Design and Evaluation of an IR-Benchmark for SPARQL Fulltext Queries

Masterarbeit im Fach Informatik
Master’s Thesis in Computer Science
von / by
Arunav Mishra

angefertigt unter der Leitung von / supervised by
Prof. Dr. Martin Theobald

begutachtet von / reviewer
Prof. Dr. Gerhard Weikum

Januar 2013
Hilfsmittelerklärung

Hiermit versichere ich, die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben.

Non-plagiarism Statement

Hereby I confirm that this thesis is my own work and that I have documented all sources used.

Saarbrücken, 29. January 2013,

(Arunav Mishra)

Einverständniserklärung

Ich bin damit einverstanden, dass meine (bestandene) Arbeit in beiden Versionen in die Bibliothek der Informatik aufgenommen und damit veröffentlicht wird.

Declaration of Consent

Herewith I agree that my thesis will be made available through the library of the Computer Science Department at Saarland University.

Saarbrücken, den 29. Januar 2013,

(Arunav Mishra)
To Nature

- Arunav
Abstract

In this thesis, we design a new IR-benchmark that aims to bridge the prevailing gap between traditional keyword-based retrieval techniques and semantic web-based retrieval techniques. We present a unique, entity-centric data collection, coined **Wikipedia-LOD**, that aims to combine the benefits of both text-oriented and structured retrieval settings. This collection combines RDF data from DBpedia and YAGO2 structured Knowledge Bases (KBs), and textual data from the contents Wikipedia articles into XML-ified documents, called the **Wiki-XML** documents, corresponding to every Wikipedia entity. To evaluate such a collection, we introduce a new query format, called **SPARQL-fulltext (SPARQL-FT)** queries. We design the SPARQL-FT query format by extending the W3C standard SPARQL with additional **FTContains** operator that constraints an entity by a set of keywords representing a fulltext condition. We design a query benchmark of 90 queries by manually translating Jeopardy-style Natural Language (NL) questions into the SPARQL-FT queries. We present the Wikipedia-LOD (v1.1) as a core collection for the newly introduced **INEX 2012-LOD track**, which defines three tasks over the collection, namely, Ad-hoc retrieval task, Faceted retrieval and the new Jeopardy task. For the **Jeopardy task**, we provide the query benchmark designed in this thesis to evaluate participating engines.

In this thesis, we further describe indexing, ranking, and query processing techniques that we implement in order to process the new kind of SPARQL-FT queries, provided in the context of Jeopardy task of the INEX 2012 Linked Data track, by introducing the **SPAR-Key** engine. For the rapid development of the new query engine that could handle this particular combination of XML mark-up and RDF-style resource/property-pairs, we decide to opt for a relational-DBMS as storage back-end, which allows us to index the collection and to retrieve both the SPARQL- and keyword-related conditions of the Jeopardy queries under one common application layer. Additionally, our engine comes with a rewriting layer that translates the SPARQL-based query patterns into unions of conjunctive SQL queries, thus formulating joins over both the DBpedia triples and the keywords extracted from the XML articles.

Finally, we perform a detailed evaluation of the effectiveness of our query engine by processing the benchmark queries. We present the results from the official INEX’12 evaluations for the Jeopardy task that was performed with Ad-hoc search style relevance assessments, obtained with the help of crowd sourcing. However, we show that such an evaluation does not truly comply with the task definition, and hence a re-evaluation with a QA-style assessment is required. For the re-evaluation, we create gold result set, or ground truth, by mapping already known correct answers of the NL questions to the Wikipedia entities. By outperforming our competitors in terms of MRR and NDCG, we show definite advantages of exploiting both structured information and unstructured information to improve Question-Answering and Entity-retrieval tasks.
Acknowledgements

I hereby express my sincere gratitude towards my supervisor Prof. Dr. Martin Theobald for providing me immense learning experience throughout my masters studies. The valuable input given in his lecture and his guidance during the course of this thesis work gave me the opportunity to learn both theoretical and practical aspects of Information Retrieval. I am also highly obliged to him for giving me an opportunity to participate in INEX’12 which provided me an interface with the larger research community and exposed me to the latest trend and directions in various areas of research being pursued.

I am highly grateful to Prof. Dr. Gerhard Weikum for reviewing my thesis. I would like to thank him for being an inspiration for the students working in his group and motivating them to achieve excellence.

I would take this opportunity to extend my gratefulness to Avishek, Sharat and Sairam for their technical help on different occasions. I would also like to thank Arnab, Bibek, Binod and Bhawna for supporting me and keeping my morals high during crunch times.
Contents

List of Tables 105

A Topics for the Evaluations 107
   A.1 50 Topics Selected for Official INEX Evaluations 107
   A.2 26 Topics Selected for INEX Re-Evaluations 107
   A.3 11 Topics Selected for INEX Re-Evaluations 108

B SPARQL-FT Queries constituting the Query Benchmark 109

Bibliography 131
Chapter 1

Introduction

In this thesis our main goal is to design and evaluate an entity-centric data collection, unifying structured RDF data and unstructured textual data into XML-ified documents where each document corresponds to a Wikipedia entity. To undergo this investigation we also introduce a unique coalition of structured and unstructured queries, by leveraging the W3C standard SPARQL query language with fulltext conditions. In this chapter we motivate our work, and enumerate contributions of this thesis.

1.1 Motivation

In the last decade, the World Wide Web (WWW) underwent an unprecedented growth leading to drastic development of Information Retrieval (IR) techniques over Web documents. Amongst many attempts to realise a fully automatic Information Retrieval system, Web search engines facilitating keyword-based fulltext searches (like Google and Bing) stand out as the most important ones. These Web search engines focus on making the information encoded in the WWW discoverable by searching over various forms of content and automatically retrieving relevant information satisfying user needs. Traditionally, a Web search engine accepts a set of keywords from a user that describes his information needs, and returns a ranked list of documents ordered by a notion of relevance to the user query.

However, in recent years, the movement of organising data into one large source over the Web or creation of so-called Semantic Web (SW) has emerged and is growing, as more and more data organisers are becoming a part of this movement. As coined by Tim Berners-Lee, the main idea behind the Semantic Web is to convert the current Web,
dominated by unstructured and semi-structured documents, into a ‘Web of Data’. This conceptually simple idea has led to the creation of initiatives like the Linked Open Data (LOD) \(^1\) cloud and is getting closer to being realized with incremental expansion of the LOD cloud each year. The notion of LOD allows structured data sources to be interlinked and stored within a generic Resource Description Framework (RDF) format [Kly04].

As an immediate application, these sources are being exploited as a form of knowledge representation, or so-called Knowledge Bases (KBs). These KBs are being viewed as integral data models to improve IR tasks, query rewriting or reformulation, and even for complete Question Answering (QA)-style query translations from natural language input questions [Van11]. Apart from this, scholars have also explored the applications of data, organised under the SW, in industry and its social potential in many domains of research [Fei07].

For Question-Answering (QA), one may very well extend to large-scale open-source KBs like DBpedia [Biz09b] and YAGO [Fab07] containing more than one billion facts or pieces of information (RDF triples). These KBs make a light weight representation of the SW [Biz09a] and since they are developed within the RDF, information can be extracted by writing structured queries over them in languages like SPARQL [Pru]. However, even with their extensiveness and fairly broad coverage, these domain-independent KBs may not prove to be exhaustive sources for processing a large body of Natural Language (NL) queries [Lop11]. Moreover it will be unreasonable to expect an end user to fully understand the complex logic based formulation of the SW and to write specific queries in languages like SPARQL. On the other hand, we cannot expect even the most sophisticated engines to automatically translate such NL questions into structured queries and efficiently process them over these KBs to obtain high quality results.

