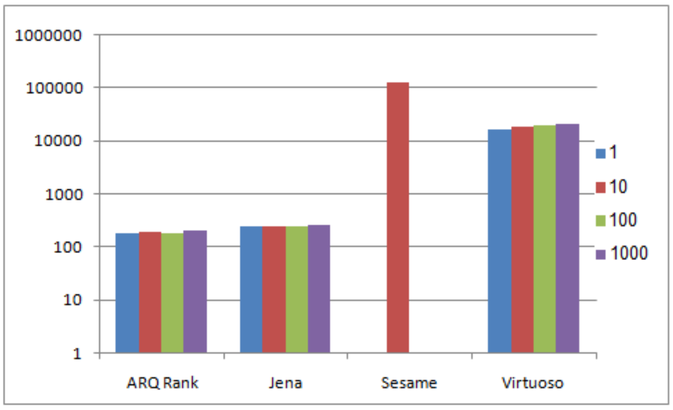
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(g) Query 7

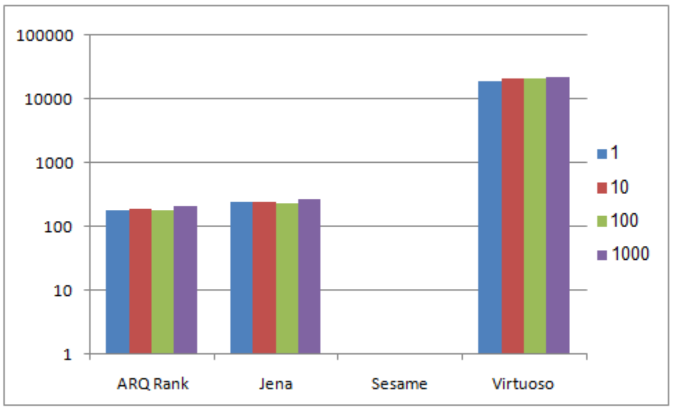


(h) Query 8

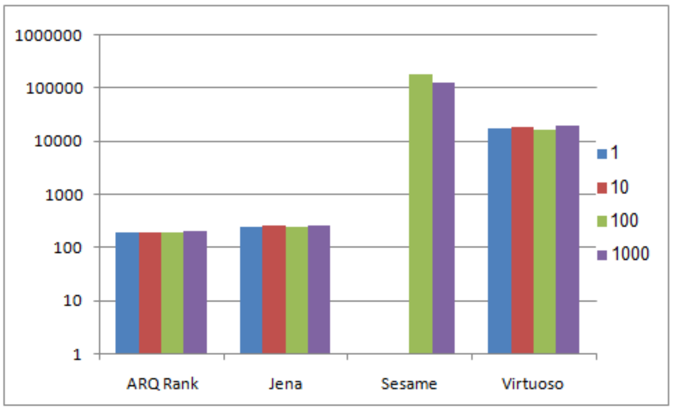


(i) Query 9

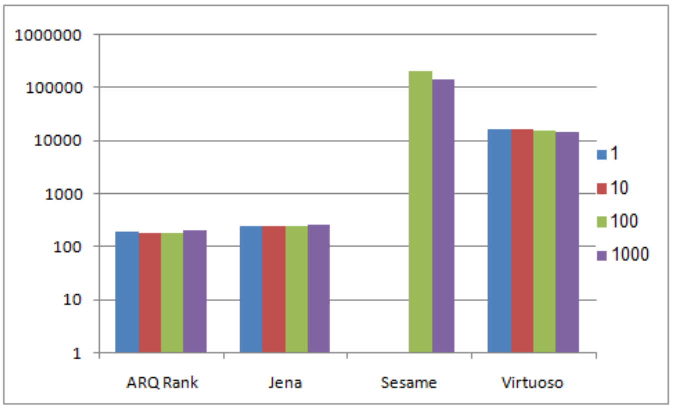
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(j) Query 10

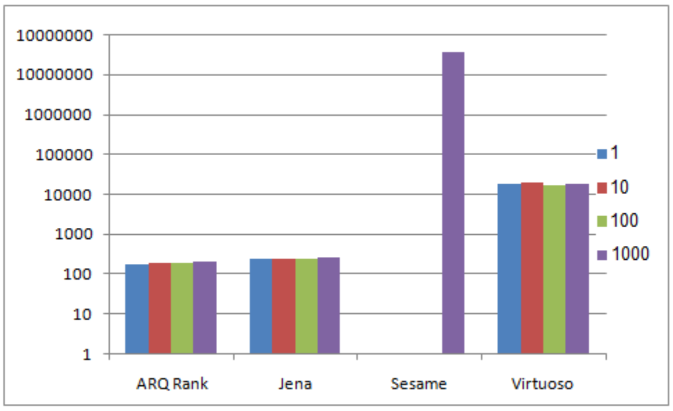


(k) Query 11

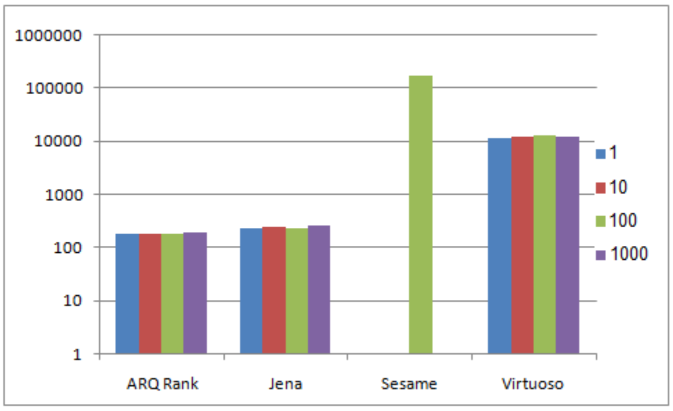


(l) Query 12

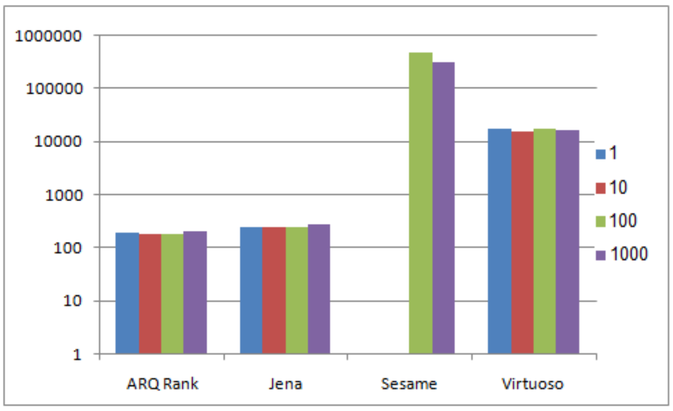
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(m) Query 13

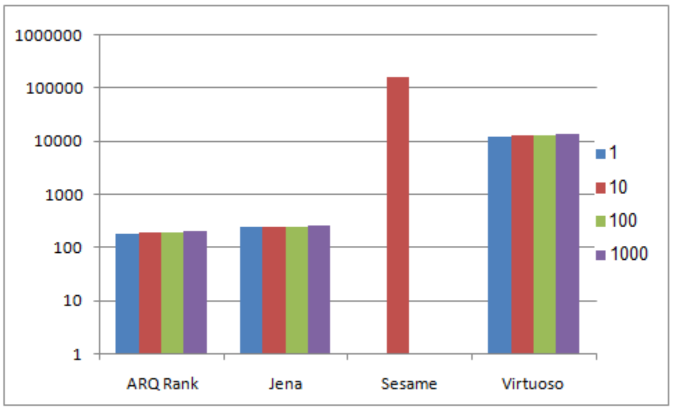


(n) Query 14

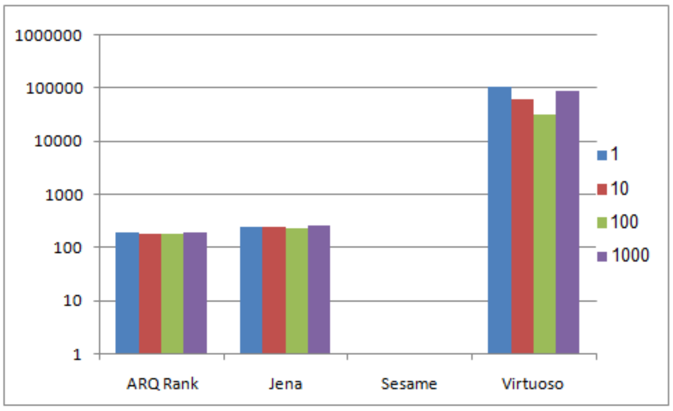


(o) Query 15

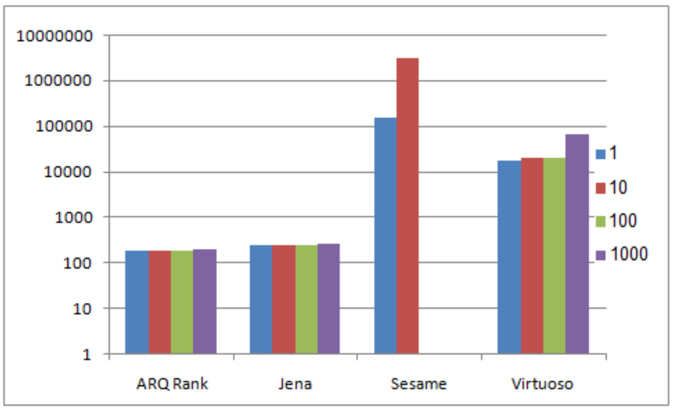
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(p) Query 16

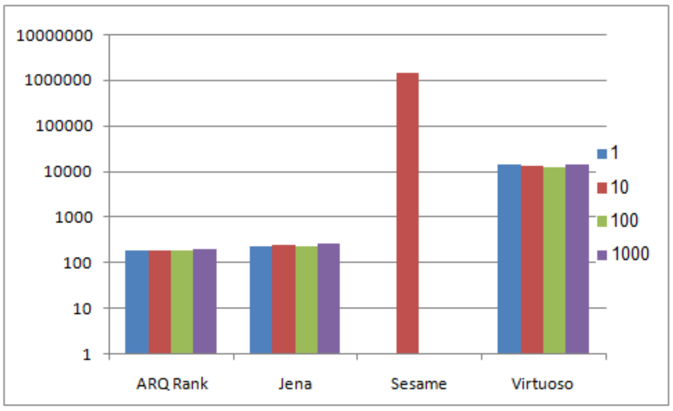


(q) Query 17

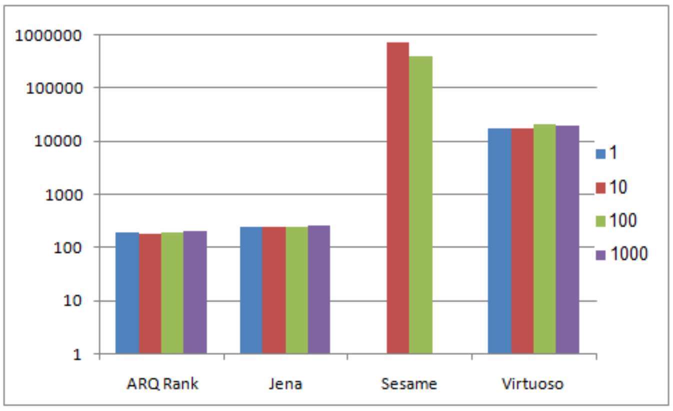


(r) Query 18

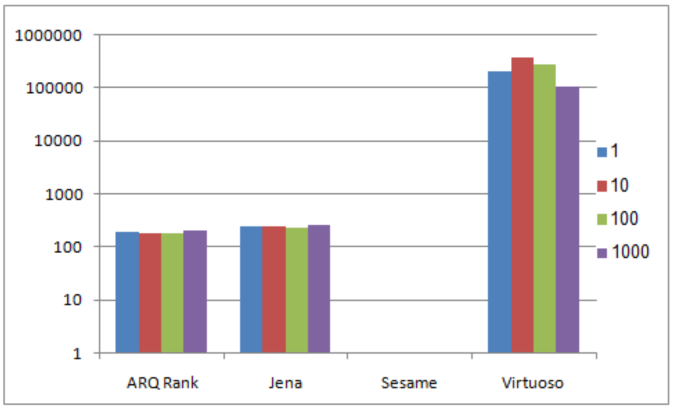
Figure 7.5: Execution Time of the Query according to different Limit (cont.)



(s) Query 19



(t) Query 20



(u) Query 21

Figure 7.5: Execution Time of the Query according to different Limit

The summery of the results which described in previous sections shows in table 7.2. In the second row of the table some abbreviations are used: A for ARQ Rank, J for Jena, S for Sesame, V for Virtuoso, and T for Total. The sign $\surd$ is used for the compatible cases and the sign $\times$ is used for the cases against our hypothesis.

	Object Number					Predicate Number					Selectivity					Limit				
Query ID	A	J	S	V	T	A	J	S	V	T	A	J	S	V	T	A	J	S	V	T
1						√	√		√	√	√	√	√	√	√		√		×	×
2						√	√		×	×							√	×	√	×
3						√	√	√	√	√	×	×	×	×	×		√		√	√
4						√	√		√	√							√		√	√
5	√	√	√	×	×	√	√		×	×							√		×	×
6						√	√		√	√	×	×	√	×	×		√		√	√
7	×	×		√	×	√	√		√	√	×	×		×	×		√		√	√
8	√	√		×	×	√	√		√	√	×	×		×	×		√		×	×
9	√	√		√	√	√	√	×	√	×							√		×	×
10	×	×		√	×	√	√		√	√	×	×		×	×		√		√	√
11						√	√	√	√	√	×	×		×	×		√	×	×	×
12						√	√		√	√							√	×	√	×
13						√	√		√	√	×	√		√	×		√		√	√
14						√	√	×	√	×	√	√		√	√		√		√	√
15						√	√	√	√	√							√	×	√	×
16						√	√	×	√	×	×	×		×	×		√		√	√
17						√	√		×	×	√	√		√	√		√	×	×	×
18						√	√	√	×	×	×	×	×	×	×		√	×	×	×
19						√	√		√	√	√	√		√	√		√		√	√
20						√	√	√	√	√							√		√	√
21	√	√		×	×	√	√		×	×							√	×	×	×
T	×	×	√	×	×	√	√	×	×	×	×	×	×	×	×		√	×	√	√

Table 7.2: Summery of experimental evaluation

Chapter 8

Conclusions and Future Work

In this thesis, I presented an automated Top-k query benchmark. Top-k queries are a very common category of queries, which aim at retrieving only the top k results ordered by a given ranking function.

The standard SPARQL query processing does not support an efficient evaluation of top-k queries, as they are mostly managed with a materialize-then-sort processing scheme that computes all the matching solutions even if only a limited number k are requested.

One of the important requirements that a meaningful benchmark should hold is resembling the reality. Using DBpedia SPARQL Benchmark which used DBpedia knowledge base, and query logs, as a foundation let us to keep close to the approach of resembling reality. The other important requirement is that the benchmark should stress the distinguish features of Top-k queries. From the syntactic perspective, we should use Top-k queries which contain ORDERBY and LIMIT clauses. From the performance perspective, the query mix should include queries with different characteristic, such as different value of k, ranging from low to high selectivity, controlling the correlation of ranking produced by multiple ranking predicates.

In this thesis, I fulfill the second requirement, by extending DBpedia SPARQL Benchmark (DBPSB) [Morsey et al., 2011] with the capabilities required to compare SPARQL engines on top-k queries. The Top-k DBpedia SPARQL Benchmark (Top-k DBPSB) proposed in this thesis uses the same dataset, performance metrics, and test driver of DBPSB. The innovative part of my work consists in an algorithm to create the Top-k queries from the Auxiliary queries of the DBPSB and its datasets.

In order to create a Top-k SPARQL query benchmark, it is necessary to define a process to create scoring function for each query. A scoring function is a combination of ranking criteria. In this work I use the simplest ranking criterion which is a ranking predicate that has an object with rankable data type. Different combinations of scoring predicates from different scoring subjects can be selected for the scoring function. It is possible that all the scoring criteria related to one subject or they be selected from different. It is also possible to consider different level of selectivity for the predicates of the scoring function. all these options, let us to create the variety of Top-k queries from a main query template. After selection of the predicates, related tripple patterns will be added to the query. The algorithm randomly generates the scoring function and adds ORDERBY and LIMIT clause to complete the Top-k queries.

In the experimental evaluation I use four SPARQL engines: ARQ Rank, Jena, Sesame, and Virtuoso. In order to evaluate these SPARQL engines I performed a series of experiments and compared the performances of the engines.

I have four research hypothesis: 1) the more rankable objects, the more execution time, 2) the more rankable predicates, the more execution time, 3) the more selectivity, the less execution time, and 4) the more the limit value, the same execution time (except ARQ Rank). I run the Top-K DBPSB against different four SPARQL Engines. The results of the extensive experimental evaluation confirm some of our hypothesis and do not confute some others, which require further research.

8.1 Future Work

The presented work is a beginning in the Top-k SPARQL query benchmarking, and it leaves several opportunities for further enhancement. In the following we present some of the possible extension points that we found during our research:

- Using complex type of scoring function in the top-k query generator.
- Using complex form of ranking criteria such as aggregated ranking criteria which use an aggregation function such as average, count, sum, maximum, and so on to compute the value of specific ranking criterion, and rankable criteria which produced by inference.
- More investigation on the hypothesis related to the selectivity. It seems that the more selectivity of the predicate, the less execution time is a

naive hypothesis and the results of the experiments show that it needs more analysis.