As we argue about the insufficiency of structured data to automatically answer NL questions, let us also examine the comparability of keyword-based retrieval techniques for the purpose. Keyword queries (or "key concepts") prove to be more flexible and user friendly than the logic based semantic queries. Moreover, keyword-based retrieval techniques for text documents are able to return relevant documents or passages based on textual similarity using relevance measures. However, even state-of-art keyword retrieval models often fail in handling descriptive queries or the Wh-queries (like “what”, “who”, “where”, “why”, etc.). Often they tend to be too specific or unduly generic and it becomes difficult to retrieve very specific information from an unstructured text collection by this retrieval model. An end user often has to try a combination of keywords or go through a set of documents containing unstructured text to extract the exact information. Moreover this retrieval paradigm is unable to retrieve information by combining two or more documents.

\(^1\)http://linkeddata.org
The following scenario illustrates some of the above mentioned short-comings of SW-based retrieval and keyword-based retrieval techniques for QA task:

**Scenario#1:** An IR student, newly exploring SW, attempts to find an answer to the question: “Which mountain range is bordered by another mountain range and is a popular sightseeing and sports destination?” by formulating a SPARQL query over DBpedia. For this, she uses the *DBpedia SPARQL endpoint* and issues the following query:

```
PREFIX DbOnto: <http://dbpedia.org/resource/>
SELECT ?p WHERE {
}
```

Note that to formulate the above query, the student 1) has to have syntactical knowledge of the complex join structure in SPARQL, 2) has to be knowledgeable of the exact formulation of DBPedia entities, and 3) has to be aware of the existence and usage of relation `<border>` and class *MountainRange* in DBpedia. Moreover, she has to represent the above entities and relations with their exact URIs, for example, *MountainRange* if changed to *mountainRange* will not yield any results. One may also note that with such a query, she is incapable of expressing the second part of the question, i.e., “is a popular sightseeing and sports destination” as a triple (or structural constraint) due to absence of appropriate entities and relations (in the current DBpedia). Due to this, the above query also returns *mountain ranges* that are do not answer the given entire NL question.

**Scenario#2:** An enthusiast of TV shows tries to find answer to this Jeopardy question for 800 $: “This mountain range is bordered by another mountain range and is a popular sightseeing and sports destination”. She tries to find the answer through a well known keyword-based search engine, for instance Google search, by issuing the entire NL question as a keyword query. The search engine returns a list of relevant documents based on a full text search using combination of statistical relevance measures (Figure 1.1). From the top results its is evident that such a

---

2[www.dbpedia.org/sparql](http://www.dbpedia.org/sparql)

3[www.google.com](http://www.google.com)
Figure 1.1: Top 5 results by Google for query “This mountain range is bordered by another mountain range and is a popular sightseeing and sports destination” (as on 24th Jan, 2013)

query formulation is incapable of retrieving any relevant document to the answer. This shows the limited reasoning capabilities of the system from keyword queries being perceived as a set of independent terms.

Hence, we observe that formulating questions in SPARQL-style query languages makes the impractical assumption of user-knowledge on the underlying KB. Moreover, resolution and disambiguation of mentions to entities (and strings to relations) is a major issue in formulating such queries. On the other hand keyword-based searches lack reasoning to capture user intent, and return documents which can be relevant to the query but not to the precise answers.
Efficient QA systems with high precision results still remain an open research challenge. With the advent of the SW there has been a recent trend to focus on linked RDF data for answer NL questions [Van11]. However, most of the existing approaches exploit only one of these (either structured or unstructured) retrieval modes in isolation, while the strength could often lie in merging the underlying data sources into one. IBM’s Watson [Fer10], for example, does not issue SPARQL queries but pursues a combination of more than 100 individual retrieval models and returns its best calculated answer based on a combination of these individual (both knowledge-based and ranking-based) models; yet each of these models uses either structured or unstructured data sources as input.

Our goal in this thesis is to combine knowledge retrieval and keyword-based document retrieval techniques over one unified data collection (Figure 1.2), where the knowledge retrieval is capable of resolving ambiguities and of providing a better interpretation of a user’s intent, and the document retrieval is capable of capturing content similarity to compensate for missing or unknown structured information and to give flexibility to the query. To achieve this, we present: 1) a unique Wikipedia-LOD collection of XMLified documents, comprising all DBpedia and YAGO2 [Hof11] triplets about entities (as either their Subject or Object) and their associated fulltext contents of the corresponding Wikipedia articles; 2) a query benchmark containing 90 manually translated SPARQL-FT queries from Jeopardy-style NL questions, which are syntactically similar to SPARQL queries with additional fulltext conditions; 3) SPAR-Key – An engine to process SPARQL-FT queries over the proposed data collection and build run results; 4) an evaluation proposed unified model of retrieval and a comparison to standalone...
traditional keyword-based retrieval; 5) an overview of newly introduced INEX’12 LOD track.

1.2 Contributions

We make the following contributions during the course of our work:

- **We create a unique unified data collection containing XML documents** [Aru12a]. The design of this new collection is based on merging structured data corresponding to a Wikipedia entity, from DBpedia and YAGO2 ontologies, and unstructured textual data from corresponding Wikipedia article into one document in XML format. This makes the collection *entity-centric* and equips it with properties that are essential for the investigations undertaken in this thesis.

- **We build a query benchmark containing manually translated (Jeopardy-style) NL questions into SPARQL-FT queries** [Aru12a]. The new SPARQL-FT queries represent a fusion of structured semantic queries and unstructured keyword queries. The main aim is to introduce a query format that can be processed on the data collection and exploit the expedient properties.

- **We propose a query engine model to process SPARQL-FT queries over the proposed data collection and compare the results** [Aru12b]. The idea is to design a simple model that could process the queries and return results as target entities. We compare results of our hybrid-retrieval technique to those of traditional keyword based techniques over the data collection and derive conclusions of our investigation. We also note that a simple model accomplishes our goals even though it may not prove to be the most efficient as efficiency is not the subject of investigation.

- **We organised and participated in the INEX’12 LOD track** [Qiu12]. We provided the proposed Wikepeida-LOD(v 1.1) data collection as the core collection of the new INEX’12 LOD track. This track defines three tasks over the collection namely, Ad-Hoc Task, Facet Search Task and Jeopardy Task. The query benchmark with 90 SPARQL-FT queries was made available for the Jeopardy Task. As a participation in this task we submitted results from an initial prototype of the query engine presented in this thesis.

We continue our efforts in INEX’13 LOD track by upgrading the Wikipedia-LOD collection to version 1.2 and providing a new query benchmark for the Jeopardy task. As a participation in the Jeopardy task for INEX’13, we plan to submit results from the final designs of query engines presented in this thesis.
This thesis forms a basis for the following publications [Aru12a, Aru12b, Qiu12].

1.3 Overview of the Thesis

The remainder of the thesis is structured as follows. In Chapter 2 we discuss the existing perspectives and approaches that are related to that of ours. We also analysis works that attempt to solve a similar problems to those targeted in this thesis. In Chapter 3 we discuss in details about the proposed data collection. In Chapter 4 we give the necessary details and insight to the new query benchmark. In Chapter 5 we present the storage model of our engine in Oracle relational database and our approach to process a query on our data storage model. We discuss the INEX’12 LOD track in Chapter 6. In Chapter 7 we present the experimental evaluations and compare our engine to INEX competitors. We also perform experiments to evaluate the various strategies introduced in this thesis to process and present the results in this Chapter. Finally, in Chapter 8 we draw conclusions to this thesis and presents our future aspirations and goals in continuation to this work.
Chapter 2

Related Work

In this chapter we analyse and discuss some systems that are related to the approaches and ideas presented in this thesis. Broadly, our objective invites research work from the database, IR and SW community. This opens up a wide horizon of research done by the three communities. Some attempts [San02, Elb11a] have been made to understand the advantages for merging semantic-based knowledge retrieval and keywords-based retrieval. Before looking in depth into these closely related works, for simplicity let us broadly categorise our work and analyse some publications in each of these categories.