- More investigation on the queries that not follow our hypothesis

Bibliography

Lora Aroyo, Chris Welty, Harith Alani, Jamie Taylor, Abraham Bernstein, Lalana Kagal, Natasha Fridman Noy, and Eva Blomqvist, editors. *The Semantic Web - ISWC 2011 - 10th International Semantic Web Conference, Bonn, Germany, October 23-27, 2011, Proceedings, Part I*, volume 7031 of *Lecture Notes in Computer Science*, 2011. Springer. ISBN 978-3-642-25072-9.

Sören Auer, Christian Bizer, Claudia Müller, and Anna V. Zhdanova, editors. *The Social Semantic Web 2007, Proceedings of the 1st Conference on Social Semantic Web (CSSW), September 26-28, 2007, Leipzig, Germany*, volume 113 of *LNI*, 2007. GI. ISBN 978-3-88579-207-9.

T. Berners-Lee, J. Hendler, and O. Lassila. The semantic web. *Scientfic American Magazine*, 2001.

Barry Bishop, Atanas Kiryakov, Damyan Ognyanoff, Ivan Peikov, Zdravko Tashev, and Ruslan Velkov. Owlim: A family of scalable semantic repositories. *Semantic Web*, 2(1):33–42, 2011.

Christian Bizer and Andreas Schultz. The berlin sparql benchmark. *Int. J. Semantic Web Inf. Syst.*, 5(2):1–24, 2009.

Jeen Broekstra, Arjohn Kampman, and Frank van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In Horrocks and Hendler [2002], pages 54–68. ISBN 3-540-43760-6.

Jeen Broekstra, Arjohn Kampman, and Frank Van Harmelen. Sesame: A generic architecture for storing and querying rdf and rdf schema. In *The Semantic Web ISWC 2002*, pages 54–68. Springer, 2002b.

Juan Pablo Cedeno. *A Framework for Top-K Queries over Weighted RDF Graphs.* PhD thesis, Arizona State University, 2010.

Jingwei Cheng, ZM Ma, and Li Yan. f-sparql: a flexible extension of sparql. In *Database and Expert Systems Applications*, pages 487–494. Springer, 2010.

David J DeWitt. The wisconsin benchmark: Past, present, and future., 1993.

Orri Erling and Ivan Mikhailov. Rdf support in the virtuoso dbms. In Auer et al. [2007], pages 59–68. ISBN 978-3-88579-207-9.

Jim Gray, editor. *The Benchmark Handbook for Database and Transaction Systems (2nd Edition)*. Morgan Kaufmann, 1993. ISBN 1-55860-292-5.

Yuanbo Guo, Zhengxiang Pan, and Jeff Heflin. Lubm: A benchmark for owl knowledge base systems. *J. Web Sem.*, 3(2-3):158–182, 2005.

Ian Horrocks and James A. Hendler, editors. *The Semantic Web - ISWC 2002, First International Semantic Web Conference, Sardinia, Italy, June 9-12, 2002, Proceedings*, volume 2342 of *Lecture Notes in Computer Science*, 2002. Springer. ISBN 3-540-43760-6.

Yannis E. Ioannidis, Dik Lun Lee, and Raymond T. Ng, editors. *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, 2009. IEEE. ISBN 978-0-7695-3545-6.

Chengkai Li, Kevin Chen-Chuan Chang, Ihab F Ilyas, and Sumin Song. Ranksql: query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 131–142. ACM, 2005.

Sara Magliacane. Towards a rank aware execution of top k sparql queries. 2011.

Sara Magliacane, Alessandro Bozzon, and Emanuele Della Valle. Efficient execution of top-k sparql queries. In *The Semantic Web–ISWC 2012*, pages 344–360. Springer, 2012.

Mohamed Morsey, Jens Lehmann, Sören Auer, and Axel-Cyrille Ngonga Ngomo. Dbpedia sparql benchmark - performance assessment with real queries on real data. In Aroyo et al. [2011], pages 454–469. ISBN 978-3-642-25072-9.

Axel-Cyrille Ngonga Ngomo and Sören Auer. Limes - a time-efficient approach for large-scale link discovery on the web of data. In Walsh [2011], pages 2312–2317. ISBN 978-1-57735-516-8.

Patrick E O'Neil. The set query benchmark., 1993.

O Orlink. Implementing a sparql compliant rdf triple store using a sql-ordbms. *URL: http://virtuoso.openlinksw.com/dataspace/dav/wiki/Main/VOSRDFWP*, 2010.

Alisdair Owens, Andy Seaborne, Nick Gibbins, and mc schraefel. Clustered tdb: A clustered triple store for jena. Technical report, Electronics and Computer Science, University of Southampton, 2008.

Bijan Parsia Pascal Hitzler, Markus Krötzsch. Owl 2 web ontology language primer (second edition), December 2012. URL `http://www.w3.org/TR/owl2-primer/`.

E. Prudhommeaux and A. Seaborne. Sparql query language for rdf w3c recommendation, January 2008. URL `http://www.w3.org/TR/rdf-sparql-query/`.

Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. Sp2bench: A sparql performance benchmark. In Ioannidis et al. [2009], pages 222–233. ISBN 978-0-7695-3545-6.

Carolyn Turbyfill, Cyril U Orji, and Dina Bitton. As3ap: An ansi sql standard scaleable and portable benchmark for relational database systems., 1993.

Andreas Wagner, Thanh Tran Duc, Günter Ladwig, Andreas Harth, and Rudi Studer. Top-k linked data query processing. In *The Semantic Web: Research and Applications*, pages 56–71. Springer, 2012.

Toby Walsh, editor. *IJCAI 2011, Proceedings of the 22nd International Joint Conference on Artificial Intelligence, Barcelona, Catalonia, Spain, July 16-22, 2011*, 2011. IJCAI/AAAI. ISBN 978-1-57735-516-8.

Appendix A

Here is the list of DBpedia SPARQL Benchmark queries which we used in our work.

Query 1

```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE ?var rdf:type ?var1 .
LIMIT 1000
```

Query 2

```
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var ?var2 ?var1.
filter(?var2 = dbpedia2:redirect || ?var2 = dbpprop:redirect)
LIMIT 1000
```

Query 3

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var5 dbpedia-owl:thumbnail ?var4 .
?var5 rdf:type dbpedia-owl:Person .
?var5 rdfs:label ?var .
?var5 foaf:page ?var8 .
OPTIONAL  ?var5 foaf:homepage ?var10 . .
LIMIT 1000
```

Query 4

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE   ?var ?var5 ?var6 . ?var6 foaf:name ?var8 .
UNION  ?var9 ?var5 ?var ; foaf:name ?var4 .
LIMIT 1000
```

Query 5

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbpp: <http://dbpedia.org/property/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var0 ?var1 FROM <http://dbpedia.org>
WHERE   ?var3 dbpp:series ?var1 ;
foaf:name ?var4 ;
rdfs:comment ?var5 ;
rdf:type ?var0 .
UNION  ?var3 dbpp:series ?var8 .
?var8 dbpp:redirect ?var1 .
?var3 foaf:name ?var4 ; rdfs:comment ?var5 ; rdf:type ?var0 .
LIMIT 1000
```

Query 6

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE ?var3 rdf:type <http://dbpedia.org/class/yago/Company108058098> .
?var3 dbpedia2:numEmployees ?var . ?var3 foaf:homepage ?var7 .
FILTER (datatype(?var) = <http://www.w3.org/2001/XMLSchema#int>)
LIMIT 1000
```

Query 7

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX umbelBus: <http://umbel.org/umbel/sc/Business>
PREFIX umbelCountry: <http://umbel.org/umbel/sc/IndependentCountry>
SELECT distinct ?var FROM <http://dbpedia.org>
WHERE  ?var0 rdfs:comment ?var1. ?var0 foaf:page ?var
OPTIONAL?var0 skos:subject ?var6
OPTIONAL?var0 dbpedia2:industry ?var5
OPTIONAL?var0 dbpedia2:location ?var2
OPTIONAL?var0 dbpedia2:locationCountry ?var3
OPTIONAL?var0 dbpedia2:locationCity ?var9;
dbpedia2:manufacturer ?var0
OPTIONAL?var0 dbpedia2:products ?var11; dbpedia2:model ?var0
OPTIONAL?var0 <http://www.georss.org/georss/point> ?var10
OPTIONAL?var0 rdf:type ?var7
LIMIT 1000
```

Query 8

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var0 ?var1
FROM <http://dbpedia.org>
WHERE   ?var2 rdf:type ?var1.
?var2 dbpedia2:population ?var0.
UNION  ?var2 rdf:type ?var1.
?var2 dbpedia2:populationUrban ?var0.
LIMIT 1000
```

Query 9

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var2 a <http://dbpedia.org/ontology/Settlement>;
rdfs:label ?var.
?var6 a <http://dbpedia.org/ontology/Airport>.
?var6 <http://dbpedia.org/ontology/city> ?var2
UNION ?var6 <http://dbpedia.org/ontology/location> ?var2
?var6 <http://dbpedia.org/property/iata> ?var5.
UNION ?var6 <http://dbpedia.org/ontology/iataLocationIdentifier> ?var5.
OPTIONAL  ?var6 foaf:homepage ?var6_home.
OPTIONAL  ?var6 <http://dbpedia.org/property/nativename> ?var6_name.
LIMIT 1000
```

Query 10

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var0 ?var1
FROM <http://dbpedia.org>  ?var3 foaf:page ?var7.
?var3 rdf:type <http://dbpedia.org/ontology/SoccerPlayer> .
?var3 dbpedia2:position ?var6 .
?var3 <http://dbpedia.org/property/clubs> ?var8.
?var8 <http://dbpedia.org/ontology/capacity> ?var1 .
?var3 <http://dbpedia.org/ontology/birthPlace> ?var5 .
?var5 ?var4 ?var0.
OPTIONAL ?var3 <http://dbpedia.org/ontology/number> ?var35.
```

```
Filter (?var4 = <http://dbpedia.org/property/populationEstimate> ||
?var4 = <http://dbpedia.org/property/populationCensus> ||
?var4 = <http://dbpedia.org/property/statPop> ) .
Filter (?var6 = 'Goalkeeper'@en ||
?var6 = <http://dbpedia.org/resource/Goalkeeper_%28association_football%29> ||
?var6 = <http://dbpedia.org/resource/Goalkeeper_%28football%29>)
LIMIT 1000
```

Query 11

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
PREFIX umbelBus: <http://umbel.org/umbel/sc/Business>
PREFIX umbelCountry: <http://umbel.org/umbel/sc/IndependentCountry>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE  ?var dbpedia2:subsid ?var3
OPTIONAL?var2 ?var dbpedia2:parent
OPTIONAL?var dbpedia2:divisions ?var4
UNION ?var2 ?var dbpedia2:parent
OPTIONAL?var dbpedia2:subsid ?var3
OPTIONAL?var dbpedia2:divisions ?var4
UNION ?var dbpedia2:divisions ?var4
OPTIONAL?var dbpedia2:subsid ?var3
OPTIONAL?var2 ?var dbpedia2:parent
LIMIT 1000
```

Query 12

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var2 rdf:type dbpedia-owl:Person .
?var2 dbpedia-owl:nationality ?var4 .
?var4 rdfs:label ?var5 .
```

```
?var2 rdfs:label ?var.
FILTER (lang(?var5) = 'en')
LIMIT 1000
```

Query 13

```
PREFIX rdfs:<http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE  ?var rdfs:comment ?var0.
FILTER (lang(?var0) = 'en')
UNION ?var foaf:depiction ?var1
UNION ?var foaf:homepage ?var2
LIMIT 1000
```

Query 14

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE  ?var4 skos:subject ?var . ?var4 foaf:name ?var6 .
OPTIONAL  ?var4 rdfs:comment ?var8 .
FILTER (LANG(?var8) = 'en') .
OPTIONAL  ?var4 rdfs:comment ?var10 .
FILTER (LANG(?var10) = 'de') .
LIMIT 1000
```

Query 15

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var2 rdf:type ?var ; rdfs:label ?var3 .
FILTER regex(str(?var3), 'pes', 'i')
```

```
LIMIT 1000
```

Query 16

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE ?var ?var4 ?var5 .
OPTIONAL ?var5 rdfs:label ?var6 .
FILTER(langMatches(lang(?var6),'EN')||
(! langMatches(lang(?var6),'*'))) .
FILTER(langMatches(lang(?var5),'EN')||
(! langMatches(lang(?var5),'*'))) .
OPTIONAL ?var4 rdfs:label ?var7
LIMIT 1000
```

Query 17

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dct: <http://purl.org/dc/terms/>
PREFIX map: <file:/home/moustaki/work/motools/musicbrainz/d2r-server-0.4/mbz_mapping_raw.n3#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX event: <http://purl.org/NET/c4dm/event.owl#>
PREFIX rel: <http://purl.org/vocab/relationship/>
PREFIX lingvoj: <http://www.lingvoj.org/ontology#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dbpprop: <http://dbpedia.org/property/>
PREFIX dbowl: <http://dbpedia.org/ontology/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX tags: <http://www.holygoat.co.uk/owl/redwood/0.1/tags/>
PREFIX db: <http://dbtune.org/musicbrainz/resource/>
PREFIX geo: <http://www.geonames.org/ontology#>
PREFIX bio: <http://purl.org/vocab/bio/0.1/>
PREFIX mo: <http://purl.org/ontology/mo/>
PREFIX vocab: <http://dbtune.org/musicbrainz/resource/vocab/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX mbz: <http://purl.org/ontology/mbz#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
?var2 <http://www.w3.org/2004/02/skos/core#subject> ?var.
?var2 <http://www.w3.org/2000/01/rdf-schema#label> ?var3.
FILTER (lang(?var3)='fr')
LIMIT 1000
```

Query 18

```
PREFIX owl: <http://www.w3.org/2002/07/owl#>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX : <http://dbpedia.org/resource/>
PREFIX dbpedia2: <http://dbpedia.org/property/>
PREFIX dbpedia: <http://dbpedia.org/>
PREFIX skos: <http://www.w3.org/2004/02/skos/core#>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE   ?var ?var3 ?var4.
FILTER ( (STR(?var3) = 'http://www.w3.org/2000/01/rdf-schema#label'
&& lang(?var4) = 'en') ||
(STR(?var3) = 'http://dbpedia.org/ontology/abstract' && lang(?var4) = 'en') ||
(STR(?var3) = 'http://www.w3.org/2000/01/rdf-schema#comment'
&& lang(?var4) = 'en') ||
(STR(?var3) != 'http://dbpedia.org/ontology/abstract' &&
STR(?var3) != 'http://www.w3.org/2000/01/rdf-schema#comment' &&
STR(?var3) != 'http://www.w3.org/2000/01/rdf-schema#label') )
UNION  ?var5 ?var3 ?var FILTER ( STR(?var3) = 'http://dbpedia.org/ontology/owner' ||
STR(?var3) = 'http://dbpedia.org/property/redirect' )
LIMIT 1000
```

Query 19

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE   ?var1 <http://www.w3.org/2000/01/rdf-schema#label> ?var
UNION  ?var1 <http://www.w3.org/2000/01/rdf-schema#label> ?var .
FILTER(regex(str(?var1),'http://dbpedia.org/resource/') ||
regex(str(?var1),'http://dbpedia.org/ontology/') ||
regex(str(?var1),'http://www.w3.org/2002/07/owl') ||
regex(str(?var1),'http://www.w3.org/2001/XMLSchema') ||
regex(str(?var1),'http://www.w3.org/2000/01/rdf-schema') ||
regex(str(?var1),'http://www.w3.org/1999/02/22-rdf-syntax-ns'))
LIMIT 1000
```

Query 20

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX geo: <http://www.w3.org/2003/01/geo/wgs84_pos#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var6 a <http://dbpedia.org/ontology/PopulatedPlace>;
<http://dbpedia.org/ontology/abstract> ?var1;
rdfs:label ?var2; geo:lat ?var3; geo:long ?var4.
?var6 rdfs:label ?var.
UNION  ?var5 <http://dbpedia.org/property/redirect> ?var6; rdfs:label ?var.
OPTIONAL  ?var6 foaf:depiction ?var8
OPTIONAL  ?var6 foaf:homepage ?var10
OPTIONAL  ?var6 <http://dbpedia.org/ontology/populationTotal> ?var12
OPTIONAL  ?var6 <http://dbpedia.org/ontology/thumbnail> ?var14
FILTER (langMatches( lang(?var1), 'de') && langMatches( lang(?var2), 'de') )
LIMIT 1000
```

Query 21

```
PREFIX dbpprop: <http://dbpedia.org/property/>
SELECT DISTINCT ?var FROM <http://dbpedia.org>
WHERE  ?var dbpprop:redirect ?var0 .
LIMIT 1000
```

Query 22

```
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var3 <http://xmlns.com/foaf/0.1/homepage> ?var2 .
?var3 <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> ?var .
LIMIT 1000
```

Query 23

```
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX xsd: <http://www.w3.org/2001/XMLSchema#>
PREFIX space: <http://purl.org/net/schemas/space/>
PREFIX dbpedia-owl: <http://dbpedia.org/ontology/>
PREFIX dbpedia-prop: <http://dbpedia.org/property/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var2 rdf:type dbpedia-owl:Person .
?var2 rdfs:label ?var . ?var2 foaf:page ?var4 .
LIMIT 1000
```