1. Storing textual data in relational-DBMS.
2. Storing and indexing RDF data in relational-DBMS.
4. Combining keyword and semantic searches.

The Wikipedia-LOD collection can be broken up into structured and unstructured textual part, and we also design a unified storage model on a relational database (described in the following chapter). We analyse the approaches to store each type of data individually so as to design a unified data storage model. Finally we look at some published query formats that work on a combined data model similar to ours.

2.1 Storing Textual Data in Relational-DBMS

Relational Database Management Systems (relational-DBMS) become the most feasible option when it comes to large scalable storage systems. Traditionally, in the IR community,
the usage of relational-DBMS to implement retrieval techniques has not been very popular. The primary reason is their slow performance in SQL, complexity of implementation, constrained element manipulation, lack of integration with existing libraries and storing overhead. However, the database systems do have advantages compared to most popular file-based systems, which are – recovery management, memory management, data transfers avoidance and data security. Implementation of search engines in SQL and usage of relational-DBMS is narrowly adopted. We suppose, it is mainly due to the prevailing gap between the research of the database community and the IR research community. Some published works [Gar08, Dav92, Dav94, Dav97, M. 00, Lun99, Hol03], implement the traditional text retrieval techniques in a relational-DBMS.

In this thesis, we intend to use relational-DBMS to design a storage model for unstructured data while the pre-processing (converting unstructured textual data into structured relational data) is performed outside the relational-DBMS. We believe that the implementation of specialised techniques like state-of-art ranking functions with user-defined-functions of a relational-DBMS, would be too complex and arguably inefficient. In this section, we analyse some attempts to follow a relational-DBMS based paradigm for information storage and retrieval.

- [Hol03] attempted to present a simple relational database schema for efficiently processing keyword queries. However, it did not address the challenge of converting unstructured data into structured data through a relational-DBMS implementation, rather preferred to do this with a standard implementation in the C programming language. As one of the early works, this proved that a framework creation for a relational-DBMS based text retrieval engine does not require a proprietary application (relational-DBMS) code which was one of the major bottlenecks for researchers in this direction.

- [Gar08] aimed to bridge the gap between information retrieval and databases. This project divides into two phases: 1) storage phase and 2) retrieval phase. For the storage phase, this approach simulates a baseline inverted file technique (a technique used by file systems) [Zob06]. In other words, they store primarily two structures: first structure contains every term in the corpus as a primary key and pointer to a document as a foreign key; and the second structure contains every term and their associated frequency set in corresponding documents as descriptor vectors. They argue that storing vectors in the relational model is difficult, and hence assume that a vector as a single column. Figure 2.1 shows the database schema of their storage model.
The systems analysed above, prove the feasibility of adopting a relational-DBMS based paradigm to store textual data. However, they also point out the fact that it becomes difficult to implement a complete automated IR system capable of handling dynamic index updates, implementing state-of-art ranking functions, etc., with the proprietary code of a commercial relational-DBMS. Viewing this, in our design, we exploit the advantages of scalability and efficient query processing of the relational-DBMS, while performing the preprocessing outside.

2.2 Storing RDF Data in Relational-DBMS

Over the past decade, after the advent of the Semantic Web, there have been many perspectives presented for viewing the semantic data expressed under the Resource Description Framework (RDF). One among many, is the relational perspective [Luo12]. This is a perspective brought by the database community which views RDF data as a special type of relational data. Following this perspective they propose two alternatives to represent data: a) vertical representation and b) horizontal representation.

- **Vertical representation** proposes that the entire data can be stored as a single relational table with the schema \((\text{Subject}, \text{Predicate}, \text{Object})\) with each column storing a component of a triple. Such a table is referred to as a *triple store*. Many real-word systems that are based on this idea, often have an additional column called the context column. This extra column is used to identify the dataset from which the triple originated.

- **Horizontal representation** proposes that there should be one column for every *Predicate* and one row for every *Subject*. The *Object* constituting a *Subject* and a
Predicate in a triple should be stored in the field at \((\text{Subject, Predicate})\). Such a representation leads to a very large table (for a very large collection) which is mostly very sparse. However, this forms the core idea of this type of representation but the actual layout is application dependent.

We further elaborate the different prospectives for RDF data in Chapter 5, Section 5.2.1. In this section we follow a discussion on the above mentioned data representations as we use them to design our system.

### 2.2.1 Relational Representation of RDF Data

As already introduced, a vertical representation is essentially a large triple store – there are many popular systems [Bro02, Cho05, HG03, Zou11, Aba09, Kev03] that follow this methodology of storing data. Let us look at some of the important models that represent the majority of the ideas with this perspective.

- **Sesame** [Bro02], proposes a storage-independent architecture for storing and efficient querying RDF and RDFS data\(^1\). This system presents a Storage And Inference Layer (SAIL) API that takes care of the integration with any underlying relational-DBMS. Moreover, a SPARQL query, which is a standard for RDF data, is translated into a native query language namely RQL that is processed over the database. This system builds a database in the underlying relational-DBMS, with multiple tables storing different meta-data. Figure 2.2 shows the data storage schema of Sesame.

- **3-store** [HG03] claims efficient query processing while maintaining a relational triple store. The 3-store almost follows the same architecture, as [Bro02], but relies on the relational-DBMS for query processing. In addition it uses a unique hashing scheme to achieve space optimisation to store RDF data. It supports RDQL queries which are translated into SQL. However, for our purpose, it would become difficult to implement a similar engine with complex hashing for SPARQL-FT queries with additional information from keyword conditions.

- In the run of achieving efficiency, **Jena2** [Kev03] claims to beat its predecessors by introducing a simple yet effective storage schema. In Jena2 the literal values of triple components are directly stored in the statement table (a large table storing all the triples). It is based on the Jena, with a more generalised database schema proposed for underlying relational-DBMS. In addition, Jena2 introduces the

\(^1\)http://www.w3.org/TR/1998/WD-rdf-schema-199980409
concept of property tables to achieve its efficiency goals. The idea is to build separate relational tables called the property tables for storing the Subject-Predicate pairs with a schema (Subject, Predicate). The authors argue that by this, similar properties (or properties generally referenced together) could be clustered together and accessed with a single table look-up which increase the efficiency. Figure 2.3 shows an architecture diagram for building a triple store.

This property table approach actually comes under the horizontal representation of the RDF data. However, there exist some fundamental problems. Firstly, since in a wide table not all the Subjects will have all the properties, this property table will be a very sparse table with a lot of NULLs. Secondly, it would be inconvenient to express multi-valued attributes like books with multiple authors or a famous location having many names in a flattened-table representation. Finally, with every update, the data has to be re-clustered on the properties. Later, more advanced techniques [Aba09] were introduced to solve these problems.

- **SW-store** [Aba09] is also based on the horizontal representation of RDF data but it claims to solve the problems faced by Jena2. This publication introduced the concept of vertical partitioning. The idea is each Predicate column is materialized as a binary table over the schema (Subject, Predicate). Thus vertical partitioning could be considered as a special case of properties table where one property table

![Figure 2.2: Data store schema of Sesame [Bro02]](image-url)
is maintained per $Predicate$. This solves the mentioned problems with Jena2 while maintaining the efficiency by sorting the binary tables lexicographically to allow fast joins. Figure 2.4 shows sample data with possible properties tables created from it. The figure also shows a conjunctive query on the schema.

- As a commercial Semantic Web middleware, the OntoBroker [Jue11] currently becomes the most comprehensive and fastest implementation. As a commercial interface machine, it is the first, and so far the only product that supports all W3C Semantic Web recommendations. These are OWL, RDF, RDFS, SPARQL and, in addition, the industry standard F-logic. This systems employees distributed query processing architecture to achieve high efficiency and at the same time deal with high query rates. It provides a Web Service interface and can also be deployed in application servers.