Query 24

```
SELECT DISTINCT ?var0 ?var1
FROM <http://dbpedia.org>
WHERE  ?var2 a <http://dbpedia.org/ontology/Organisation> .
?var2 <http://dbpedia.org/ontology/foundationPlace> ?var0 .
?var4 <http://dbpedia.org/ontology/developer> ?var2 .
?var4 a ?var1 .
LIMIT 1000
```

Query 25

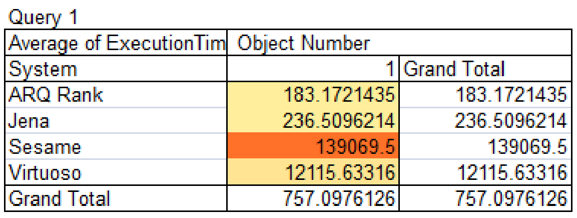
```
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX dbpprop:<http://dbpedia.org/property/>
```

```
SELECT DISTINCT ?var
FROM <http://dbpedia.org>
WHERE  ?var6 rdf:type ?var.
?var6 dbpprop:name ?var0.
?var6 dbpprop:pages ?var1.
?var6 dbpprop:isbn ?var2.
?var6 dbpprop:author ?var3.
LIMIT 1000
```

Appendix B

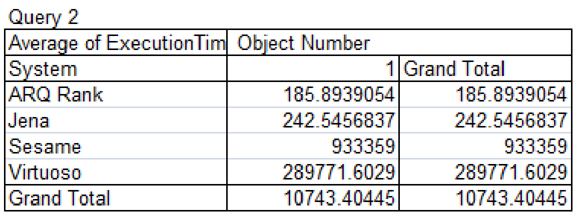
Here is the numeric tables of our experiments.

Execution Time of the Query according to different Object Number:



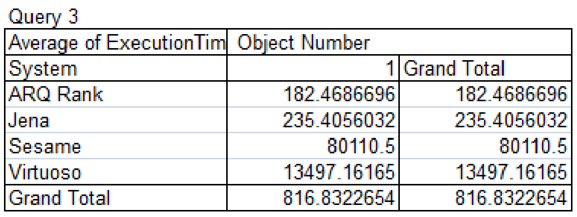
Query 1

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	183.1721435	183.1721435
Jena	236.5096214	236.5096214
Sesame	139069.5	139069.5
Virtuoso	12115.63316	12115.63316
Grand Total	757.0976126	757.0976126



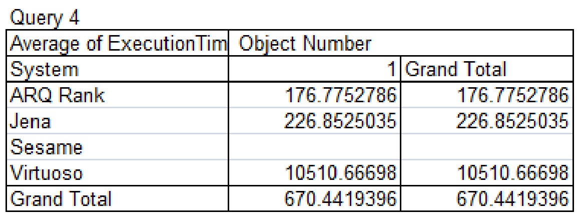
Query 2

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	185.8939054	185.8939054
Jena	242.5456837	242.5456837
Sesame	933359	933359
Virtuoso	289771.6029	289771.6029
Grand Total	10743.40445	10743.40445



Query 3

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	182.4686696	182.4686696
Jena	235.4056032	235.4056032
Sesame	80110.5	80110.5
Virtuoso	13497.16165	13497.16165
Grand Total	816.8322654	816.8322654



Query 4

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	176.7752786	176.7752786
Jena	226.8525035	226.8525035
Sesame		
Virtuoso	10510.66698	10510.66698
Grand Total	670.4419396	670.4419396

Query 5

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	177.9082332	189.538045	199.9543236	183.9376897
Jena	227.7878313	250.77173	265.146072	238.9209373
Sesame	3117325	7545832		5331578.5
Virtuoso	135391.911	17654.80984	20135.6609	79709.34322
Grand Total	5192.245084	1536.395604	1257.041957	3644.392662

Query 6

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	175.6788586	175.6788586
Jena	228.3555329	228.3555329
Sesame	90231298	90231298
Virtuoso	15740.11209	15740.11209
Grand Total	4725.4183	4725.4183

Query 7

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	180.3778515	182.9069767	173.8108108	180.3723721
Jena	226.6654342	263.2857143	245.5925926	226.77718
Sesame				
Virtuoso	14322.01719	18656	22292	14346.17369
Grand Total	846.3707453	423.8924731	2097.342857	847.3916933

Query 8

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	179.9437249	186.1606922	234.5996241	182.1572437
Jena	230.2641791	250.1784884	268.9818594	234.4122922
Sesame	84534712			84534712
Virtuoso	32297.9639	55975.74074	52618.21053	37638.7413
Grand Total	3466.543521	2529.104111	2218.51632	3282.80942

Query 9

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	181.0719367	187.1108446	208.7264231	185.1682655
Jena	232.3611985	250.0191983	260.862087	239.0525908
Sesame	121701			121701
Virtuoso	16543.09821	18719.468	22827.36792	18048.35323
Grand Total	681.9262537	1050.554392	1302.975184	828.7285506

Query 10

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	184.0731307	190.0774119	184.4458874	184.5009185
Jena	235.481847	257.7196198	250.4519231	237.2204934
Sesame				
Virtuoso	19770.06253	21483.08696	28075.875	20008.38397
Grand Total	1086.238627	1375.895218	1659.87473	1112.608455

Query 11

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	185.5710112	185.5710112
Jena	241.2357485	241.2357485
Sesame	148978.5	148978.5
Virtuoso	17541.57758	17541.57758
Grand Total	1014.913437	1014.913437

Query 12

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	183.7755235	183.7755235
Jena	240.7505695	240.7505695
Sesame	169975	169975
Virtuoso	15296.71051	15296.71051
Grand Total	905.8009685	905.8009685

Query 13

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	179.0293097	179.0293097
Jena	234.1501139	234.1501139
Sesame	36469554	36469554
Virtuoso	17581.54423	17581.54423
Grand Total	1768.832262	1768.832262

Query 14

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	179.3566559	179.3566559
Jena	234.6448548	234.6448548
Sesame	165151	165151
Virtuoso	11733.37634	11733.37634
Grand Total	735.1676079	735.1676079

Query 15

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	184.8235294	184.8235294
Jena	240.8563327	240.8563327
Sesame	382692	382692
Virtuoso	15980.88235	15980.88235
Grand Total	946.6226395	946.6226395

Query 16

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	185.4400082	185.4400082
Jena	238.357909	238.357909
Sesame	153177.5	153177.5
Virtuoso	12432.4113	12432.4113
Grand Total	775.1043233	775.1043233

Query 17

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	183.112445	183.112445
Jena	234.8482378	234.8482378
Sesame		
Virtuoso	69364.74545	69364.74545
Grand Total	3283.07504	3283.07504

Query 18

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	183.2870556	183.2870556
Jena	237.654452	237.654452
Sesame	1560428	1560428
Virtuoso	29663.44094	29663.44094
Grand Total	1543.403588	1543.403588

Query 19

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	180.3671062	180.3671062
Jena	233.8899714	233.8899714
Sesame	1392237	1392237
Virtuoso	13042.03487	13042.03487
Grand Total	818.3056472	818.3056472

Query 20

Average of ExecutionTim	Object Number	
System	1	Grand Total
ARQ Rank	185.1805119	185.1805119
Jena	239.4370123	239.4370123
Sesame	539009.5	539009.5
Virtuoso	18104.10607	18104.10607
Grand Total	1049.736218	1049.736218

Query 21

Average of ExecutionTim	Object Number			
System	1	2	3	Grand Total
ARQ Rank	181.3538245	190.2180451	199.4266455	183.4872842
Jena	232.3944292	251.5360312	256.8005725	236.2478309
Sesame				
Virtuoso	348833.0885	16994.29412	18636.57851	243821.8548
Grand Total	8074.002958	929.7630184	1142.234568	6722.481388

Execution Time of the Query according to different Predicate Number:

Query 1

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.4187538	186.1120224	198.0988731	183.1721435
Jena	210.5126039	243.8720668	263.4367715	236.5096214
Sesame		139069.5		139069.5
Virtuoso	8277.576316	11959.85034	18023.41586	12115.63316
Grand Total	567.0116959	731.7800078	1098.418796	757.0976126

Query 2

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	170.3036696	187.496121	197.9925036	185.8939054
Jena	210.1710686	246.5793463	266.8957198	242.5456837
Sesame			933359	933359
Virtuoso	26768.50812	476988.2529	312453.6901	289771.6029
Grand Total	960.4972372	17171.2882	13724.67646	10743.40445

Query 3

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.2846787	181.1436324	197.9007405	182.4686696
Jena	208.3041207	240.5605394	258.4804512	235.4056032
Sesame	61085	99136		80110.5
Virtuoso	9500.442982	12322.02817	18758.73919	13497.16165
Grand Total	596.3876304	780.3194902	1090.999462	816.8322654