The systems analysed in this subsection represent the state-of-art RDF stores. They solely focus on minimising the storage complexity of RDF data and achieving high efficiency in
processing a semantic query over it. We follow the vertical representation of the RDF data and design a triple store for our system. However, we do not follow the compression schemes presented by these systems as we aim to build an unified relational-DBMS based storage model for unstructured textual and structured RDF data.

### 2.2.2 Indexes Over RDF Data

All the systems discussed so far aim for translating the RDF data into a relational form that could be stored in a highly scalable relational-DBMS. Standard relational-DBMS provide functionality to build indexes over relational table to boost the look-up time of a record. This is the right way to go for us, considering the large size of data stored in a single relational table. Let us analyse some real world systems that follow this technique to boost their efficiency.
One of the first systems to propose the creation of unclustered indexes over RDF data to boost the look up time was RStar [Ma.04]. This system proposed creation of four sets of unclustered indexes over the storage schema. Since RStar focused on enterprise resource management i.e. to store a large amount of data at enterprise levels, it argues the usage of relational-DBMS over file systems. It builds on the popular IBM DB2 relational database management system. This system proposes creation of four different sets of indexes over their statement table or the main table storing the triples with schema (Subject, Predicate, Object).

1. The first set of indexes contains – indexes on each of the columns i.e., (Subject), (Predicate) and (Object) alone.
2. The second set of indexes contains – one composite index with a key pair, (Subject, Predicate) and an index on (Object) alone.
3. The third set of indexes contains – a single composite index on with a pair key pair, (Predicate, Object) and an index on (Subject) alone.
4. Finally the fourth set of indexes contain – a composite index on all the three columns as (Subject, Predicate, Object)

In this publication, the authors argue on the utility of such index creation and provide an analysis of the classes of queries that are processed efficiently due to such sets of indexes built on the statement table.

Building multiple indexes on the main table brings a large storage overhead as shown in [Ma.04]. A more recent publication, Parliament [Dav09], provides methods to reduce this storage over head by clever implementation with linked lists. However, traversing through a linked list may involve a lot of disk I/O which makes this approach less efficient.

RDF-3X [Neu08, Neu10] has six indexes to answer queries on any pattern with variables namely, SPO, SOP, OSP, OPS, PSO and POS. In addition RDF-3X adds projection indexes over the six indexes. One projection index is created for each subset of Subject (S), Predicate (P) and Object (O). This makes additional nine indexes over the six indexes specifically, S, P, SP, PS, SO, OS, OP and PO. The projection indexes become useful in solving aggregate queries by evading some intermediate join computations. In addition, these indexes are also used to build statistics for the cost-based optimiser used by RDF-3X.

RDF-3X also implements advanced selectivity estimation techniques to formulate query plans and offers data compression techniques to counter the space overhead for the additional indexes. For estimating selectivity, it builds statistical histograms over each order of entries of the triple. In addition, it exploits a frequent path
technique where the frequent paths in the data graph are precomputed and their exact join statistics are stored. To contain the storage space required, RDF-3X uses a compression scheme at a B-tree leaf block level.

- The popular relational-DBMS and RDF engine Virtuoso \cite{cit,Erl07} maintains the data in quad tables or tables with four columns namely, Subject (S), Predicate (P), Object (O) and Context (C). On this quad table, it can build indexes different permutations of SPOC columns (as per requirements). However, the default indexes are of type clustered B-tree over CSPO and OCPS columns (although other indexes can be enabled as pre requirements). This system introduces data-level as well as page-level compression schemes to reduce the storage overhead of multiple indexes. In addition, it builds bitmap indexes for each POC by default to exploit the fact that multiple Subjects having the same Predicate and Objects \cite{Erl07}.

The above mentioned systems present different indexing schemas and techniques to achieve efficiency in semantic query processing. In our systems, we build three composite indexes over all the columns of the triple store. Each of these three indexes is built for a column as the key. We note that we achieve decent efficiency in processing a semantic query with the three indexes. However, our goal is to investigate the effectiveness of the queries and not efficiency.

2.3 Structured Knowledge based Question Answering (QA)

Ontology-based Question Answering (QA) has emerged in recent years, with the main goal of providing concise answers to a Natural Language (NL) questions by exploiting the Semantic Web. This area of research has witnessed many closed-domain QA systems that presume that the knowledge is contained in one or a set of homogeneous ontologies. Some prominent works are \cite{Abr06,Cim07,Tab08,Wan07}. However, to scale the QA over open-domain or domain independent Knowledge Bases (KBs) (like the Linked Open Data), still remains a research challenge for the IR as well as the NLP communities. In this section we shall visit some systems that make use of ontologies to concisely answer NL Questions.

- QuestIO \cite{Tab08} as its core processing model translates a NL question into a formal query by initialising gazetteers for the underlying KB. It accepts user keyword queries of any length and form, and primarily works by recognising concepts in the query. This makes it independent of the individual terms in the query. It analyses relationships between concept pairs and finally uses a ranking model that combines
Figure 2.5: FREyA work-flow chart [Dam12]

string similarity measures, specificity of the relations and the distance between the terms. As a main contribution, it handles both conjunctive and disjunctive queries.

- **PANTO** [Wan07] is an interface that primarily translates a NL question into a SPARQL query. It then processes the SPARQL query over the underlying KB. It generates triples (*Subject*, *Predicate* and *Object*) from a NL question by parsing it with the Stanford NLP parser [Dan03]. These triples are mapped to the lexicon created while loading the KB into the system. It uses the two intermediate representation namely, Query-Triples, which are generated by linguistic analysis of a NL question and Onto-Triples, that are extracted using the lexicon, string distance metrics and WordNet. PANTO can handle both conjunctive and disjunctive queries.

- **FREyA** [Dam12] is a successor of QuestIO (described above), and achieves improvements to question translation. FREyA supports multi-domain ontologies and provides a better handling to ambiguities to queries than its predecessor. Like QuestIO, it uses the Stanford NLP parser to build a parse tree from a given NL question. In addition, it provides assistance to a user to formulate a query via clarification dialogues. The user choices are stored and are used as training data to improve performance. Given a user query, FREyA finds the ontology-based annotations in the query. These annotations are generated by an ontology based gazetteer called OntoRoot. Figure 2.5 illustrates the work flow of FREyA.

- A more recent, and the first system to perform QA over open-domain ontologies is **PowerAqua** [LFMS11]. The main idea is to overcome the the issues of knowledge acquisition bottlenecks for a homogeneous ontology by sparing over wide range
Figure 2.6: Architectural components of PowerAqua [LFMS11]

Almost all the systems mentioned above either focus on translating a NL question into a keyword query or semantic query. Some systems also present interactive interfaces to accept user inputs in desired formats. However, in this thesis we present a new query format and motivate that it becomes easier to express a NL question in this format. We describe a manual translation framework for translating a NL question into the desired format and leave its automation as future work.
Chapter 2 Related Work

2.4 Combining Keyword and Semantic Searches

In this thesis, we have so far motivated the advantages to combining knowledge retrieval and keyword based retrieval. In this section, we analyse some systems that attempt to investigate similar style of query processing. Some systems [Bik10, Kas08, Mag11, Bha08] present interactive interfaces to accept semantic and keyword queries, while more recent approaches [Elb09, Elb12] present a query formats by combining them. In this section, we analyse some of the above mentioned works.

- **K-Search** [Bha08] is an attempt to capture the flexibility of keyword search with the reasoning capabilities of semantic queries. This hybrid search engine supports three types of queries, namely pure semantic queries (i.e., identification of concepts/relations/instances), pure keyword queries, and keyword-in-context queries. For keyword-in-context queries, a keyword search is performed on a subset of documents annotated to nodes within a specific concept or relation. This prototype handles the keyword and semantic parts of a hybrid query by processing them separately and then finally merging the results. In order to merge the intermediate results from keyword search and semantic search, a conjunction is formulated. This is essentially an intersection over the two result sets as shown below.

\[
\text{ResultSet}_{hybrid} = \text{ResultSet}_{keyword} \cap \text{ResultSet}_{semantic}
\]

- **NAGA search engine** [Kas08] focuses on efficient semantic searches over knowledge bases organised as labelled directed graphs. It supports graph-based queries with regular expressions for so-called "expert users" to retrieve sub-graphs from the underlying KBs. To retrieve answers to such a query, the system searches for sub-graphs with matching query structure and labels, and binds the missing labels in the query. As a unique contribution the NAGA engine ranks the retrieve sub-graphs and present an ordered result list to user with a notion of relevance. Moreover, It also provides keyword search facilities for so-called “casual users” over the underlying KBs. NAGA introduces a new query language that is similar to the W3C standard SPARQL for writing queries. Like [Elb09], it also uses statistical language models but suffers from the too-few-result and too-many-result problems from some queries. Moreover, it does not support query relaxation and fails to handles queries representing coalition of keywords and semantic queries.

- [Elb09] become one of the first approaches to propose a definite query format to integrate key word queries into semantic queries. However, the main contribution of this work is a novel ranking model based on statistical language models for
Chapter 2 Related Work

<table>
<thead>
<tr>
<th>Example Query</th>
<th>Woody_Allen produced ?x . Woody_Allen directed ?x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example Query with keywords</td>
<td>Woody_Allen produced ?x{murder lover} . Woody_Allen directed ?x</td>
</tr>
<tr>
<td>Relaxed Queries</td>
<td>Woody_Allen ?y ?x{murder lover} . Woody_Allen directed ?x</td>
</tr>
<tr>
<td></td>
<td>Woody_Allen produced ?x{murder} . Woody_Allen ?y ?x</td>
</tr>
<tr>
<td></td>
<td>Woody_Allen produced ?x . Woody_Allen ?y ?x</td>
</tr>
</tbody>
</table>

**Figure 2.7:** Query framework [Elb09]

Keyword integrated queries. To represent the keyword-augmented queries they pitched the idea to extend the SPARQL language rather than introducing a new query language. Moreover, they presented efficient processing for three types of queries, namely exact queries (purely keyword or semantic queries), relaxed queries, and keyword integrated queries by solving too-few-results and too-many-results problems effectively. Figure 2.7 shows different formulation of a query.

In Figure 2.7, it is clearly illustrated that the keywords-augmentation, represented within the curly brackets, are associated to individual triples of the SPARQL query. Thus processing such augmented keyword queries independently results in the identification of relevant graph patterns which are then further cleaned by the SPARQL part. Even though this work becomes our primary motivation for extending the SPARQL language rather than introducing a new language, but remains different to ours in the augmentation format of the keywords to the SPARQL query. That is they motivate to associate keywords to an entire triple whereas we attempt to associate keywords to the entities occurring in the triple patterns (*Subject* or *Object*). Thus, with our representation, one can associate a triple with two sets of keywords to describe both *Subject* and *Object*.

- **GoNTogle** [Bik10] provides a framework for document annotation and retrieval within the traditional keyword-based and semantic-web based retrieval techniques. It supports a hybrid search by allowing a user to formulate queries that contain the underlying ontology classes and keywords over a document collection with semantic annotations to the ontology classes. To process such a query, it formulates a conjunction over the two components of a query (ontology classes and keywords) and reports an intersection of the result sets obtained from the standalone keyword search and semantic search.
\( \text{ResultSet}_{\text{hybrid}} = \text{ResultSet}_{\text{keyword}} \cap \text{ResultSet}_{\text{semantic}} \)

The above set notation models a hybrid search supported by GoNTogle. This \( \text{ResultSet}_{\text{hybrid}} \) contains ranked documents with an weighted average of the individual scores of a semantic search and keyword search. This is a naïve implementation and does not really exploit all the advantages of either semantic or keyword search techniques. However, in this thesis we follow a similar idea of preserving two components to a hybrid query and handling them separately.

- \cite{Mag11} provides a fairly similar approach to that proposed in this thesis with few fundamental differences. The key idea lies in the combinatorial representation of structured and unstructured data as an Entity-Relationship (ER) graph. From the related work’s perspective, the semantic data is interpreted as an Entity Relationship (ER) graph with nodes annotated with documents containing unstructured textual data. To create such a graph, they choose the YAGO2 \cite{Hof11} ontology (which forms the ER graph) and Wikipedia articles (which are the textual documents annotated to the nodes of the ER graph). This is where their approach stands in-line with that of ours. However, they define two search tasks on their core data collection namely, Context-Aware Fact Search and Context-Aware Category Search.

  - \textit{Context-Aware Fact Search}: Search for entities and facts by specifying the interesting aspects with keywords.
  - \textit{Context-Aware Category Search}: Search with keywords for abstract categories which are not linked to a special textual description.

Abstractly, the query processing algorithm follows three major steps: obtain an ordered set of seed graph patterns by performing a keyword search over the annotated text documents; propagate the scores over the ER graph to find similar graph patterns (these new pattern may be unrelated to the original keyword query); and finally formulate a semantic structured query (specifically SPARQL Select query) to filter and obtain the final result set containing graph patterns. It is clearly understood that the unstructured query or the keyword query is used for obtaining potentially relevant graph patterns. This notion stands different from that of ours as we attempt to identify individual nodes (or entities) with a keyword search over the annotated text documents. These nodes then represent the candidates for further processing. Moreover, we do not restrict to special classes of queries and rather exploit the advantages of complex semantic query formulation by extending SPARQL.
Chapter 3

Design of the Data Collection

As a core of the effort in this thesis, we coin the "Wikipedia-LOD collection" comprising XML-ified documents or so-called "WikiXML" documents, containing unstructured textual content from Wikipedia and semantic information from DBpedia and YAGO2 ontologies as a basic unit of the collection. In this chapter, we introduce the general idea behind the design and discuss in details its physical structure and properties.

3.1 Introduction

We introduce a unique data collection that attempts to close the prevailing gap between keyword-based and semantic-web based knowledge retrieval techniques. In other words, this data collection pertain properties that enables it to be a basis for Entity Retrieval (ER) tasks, Question-Answering (QA) style retrieval tasks, Ad-hoc retrieval tasks (classical keyword-based retrieval techniques) and also for Hybrid retrieval tasks (like Jeopardy task introduced in INEX 2012). To achieve this, the collection contains two types of data – highly structured RDF data and unstructured (or semi-structured) textual data. To create such a data collection we identify Wikipedia, DBpedia ontology and YAGO2 ontology as appropriate sources. It is immediately apparent that the rich textual content of Wikipedia articles form an extensive source for the unstructured part of the collection; and the RDF facts contained in DBpedia and YAGO2 ontologies, which are essentially extracted from Wikipedia articles, form an extensive source for the structured part of the collection. Moreover, we design the collection to be entity-centric, i.e., each document in the collection combines the multi-form information corresponding to a Wikipedia entity, from the sources within XML tags. In each document, we also preserve the links to the
entities in the associated LOD sources. Due to this we choose to call this collection as Wikipedia-LOD.

If we crudely examine the exhaustiveness of these selected sources, we observe that the English version of the DBpedia [Biz09b] currently describes 2.35 million things classified in a consistent ontology, including 764,000 persons, 573,000 places (including 307,000 populated places), 333,000 creative works (including 112,000 music albums, 72,000 films and 18,000 video games), 192,000 organizations (including 45,000 companies and 42,000 educational institutions), 202,000 species and 5,500 diseases. YAGO2 [Hof11] comprises of more than 9.8 million entities and 76 million facts about these entities. Thus these ontologies are decently extensive and have fairly broad coverage and can easily be considered as good sources for semantic information of Wikipedia entities. We assume that the information contained in these ontologies are fairly accurate and ignore occurrence of any noise data (if any), or wrongly extracted semantic data from unstructured Wikipedia articles.