Query 4

Average of ExecutionTim	Predicate Number		
System	1	2	Grand Total
ARQ Rank	169.2061391	186.097932	176.7752786
Jena	214.1916104	242.0417955	226.8525035
Sesame			
Virtuoso	8694.672788	12774.51821	10510.66698
Grand Total	581.9780131	778.2154618	670.4419396

Query 5

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.5912297	186.2098369	196.8427953	183.9376897
Jena	211.2203033	238.7612056	267.5134083	238.9209373
Sesame		5331578.5		5331578.5
Virtuoso	17075.58052	179303.3666	34101.53888	79709.34322
Grand Total	715.1367553	8756.203675	1805.470039	3644.392662

Query 6

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	166.8264327	182.8224224	194.0048611	175.6788586
Jena	209.2904449	245.990706	265.345116	228.3555329
Sesame	90231298			90231298
Virtuoso	14366.84561	15262.17333	21091.469	15740.11209
Grand Total	7508.237478	880.856259	1148.79183	4725.4183

Query 7

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	170.7790954	190.5820424	197.62182	180.3723721
Jena	210.9872671	240.6183503	260.1143026	226.77718
Sesame				
Virtuoso	11933.34646	15632.7677	21314.74556	14346.17369
Grand Total	737.2677495	905.9169654	1139.391003	847.3916933

Query 8

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.8995971	184.7581317	201.6920964	182.1572437
Jena	211.2400903	241.3261021	265.3931666	234.4122922
Sesame	84534712			84534712
Virtuoso	26847.36735	41704.6217	53292.08015	37638.7413
Grand Total	5738.630419	1670.581038	1770.049264	3282.80942

Query 9

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	168.1853389	184.8818495	200.92358	185.1682655
Jena	211.9714662	240.4348024	263.4363309	239.0525908
Sesame	139242		104160	121701
Virtuoso	11142.40063	15770.38889	22601.33289	18048.35323
Grand Total	427.5123469	772.3266439	1242.50959	828.7285506

Query 10

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	170.1469008	187.5837959	200.4465591	184.5009185
Jena	211.3980833	242.9071585	265.737939	237.2204934
Sesame				
Virtuoso	13309.77636	20226.84347	28844.04545	20008.38397
Grand Total	795.0930307	1113.571468	1535.818208	1112.608455

Query 11

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	173.0259908	187.1089067	199.9383274	185.5710112
Jena	212.5623589	247.5718793	269.351828	241.2357485
Sesame	121176	176781		148978.5
Virtuoso	12062.66984	15913.15054	26748.86348	17541.57758
Grand Total	733.5364405	949.0376048	1479.897189	1014.913437

Query 12

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	166.649819	183.1595911	198.0108889	183.7755235
Jena	211.6096037	243.1035426	261.9006048	240.7505695
Sesame			169975	169975
Virtuoso	12357.11442	13237.3498	19896.59176	15296.71051
Grand Total	745.810805	835.4507736	1098.865265	905.8009685

Query 13

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.327058	181.6563636	194.4535206	179.0293097
Jena	210.0698954	247.4393204	261.8232056	234.1501139
Sesame	36469554			36469554
Virtuoso	13245.8349	16398.22345	28078.77987	17581.54423
Grand Total	2533.125677	946.2696039	1499.038063	1768.832262

Query 14

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.4150943	181.5710459	196.6798299	179.3566559
Jena	212.0230544	244.0513024	265.7780172	234.6448548
Sesame		182006	148296	165151
Virtuoso	8623.150985	11634.6109	18311.46622	11733.37634
Grand Total	566.6308439	741.3452518	1083.762004	735.1676079

Query 15

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	166.3522806	183.8860076	202.498194	184.8235294
Jena	209.9736022	241.4886115	267.9884632	240.8563327
Sesame	309672	455712		382692
Virtuoso	12668.63566	14672.68187	20529.03272	15980.88235
Grand Total	785.3180971	893.2464402	1153.400762	946.6226395

Query 16

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	170.8396085	189.7880435	197.2140673	185.4400082
Jena	209.097243	245.8018967	263.5202365	238.357909
Sesame	209285		97070	153177.5
Virtuoso	8880.91634	11806.01275	17024.17151	12432.4113
Grand Total	596.3191983	751.5569098	997.4811022	775.1043233

Query 17

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	168.0435624	189.0023279	197.5941715	183.112445
Jena	210.0866039	242.2609549	260.364441	234.8482378
Sesame				#DIV/0!
Virtuoso	23447.21745	128117.9699	39962.3546	69364.74545
Grand Total	1137.807507	6288.112705	2031.495697	3283.07504

Query 18

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	165.9440137	183.6031938	203.3226109	183.2870556
Jena	211.685931	242.1781036	262.2726129	237.654452
Sesame		151310	2969546	1560428
Virtuoso	23984.83826	37713.39736	27390.22692	29663.44094
Grand Total	1197.972148	1821.233957	1636.198506	1543.403588

Query 19

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	167.7981998	185.7211397	197.4167569	180.3671062
Jena	207.887121	245.112871	268.2903553	233.8899714
Sesame	1392237			1392237
Virtuoso	10077.78132	13145.85662	19290.89952	13042.03487
Grand Total	707.9591283	791.7142003	1117.581375	818.3056472

Query 20

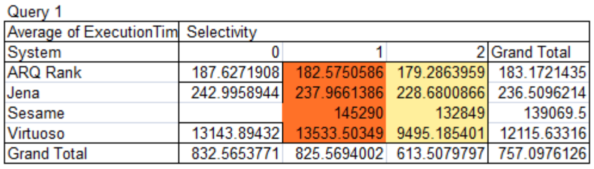
Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	169.0072197	184.4951097	199.370497	185.1805119
Jena	208.9715882	242.4431834	262.6262129	239.4370123
Sesame	380200		697819	539009.5
Virtuoso	13161.89552	15972.83984	24966.92649	18104.10607
Grand Total	822.9010997	958.1464107	1330.645827	1049.736218

Query 21

Average of ExecutionTim	Predicate Number			
System	1	2	3	Grand Total
ARQ Rank	167.1838133	182.5703589	201.6108859	183.4872842
Jena	211.2976738	238.8919758	260.893877	236.2478309
Sesame				
Virtuoso	14791.70145	333587.6694	331054.2065	243821.8548
Grand Total	495.7041771	8725.223817	11423.05732	6722.481388

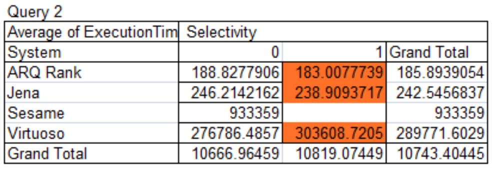
Execution Time of the Query according to different Selectivity:

Query 1



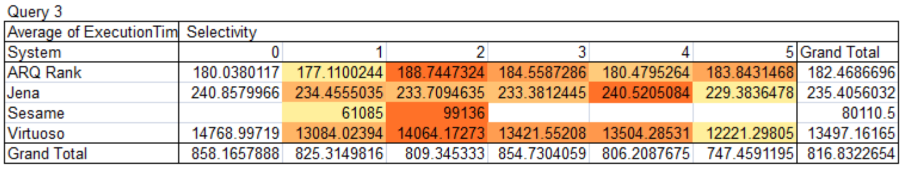
Average of ExecutionTim	Selectivity			
System	0	1	2	Grand Total
ARQ Rank	187.6271908	182.5750586	179.2863959	183.1721435
Jena	242.9958944	237.9661386	228.6800866	236.5096214
Sesame		145290	132849	139069.5
Virtuoso	13143.89432	13533.50349	9495.185401	12115.63316
Grand Total	832.5653771	825.5694002	613.5079797	757.0976126

Query 2



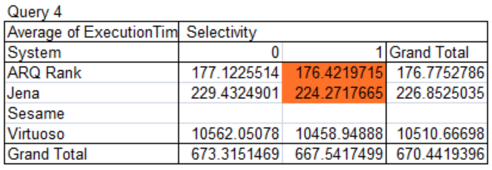
Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	188.8277906	183.0077739	185.8939054
Jena	246.2142162	238.9093717	242.5456837
Sesame	933359		933359
Virtuoso	276786.4857	303608.7205	289771.6029
Grand Total	10666.96459	10819.07449	10743.40445