3.2 An XML-ified Representation of Data

After having established the data sources, the challenge remains to compose the data into a standard format. We keep in mind the following aspects:

- **Consistency in merged data**, i.e., a document referring to one Wikipedia entity (or so-called resource) should contain unstructured text from the Wikipedia article describing that entity and the RDF facts from DBpedia and YAGO2 ontologies corresponding to that entity.

- **Extensible representation of each document** that logically simplifies the extension of the collection to new LOD sources.

- **Simplified structure** that makes parsing and information extraction easy.

Thus to meet the above mentioned desiderata, our data collection comprises of XML-ified documents, that employ a fusion of Wikipedia articles along with RDF facts corresponding to every Wikipedia entity. We refer to a document as "WikiXML Document". The structured part of a WikiXML document corresponding to an entity, contains semantic information of the entity occurring as both Subject (first argument) and Object (second argument) of a triple (basic unit of RDF data). The core data collection is based on the

---

1http://dbpedia.org/About
well-accepted WikiMedia format\(^2\) with substantial modifications. These modifications include replacement of the Wiki-markup with valid XML tags and CDATA sections (for infoboxes and templates). In addition, all internal Wikipedia links have been enriched with links to both their corresponding DBpedia and YAGO2 entities.

### 3.3 Insight into a Document of the Collection

As mentioned earlier, the Wikipedia-LOD collection comprises of XML-ified documents called WikiXML documents comprehending information about a Wikipedia entity. Now, let us take a closer look at a sample document in the collection. Figure 3.1 depicts the most important structural information of a WikiXML document corresponding entity:

\(^2\)http://dumps.wikimedia.org/enwiki/20111201/
Albert Einstein. As illustrated in the figure, each WikiXML document can be divided into the following sections based on the XML tags.

1. **Document Title**: The `<article title>` tag holds the title of the Wikipedia entity.

2. **Meta Data**: This section describes the meta information of the document like title, author, etc. and provides a unique Wikipedia ID. The information in this section can be used to create a map from DBpedia and YAGO2 facts to the Wikipedia articles. Thus this section provides unambiguous matching between Wikipedia resources and DBpedia or YAGO2 entities.

3. **LOD Links**: This section of the document comprises of the links or URIs to entity’s page of the associated data sources. By default, the intra-Wiki links which point to the other Wikipedia entities are extended to include the links to both YAGO2 and DBpedia resources. As shown in Figure 3.1, each link has three sub-links: Wiki-link, YAGO-link, and DBpedia-link. Wiki-link is local hyperlink to the entity, while YAGO and DBpedia links point to their respective sources.

4. **Wikitext**: This section of the document comprises of the rich textual data obtained from the Wikipedia article. This information is contained in well-formed XML syntax. As mentioned earlier, we translate the Wikipedia contents on the basis of MediaWiki format and additionally add XML tags for all infobox attributes with their values to replace all Wiki markup by proper XML tags.

5. **Properties**: This section comprises of all facts regarding the article entity in DBpedia and in YAGO2 ontologies. The facts contained in the individual ontologies can be distinguished by the following named tags

   (a) `<dbpediaproperties>...<dbpediaproperties>` – holding the facts about the article entity contained in the DBpedia ontology and,

   (b) `<yagoproperties>...<yagoproperties>` – holding the facts about the article entity contained in the YAGO2 ontology.

### 3.4 Collection Statistics

For converting the raw Wikipedia articles into our XML format, we used a parser derived from the wiki2xml\(^3\) parser provided by MediaWiki\(^4\). The parser generates an XML file from the raw Wikipedia article (originally in Wiki markup) by transforming infobox

\(^3\)http://www.mediawiki.org/wiki/Extension:Wiki2xml

\(^4\)http://www.mediawiki.org/wiki/MediaWiki
Table 3.1: Wikipedia-LOD (v1.1) collection statistics

<table>
<thead>
<tr>
<th>Property</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML Documents</td>
<td>3,164,041</td>
</tr>
<tr>
<td>XML Elements</td>
<td>1,173,255,397</td>
</tr>
<tr>
<td>Wikipedia Category Articles</td>
<td>266,134</td>
</tr>
<tr>
<td>Wikipedia Entity Articles</td>
<td>2,053,050</td>
</tr>
<tr>
<td>Wikipedia Entity Articles with Infoboxes</td>
<td>907,304</td>
</tr>
<tr>
<td>Other Wikipedia Articles</td>
<td>844,857</td>
</tr>
<tr>
<td>Resolved DBpedia Links</td>
<td>36,941,795</td>
</tr>
<tr>
<td>Resolved YAGO2 Links</td>
<td>32,941,667</td>
</tr>
<tr>
<td>Intra-Wiki Links</td>
<td>22,235,753</td>
</tr>
<tr>
<td>External Web Links</td>
<td>7,214,827</td>
</tr>
<tr>
<td>Imported DBpedia Properties</td>
<td>168,374,863</td>
</tr>
<tr>
<td>Imported YAGO2 Properties</td>
<td>23,634,511</td>
</tr>
</tbody>
</table>

information to a proper XML representation, comprehending links with DBpedia and YAGO2 entities, and finally annotating each article with a list of RDF properties from the DBpedia and YAGO2 knowledge sources.

The Wikipedia-LOD collection currently contains 3.1 Million XML documents in 3 compressed tar.gz files counting to the size of 61 GB in uncompressed form. Table 3.1 provides more detailed numbers about different properties of the collection.
Chapter 4

Design of the Query Benchmark

In Chapter 3, we designed the Wikipedia-LOD collection unifying structured RDF data and unstructured textual data corresponding to Wikipedia entities. To retrieve information from such a unified collection, we introduce a new query format that represents a coalition of semantic and keywords query. We refer to this new query format as SPARQL-Fulltext (SPARQL-FT). In this chapter, we discuss the design details of the new query format and also introduce a new query benchmark created out of these queries.

4.1 Introduction

A semantic query over RDF data graph can be considered as a basic graph pattern formulated by logical joins of triples. Semantic query processing is the matching of graph query pattern over the RDF data graph. A correct match is the mapping of the variables occurring as Subject (S) or Object (O) to the vertices of the graph; and the variables occurring as Predicate (P) to the edges. However, in order to retrieve appropriate results, we have to carefully formulate the semantic queries. A 'casual formulation' may lead to either a very small (or empty) result set or a large result set [Sha10]. Let us take a sample RDF data, as shown in Figure 4.1, and analyse specific requirements for formulating semantic queries.

1. We observe that there is often a diversity of property names even though they are repetitive across the data set. This means an end user has to be acquainted with the semantics behind the properties specific to an ontology, in order to formulate queries.
2. A semantic query processing is essentially boolean matching of a query, over the RDF data graph. Thus a query would return empty result set if the formulated query is too specific or overwhelmingly large result set if the query is too generic.

On contrary, classical keyword queries are schema-less and provide enough flexibility. A classical keyword-based search engine return results based on fulltext search over a document corpus, containing unstructured textual information, with a notion of relevance to the query. Modelling this relevance is a well studied problem and there exist state-of-art scoring functions (like Okapi BM25 [Cla05]) that perform very well in specific settings. Even with these relevance functions performing well for simple keyword queries, there are still traditional challenges like interpreting the exact user intent from a set of keywords, or "concepts", disambiguation of the keyword titles, context finding, etc. Also keyword-based retrieval techniques are unable to return results by combining information located across two or more documents in the corpus. As a result, keyword-based search engines return a ranked list of documents (or paragraphs) best matched to the given query but the user still has to look for the desired information in the document.