Query 3



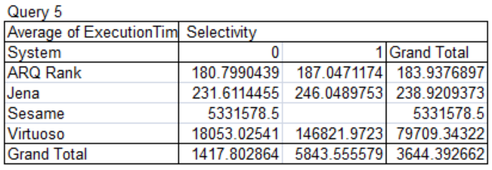
Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	180.0380117	177.1100244	188.7447324	184.5587286	180.4795264	183.8431468	182.4686696
Jena	240.8579966	234.4555035	233.7094635	233.3812445	240.5205084	229.3836478	235.4056032
Sesame		61085	99136				80110.5
Virtuoso	14768.99719	13084.02394	14064.17273	13421.55208	13504.28531	12221.29805	13497.16165
Grand Total	858.1657888	825.3149816	809.345333	854.7304059	806.2087675	747.4591195	816.8322654

Query 4



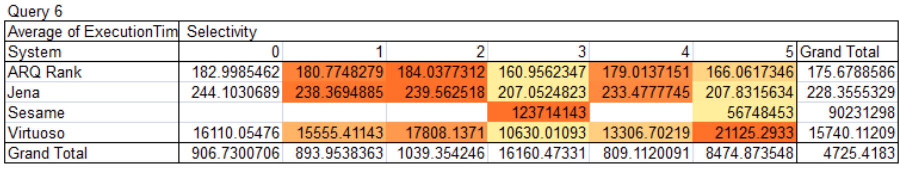
Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	177.1225514	176.4219715	176.7752786
Jena	229.4324901	224.2717665	226.8525035
Sesame			
Virtuoso	10562.05078	10458.94888	10510.66698
Grand Total	673.3151469	667.5417499	670.4419396

Query 5



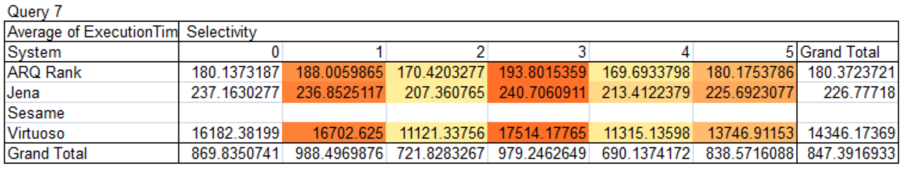
Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	180.7990439	187.0471174	183.9376897
Jena	231.6114455	246.0489753	238.9209373
Sesame	5331578.5		5331578.5
Virtuoso	18053.02541	146821.9723	79709.34322
Grand Total	1417.802864	5843.555579	3644.392662

Query 6



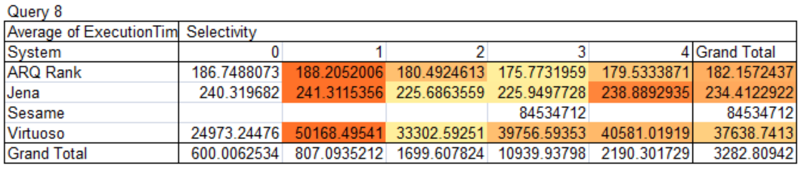
Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	182.9985462	180.7748279	184.0377312	160.9562347	179.0137151	166.0617346	175.6788586
Jena	244.1030689	238.3694885	239.562518	207.0524823	233.4777745	207.8315634	228.3555329
Sesame				123714143		56748453	90231298
Virtuoso	16110.05476	15555.41143	17808.1371	10630.01093	13306.70219	21125.2933	15740.11209
Grand Total	906.7300706	893.9538363	1039.354246	16160.47331	809.1120091	8474.873548	4725.4183

Query 7



Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	180.1373187	188.0059865	170.4203277	193.8015359	169.6933798	180.1753786	180.3723721
Jena	237.1630277	236.8525117	207.360765	240.7060911	213.4122379	225.6923077	226.77718
Sesame							
Virtuoso	16182.38199	16702.625	11121.33756	17514.17765	11315.13598	13746.91153	14346.17369
Grand Total	869.8350741	988.4969876	721.8283267	979.2462649	690.1374172	838.5716088	847.3916933

Query 8



Average of ExecutionTim	Selectivity					
System	0	1	2	3	4	Grand Total
ARQ Rank	186.7488073	188.2052006	180.4924613	175.7731959	179.5333871	182.1572437
Jena	240.319682	241.3115356	225.6863559	225.9497728	238.8892935	234.4122922
Sesame				84534712		84534712
Virtuoso	24973.24476	50168.49541	33302.59251	39756.59353	40581.01919	37638.7413
Grand Total	600.0062534	807.0935212	1699.607824	10939.93798	2190.301729	3282.80942

Query 9

Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	183.9821285	186.3884886	185.1682655
Jena	237.1349292	240.9405541	239.0525908
Sesame	104160	139242	121701
Virtuoso	16855.69587	19226.26823	18048.35323
Grand Total	779.4842904	878.3450883	828.7285506

Query 10

Average of ExecutionTim	Selectivity					
System	0	1	2	3	4	Grand Total
ARQ Rank	186.69854	180.5814755	175.0615292	195.0952381	185.3704458	184.5009185
Jena	244.1781935	238.8718385	223.5810069	242.1416833	237.7712403	237.2204934
Sesame						
Virtuoso	18730.61538	16896.49642	12597.885	34739.41986	16287.51648	20008.38397
Grand Total	1086.565222	945.8913893	716.7953498	1862.882334	970.3478216	1112.608455

Query 11

Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	182.5354016	182.7352867	185.8660823	190.4014545	182.501351	189.2143564	185.5710112
Jena	242.4558736	259.2814281	239.8745734	240.904302	225.6871183	239.8076583	241.2357485
Sesame	176781					121176	148978.5
Virtuoso	12717.52151	14202.4	31335.07849	24382.30769	11609.92573	11826.41713	17541.57758
Grand Total	835.1229933	840.7298218	1554.665244	1326.048065	747.5647268	766.9802513	1014.913437

Query 12

Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	179.7899187	187.7174407	183.7755235
Jena	240.8718174	240.6308285	240.7505695
Sesame		169975	169975
Virtuoso	14931.47053	15633.03025	15296.71051
Grand Total	857.0873993	953.7885988	905.8009685

Query 13

Average of ExecutionTim	Selectivity			
System	0	1	2	Grand Total
ARQ Rank	181.9272618	176.4400147	178.6672874	179.0293097
Jena	240.233401	234.9800058	227.168643	234.1501139
Sesame	36469554			36469554
Virtuoso	20357.12367	18903.24896	13132.96491	17581.54423
Grand Total	3446.673233	1060.22818	769.0367373	1768.832262

Query 14

Average of ExecutionTim	Selectivity			
System	0	1	2	Grand Total
ARQ Rank	182.849098	178.8747556	176.3096559	179.3566559
Jena	240.2596196	234.9560933	228.6801111	234.6448548
Sesame	165151			165151
Virtuoso	12720.52006	13369.7215	9321.982804	11733.37634
Grand Total	782.9768222	782.7709469	639.2902979	735.1676079

Query 15

Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	187.3628051	182.2840464	184.8235294
Jena	246.0343281	235.7397128	240.8563327
Sesame		382692	382692
Virtuoso	16197.04986	15755.09375	15980.88235
Grand Total	960.9077777	932.3839463	946.6226395

Query 16

Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	179.0857577	182.2339238	189.8033694	186.7164825	187.6867943	186.8809045	185.4400082
Jena	237.4885965	236.0877454	239.0975259	237.8302712	238.2284546	241.3028915	238.357909
Sesame	97070	209285					153177.5
Virtuoso	12315.02051	12499.13988	13897.40385	12234.78056	11972.2235	11659.79167	12432.4113
Grand Total	826.9017259	761.3438619	835.3209506	754.2642435	733.9278613	738.4376356	775.1043233

Query 17

Average of ExecutionTim	Selectivity			
System	0	1	2	Grand Total
ARQ Rank	185.9462981	184.1156222	179.2948655	183.112445
Jena	241.9513739	240.3013205	222.3735992	234.8482378
Sesame				
Virtuoso	71680.2656	114274.7771	26505.73533	69364.74545
Grand Total	3317.972369	5173.343011	1420.632603	3283.07504

Query 18

Average of ExecutionTim	Selectivity						
System	0	1	2	3	4	5	Grand Total
ARQ Rank	191.4067964	178.9480745	178.0401753	179.3863357	187.9423868	183.9400585	183.2870556
Jena	237.9575758	233.9703196	239.1547105	236.7763541	240.1508396	237.9700115	237.654452
Sesame		151310			2969546		1560428
Virtuoso	23219.05363	17216.75096	28135.0659	21680.59831	50233.49077	32016.69189	29663.44094
Grand Total	1162.151993	793.8621486	1435.657247	1178.433134	2964.709353	1690.549736	1543.403588

Query 19

Average of ExecutionTim	Selectivity			
System	0	1	2	Grand Total
ARQ Rank	181.0416107	183.2660786	176.7804399	180.3671062
Jena	234.9586177	237.3481078	229.3366627	233.8899714
Sesame	1392237			1392237
Virtuoso	13243.24626	14292.00145	11653.97799	13042.03487
Grand Total	893.4340663	828.1672953	731.9498533	818.3056472