Thus in this chapter, we propose a new format of a query that aims to solve the above identified limitations. This format is essentially an extension of the W3C standard SPARQL with a fulltext condition operator called FTContains. We call this new kind of queries as SPARQL-FT queries.
4.2 W3C SPARQL in a Nutshell

SPARQL (SPARQL 1.1), a W3C standard, can be used to formulate simple graph pattern matching queries to more complex queries along with their conjunctions and disjunctions over diverse data sources. It supports both ASK queries (returning boolean results) and CONSTRUCT queries (returning graph patterns). In addition to the standard functionalities like SPARQL select, etc. SPARQL also supports a number of features like sub-queries, value assignment, path expressions, and aggregations such as COUNT \(^1\).

For example, a SPARQL query to retrieve information of a music album could be formulated as follows:

```
SELECT ?subject WHERE
{
?subject <artist> ?artist.
}
```

The query is composed of two triple patterns each having one SPO component. The first triple contains two variables represented with prefix "?". The variable ?subject represents the S and the variable ?artist represents the O. The variables become a place holder for entities matching the specified graph pattern in the RDF data graph. The dots at the end of each triple denote logical conjunctions and the same variable names in the two triples (?subject in this example) specify a join condition.

We see that the query is highly structured and since there exists a wide diversity in the property names, this query could be formulated in many ways. It is almost immediately apparent that it would be difficult for an end user to retrieve specific information from a RDF data graph by formulating such queries. For example, a user cannot retrieve a music album containing "Endless" in its title just by formulating joins.

One relevant feature in SPARQL specific to this context is the FILTER constraint with the regex() operator (specified in SPARQL 1.1). This feature adds a constraint over the whole group in which the FILTER keyword appears. Let us formulate a sample SPARQL query with FILTER regex constraint to retrieve information for the above information requirement.

```
SELECT ?subject WHERE
{
?subject <artist> ?artist.
FILTER regex(STR(?title), "Endless")
}
```

\(^1\)http://www.w3.org/TR/2012/PR-sparql11-overview-20121108/#sparql11-query
The above SPARQL query retrieves all the entities as S having a semantic relationship named <title> to another entity (or entities). Then string-matching is performed with the specified expression 'Endless' in the FILTER condition on the URIs of the Objects. For this query, Blue_Blood is returned the result. It is to note that this functionality in some sense allows a user to specify a keyword (or a simple regular expression), it is restricted to pattern matching on the URIs of the entities and cannot be extended to the annotated text.

4.3 Extending SPARQL Queries to SPARQL-FT Queries

As already mentioned, the Wikipedia-LOD collection is composed of WikiXML documents corresponding to Wikipedia entities, comprising RDF data from DBpedia and YAGO, and textual data from Wikipedia articles. Information can be retrieved from the RDF data alone, by formulating SPARQL queries, however, retrieving information from the unified data source is the challenge at hand. So, one could think to extend SPARQL queries to this annotated unstructured information. Such an extension could be used to formulate stronger queries. For example, it is fairly difficult to retrieve "a music album that was an all time best seller" from the given sample data set in Figure 4.1 using the existing features of SPARQL. A primary reason is insufficiency of structured data. So a rational approach would be to perform fulltext search on the unstructured textual information annotated to entities, that are retrieved as music albums by a structured query, with "music album that was an all time best seller" as keywords or key-concepts.

We extend SPARQL with a special FTContains operator. FTContains operator takes two arguments:

1. the first argument specifies the bounded entity (represented as a variable) and
2. the second argument specifies a set of keywords.

The main goal is to perform a fulltext search on the annotated text to an entity specified by the first argument, with the set of keywords specified by the second argument. The following example illustrates a new SPARQL-FT query to retrieve information for "a music album that was an all time best seller".
SELECT ?subject WHERE 
{
 ?subject <type> <Album>.
FILTER FTContains ( ?subject, "music album all time best seller" ).
}

In the above example, FTContains binds the ?subject variable which occurs as S in the triple pattern to a set of keywords as "music album all time best seller" (after excluding the stop words). This operator implies a fulltext search on the annotated unstructured content of the matching entities. The FTContains operator is close in spirit to the operator specified in the XQuery fulltext standard \[\text{Ame06a}\], except we deal with RDF data instead of XML trees.

4.4 American TV show - Jeopardy!

In the previous section, we introduced the SPARQL-FT queries. For creating the query benchmark, we select Jeopardy style natural questions (or Jeopardy clues) and express them with SPARQL-FT queries. In this section, we justify the selection of Jeopardy-style NL questions for creating the benchmark by showing that solving a Jeopardy clue is essentially performing entity-retrieval.

4.4.1 The Game

This subsection, briefly describes the popular American TV show called Jeopardy!\(^2\).

The first round of the game is called Jeopardy! and starts with the host revealing six categories of questions on a screen. Each category has 5 questions worth $100 to 500\$, arranged in ascending order, in a category column on the screen as shown in Figure 4.2. In the beginning, only the categories are displayed along with their worth. The previous episode winner starts by picking a category and an amount. For example, he could start with the category – "Religion" and amount – $300 and the corresponding question would be revealed. Also each player gets a limited time to answer a question. The difficulty of the question is proportional to the worth of the question. The round ends if all the questions on the screen are answered or a timer runs out.

Some questions are hidden Daily Double questions. If a player luckily selects a Daily Double then he can bet an amount from his already scored amount which gets doubled

\(^2\)A reader acquainted with the game show may skip to the last paragraph of the subsection
and added if answered correctly or is negated if answered wrongly. In the first round there is one hidden Daily Double amongst the 30 questions.

The second round is called the Double Jeopardy round. Rules of this round is similar to the first round except the worth of the questions are doubled, i.e., it ranges from $200 to $1000. In addition, there are two hidden Daily Double questions in this round. These questions can change a player’s position from losing to winning or vise-versa if not played strategically correct.

The third round is called the Final Jeopardy and is probably the most interesting round of the game. This round can prove to be a game changer for any player. In this round there is only one question that is asked to all the players and the players have to write the answer on a card. All the players are given 30 seconds after which all their answers are revealed. Along with the answer each player has to write an amount that is a bet for the question. If one gets the answer right then twice the betted amount is added to their total otherwise the amount is negated from the total. Of course if a player having 0 or negative balance before the round cannot participate in the round.

This game is interesting to us because it poses peculiar challenges in context of IR. IBM’s Watson was the first system [Fer10] focused in answering the Jeopardy! clues. It used a
combination of more than 100 models that worked on structured and unstructured data separately. In this thesis we attempt to investigate with the proposed data collection, the improvement of retrieval techniques on these Jeopardy! clues translated into the proposed SPARQL-FT query format.

4.4.2 Jeopardy Clues

The main essence of the game show is its unique question-answering format. A player, instead of being asked a direct question like in a traditional quiz show, is given an answer or a clue. He has to formulate the right question in order to win the money. To understand better, let us take the previous example from the query benchmark.

Niagara Falls has its source of origin from this lake.

In order to win money, a player has to correctly identify the entity pointed by the clue and formulate a question as "What is lake Erie?". He will not win if the question is wrongly formulated even though he correctly identifies the implied entity.

With respect to IR, modelling a system, that automatically interprets the exact query intention, identifies the implied entity and then formulates an appropriate 'question' as an answer is extremely complex and still remains an open research challenge [Van11]. However, some attempts [YBE+12, YBE+] have been made to automatically translate NL questions into structured SPARQL queries. Therefore, in this thesis we manually identify the exact query intention and translate a Jeopardy clue into SPARQL-FT query. Since this retrieval task works in QA-setting, we focus on the correct entity retrieval, and the final question formulation is left as a future work.