Query 20

Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	182.9074379	187.4187799	185.1805119
Jena	240.4656354	238.4284343	239.4370123
Sesame	539009.5		539009.5
Virtuoso	18970.42831	17246.56682	18104.10607
Grand Total	1114.631974	985.9318096	1049.736218

Query 21

Average of ExecutionTim	Selectivity		
System	0	1	Grand Total
ARQ Rank	185.8129868	181.1810569	183.4872842
Jena	237.2589451	235.2481203	236.2478309
Sesame			
Virtuoso	252850.6598	232893.943	243821.8548
Grand Total	7624.844335	5824.07975	6722.481388

Execution Time of the Query according to different Limit value:

Query 1

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	180.9560054	177.9591935	180.6392828	193.0536869	183.1721435
Jena	229.3119119	227.1840834	234.8248588	254.6057487	236.5096214
Sesame			139069.5		139069.5
Virtuoso	11705.73333	11875.61618	13051.47104	11882.32673	12115.63316
Grand Total	756.7531228	767.6210069	785.1185123	718.4815003	757.0976126

Query 2

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	180.4494261	184.3622895	179.71717	197.2095621	185.8939054
Jena	237.7119184	241.0578725	230.2264112	258.6552337	242.5456837
Sesame		933359			933359
Virtuoso	233586.7367	283492.4605	215311.1455	393257.1352	289771.6029
Grand Total	8439.225912	10489.85511	7576.547837	15617.24206	10743.40445

Query 3

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	177.6390399	178.8780615	178.6677552	194.3416958	182.4686696
Jena	232.767317	226.4140271	230.4792087	251.2553558	235.4056032
Sesame		61085		99136	80110.5
Virtuoso	13510.72231	13215.24187	13777.91573	13456.69309	13497.16165
Grand Total	866.5985839	782.362311	843.6838847	770.2268076	816.8322654

Query 4

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	172.2933447	173.9362942	175.6778532	184.6802666	176.7752786
Jena	221.6910259	222.8725396	223.3062977	239.2829099	226.8525035
Sesame					
Virtuoso	10622.61856	10309.81702	10944.80758	10057.67096	10510.66698
Grand Total	641.5196892	690.7623804	688.0326485	660.1782711	670.4419396

Query 5

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	183.2432383	179.27108	181.4173982	192.3242174	183.9376897
Jena	229.6248934	234.5114845	238.464065	252.3960674	238.9209373
Sesame			5331578.5		5331578.5
Virtuoso	13806.57727	15720.19222	202931.7029	48375.47044	79709.34322
Grand Total	769.0495096	873.4871962	9193.000421	1965.062121	3644.392662

Query 6

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	172.9247453	170.7050125	173.4881918	186.8217822	175.6788586
Jena	222.9875591	222.1939451	221.5472125	248.5092089	228.3555329
Sesame		90231298			90231298
Virtuoso	15971.77687	15300.66608	15018.40974	16691.31276	15740.11209
Grand Total	929.9714957	15589.61285	866.2834602	948.3533883	4725.4183

Query 7

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	172.069508	176.6887218	179.66007	191.5084929	180.3723721
Jena	215.9115776	221.4343454	223.9982974	243.9188347	226.77718
Sesame					
Virtuoso	14052.89655	14126.91923	14813.88342	14364.36092	14346.17369
Grand Total	868.2442784	817.7900822	846.2644308	858.3006719	847.3916933

Query 8

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	179.4426306	175.7130787	173.1959964	197.9177691	182.1572437
Jena	231.9491859	224.0484909	231.2489608	248.9067442	234.4122922
Sesame	84534712				84534712
Virtuoso	15865.14541	22374.85333	25887.01385	78481.08389	37638.7413
Grand Total	7508.006122	936.1591938	1088.900259	3033.294939	3282.80942

Query 9

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	178.5661157	181.884369	180.1183242	198.4983488	185.1682655
Jena	229.914717	234.492236	233.9218098	255.9893352	239.0525908
Sesame		121701			121701
Virtuoso	15624.05398	17936.15385	18929.55586	19576.62385	18048.35323
Grand Total	702.7662966	850.5698324	877.4478912	887.9283244	828.7285506

Query 10

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	177.4035985	181.4616684	173.1600778	203.5429165	184.5009185
Jena	231.293057	233.6212925	227.9244882	254.3882969	237.2204934
Sesame					
Virtuoso	18492.82213	20390.89528	19875.07762	21077.73529	20008.38397
Grand Total	1043.550727	1060.378087	1113.888609	1217.013118	1112.608455

Query 11

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	183.9851515	182.3287749	181.376236	194.3053212	185.5710112
Jena	232.6156199	245.1109123	229.9532564	257.8509036	241.2357485
Sesame			176781	121176	148978.5
Virtuoso	17115.31754	17900.17078	16121.42413	19062.28957	17541.57758
Grand Total	963.5167754	1052.096254	947.7981728	1104.230948	1014.913437

Query 12

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	186.0076911	175.2420878	179.777259	195.4846817	183.7755235
Jena	238.8462851	237.5159738	233.8266643	255.0202791	240.7505695
Sesame			199778	140172	169975
Virtuoso	16143.64444	16085.46679	15214.81932	13676.87838	15296.71051
Grand Total	915.9576598	935.194258	876.9826196	896.1637931	905.8009685

Query 13

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	173.6978391	175.8577198	175.3385251	190.6908946	179.0293097
Jena	229.7729673	230.7490369	223.7111111	252.0939713	234.1501139
Sesame				36469554	36469554
Virtuoso	17073.13078	19579.08348	16459.55455	17062.34637	17581.54423
Grand Total	980.5265853	1094.990212	938.7959708	4004.237711	1768.832262

Query 14

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	175.0303867	174.3446108	177.5961571	191.6689354	179.3566559
Jena	229.3929743	232.5776375	227.4452731	249.691343	234.6448548
Sesame			165151		165151
Virtuoso	11029.52952	11765.24593	12688.92516	11525.27894	11733.37634
Grand Total	679.2685497	753.8068385	757.2918168	750.5186144	735.1676079

Query 15

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	188.0640367	178.6514627	175.2707222	195.4768643	184.8235294
Jena	233.1132603	233.9453552	232.9639296	259.3024505	240.8563327
Sesame			455712	309672	382692
Virtuoso	16360.53484	15355.66551	17059.69106	15390.51901	15980.88235
Grand Total	953.032386	914.4183348	1009.642686	920.4970288	946.6226395

Query 16

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	179.3981181	183.3283609	184.1050666	194.2744917	185.4400082
Jena	229.573821	231.7852406	233.531274	256.745336	238.357909
Sesame		153177.5			153177.5
Virtuoso	11796.11014	12426.10352	12539.89455	12974.37943	12432.4113
Grand Total	758.1779398	749.2774577	794.5459419	794.805677	775.1043233

Query 17

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	184.5236308	179.3578199	179.3477666	189.7729674	183.112445
Jena	237.1936696	230.7946413	227.618823	244.3540258	234.8482378
Sesame					
Virtuoso	102628.543	61005.81508	31194.23092	84427.62448	69364.74545
Grand Total	4891.61946	2811.628486	1645.660174	3829.04764	3283.07504

Query 18

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	183.3662171	184.4569415	176.3707902	189.4526144	183.2870556
Jena	234.6524279	234.0377358	235.6214712	247.6061321	237.654452
Sesame	151310	2969546			1560428
Virtuoso	16666.12979	19764.71727	20009.77636	63962.16082	29663.44094
Grand Total	846.9744422	1268.035643	1136.89226	3117.198836	1543.403588

Query 19

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	177.9538632	174.4903047	173.8634277	195.0988324	180.3671062
Jena	228.023033	230.5697303	226.2476959	249.9037873	233.8899714
Sesame		1392237			1392237
Virtuoso	13663.53137	12621.70675	12333.66667	13496.76481	13042.03487
Grand Total	794.3881119	881.4724109	756.6328016	825.4828256	818.3056472

Query 20

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	182.3293026	179.4730695	188.4050472	194.7599478	185.1805119
Jena	237.7431224	233.6981589	241.977591	248.1590498	239.4370123
Sesame		697819	380200		539009.5
Virtuoso	17042.03795	17066.72257	20006.35356	19010.71854	18104.10607
Grand Total	955.2758924	1016.347863	1138.877493	1140.925469	1049.736218

Query 21

Average of ExecutionTime	Limit				
System	1	10	100	1000	Grand Total
ARQ Rank	180.5238953	178.2693975	178.6566421	198.2144285	183.4872842
Jena	232.3789491	232.9508743	229.3373743	251.0012672	236.2478309
Sesame					
Virtuoso	198986.3601	353240.4167	264507.9236	99562.67347	243821.8548
Grand Total	4768.568148	10517.72436	8864.592466	2544.321282	6722.481388