4.5 Manual Translation Framework - NL Questions to SPARQL-FT Queries

So far, we have motivated the need to create coalition queries, described as the SPARQL-FT queries. We pointed out that a set of 90 SPARQL-FT queries were translated from Jeopardy-style natural language questions and provided as a benchmark. In this section, we discuss the characteristics of the SPARQL-FT queries and the basic strategies we follow to manually translate Jeopardy-style NL questions into SPARQL-FT queries. However, as mentioned above, automatic translation of these clues or natural language queries into the proposed SPARQL-FT query format still remains an open research challenge.
4.5.1 Characteristics of the Queries

The SPARQL-FT queries have the following characteristics:

1. **Portability**: The SPARQL-FT queries are syntactically similar to the SPARQL queries with an additional FTContains operator. This operator binds an entity occurring as *Subject* or *Object* in a triple to a set of keywords. This explicit representation makes it fairly easy to separate the additional fulltext conditions by any standard text (query) parser. The remnant (essentially a SPARQL query) can be processed in any standard SPARQL query processors like Apache Jena, RDF-3x, ORACLE over the structured part of the proposed data collection. On the other hand, the separated set of keywords from the fulltext condition can be processed with any standard keyword-based search engine. Thus, this query format remains portable and does not demand eccentric settings.

2. **Scalability**: To analyse the scalability, let us individually consider the structured part (SPARQL) and the unstructured part (fulltext conditions) of the query.

   (a) **Structured part**: To formulate the structured part of the query, we use extensive ontologies like DBpedia [Biz09b] and YAGO2 [Hof11]. These ontologies have fairly broad coverage with more than a billion facts (forming the structured part of the collection).

   (b) **Unstructured part**: To formulate the unstructured part of the query, i.e., to constraint an entity with a fulltext condition, we describe the bounded entity by manually extracting relevant keyword titles from the given NL question. These set of keywords can be processed over Wikipedia articles (forming the unstructured part of the collection).

   The query coverage can be easily scaled by simply adding structured and unstructured information to the collection.

3. **Result Type**: The queries in the benchmark can be classified into two types, depending on their target entities.

   (a) **Queries that aim for one target entity**: These queries are translated from original Jeopardy questions taken from J-archive\(^3\), an online archive of actual Jeopardy questions asked in the show in past 20 years. An example is shown in Figure 4.3.

   In the example, the given NL question is ’Middle name of ’Naked and the Dead’ author Mailer or first name of ’Lucky Jim’ author Amis’. The translated

\(^3\)www.j-archive.com
Chapter 4 Design of the Query Benchmark

Middle name of "Naked and the Dead" author Mailer or first name of "Lucky Jim" author Amis

SELECT ?s WHERE {
  FILTER FTContains (?x, "Lucky Jim") .
}

Figure 4.3: Type 1: Query that aims for one target entity

What are famous couples of actors acting in crime movies

SELECT Distinct ?s ?o WHERE {
  FILTER FTContains (?o, "crime movie") .
}

Figure 4.4: Type 2: Query that aims for one a ranked list of target entities

SPARQL-FT query aims for only one entity as correct answer which is "Kingsley Amis".

(b) Queries that aim for a ranked list of one or more target entities: These type of queries are translated form NL questions that are hand crafted in the spirit of Jeopardy!. An example is shown in Figure 4.4.

The example shows a simple NL question, "What are famous couples of actors acting in crime movies", and its translation to a SPARQL-FT query. Such queries are easy for humans but difficult for an information retrieval system to answer.

4.5.2 Query Relaxation Strategy

Translation of a NL question into SPARQL-FT, if not carefully done, will lead to a result set that is either too specific or overwhelmingly large. This leads to the query relaxation problem. This problem puts forth the challenge to formulate the queries that would return a bigger result set without affecting the quality. Since this is done manually, it becomes important to briefly mention the strategies undertaken to relax queries. However there are some methods, like [Elb11b], that claim to handle the query relaxation problem automatically.

We follow the following strategies:

1. Replace very specific predicates with similar but more generic predicates. While trying to describe a sentence in the NL question in form of triple, a
human translator has to carefully select a predicates that lead to a big result set with the desired entities. It is difficult to manually adopt a brute-force methodology i.e., to identify all the possible formulations and then select the most appropriate one.

2. **Replace entities with fulltext conditions.** Without sufficient knowledge of the entity representations along with disambiguations, it is difficult to formulate triples from a given NL question. Also an arbitrary selection of an entity might lead to empty result set or too much noise. To tackle this problem, we replace an entity variable and bind the variable to a set of keywords correctly describing the entity. To understand this let us take an example as follows.

Consider a simple NL question: *'What is the capital of Philippines?'*. We can easily express this question as a triple and formulate a SPARQL query as shown below.

```sparql
SELECT ?Capital WHERE {
  <http://dbpedia.org/resource/Second_Philippine_Republic>  
    <http://dbpedia.org/ontology/capital>  
    ?Capital.
}
```

or

```sparql
SELECT ?Capital WHERE {
  <http://dbpedia.org/resource/Philippines>  
    <http://dbpedia.org/ontology/capital>  
    ?Capital.
}
```

Note in the both the above shown SPARQL queries, the occurring surface form "Philippines" is mapped into two different entities and in fact, it can be mapped into 10 different DBpedia entities to retrieve the desired information. Thus this query can be formulated in 10 different ways. And if there exist a join of this triple with another triple representing another constraint it becomes important to exactly identify which out of the 10 DBpedia entities to use in the query. This is an observed problem with SPARQL as pointed in Chapter 4.1. To solve this we can formulate a SPARQL-FT query as :

```sparql
SELECT ?Capital WHERE {
  FILTER FTContains(?Country,"Philippines")
}
```
In the above query formulation, the variable \texttt{?Country} is replaced by the best-matched entities by a fulltext search on the Wikipedia articles and filtered by the graph pattern specified by the triple. In the result set we get a list of entities which implies processing of a more relaxed query.

### 4.6 Summary of the Translation Procedure

Finally we can list down comprehensive steps to translate a NL question into a SPARQL-FT query. Given a NL question, we perform the following steps.

1. Identify probable entities and relationships from the NL question.
2. Identify the surface forms or keyword titles describing entities from the NL question.
3. For each probable entity, define a variable (or place-holder) in a triple and create a 
   FTContains constraint (fulltext condition). Pass the defined variable as the first
   argument and pass the keyword set as the second argument.
4. Determine DBpedia or YAGO2 properties from the identified entity relationships.
5. Formulate a triple pattern with the variables for entities (as \textit{Subject} or \textit{Object}),
   DBpedia or YAGO2 properties and fulltext constraints.
6. For a more complex NL question, formulate similar triples and design a join
   structure that should return the correct answer in the final result set.
Chapter 5

SPAR-Key : Rewriting SPARQL-FT to SQL

In this chapter we introduce a query engine, called SPAR-Key, that translates a SPARQL-FT query into a conjunctive SQL query and processes it over a relational database schema. We delineate three translators, namely SPAR-Key Identity, SPAR-Key Supremacy and SPAR-Key Ultimatum, and analyse the results from each. We show that each translator introduces new translator strategies and promises either an improvement of efficiency in query processing or improvement in the quality of results. Finally, we underline two translation algorithms, where the first one translates any SPARQL-FT query, while the second translates a specific subset of SPARQL-FT queries with special properties and return better results compared to the first one.

5.1 Introduction

Semantic query processing is a boolean matching of the graph pattern over a data graph (as described in Chapter 4). Such semantic queries represent basic graph patterns composing triple patterns and can be translated into conjunctive SQL queries. Many real-world prototypes [Bro02, Cho05, HG03, Zou11, Aba09, Kev03] prove this by proposing a translation framework of structured queries in SPARQL into conjunctive SQL queries.

In Chapter 4, we proposed a new query format which is an extension to the W3C standard SPARQL for processing structured queries over RDF data. This new query format is syntactically similar to SPARQL queries with additional FTContains operator. This additional operator, with two arguments, represents a fulltext condition constricting an
entity. The first argument takes a variable, which is a place holder for an entity occurring as a Subject or an Object in a triple and the second argument takes a set of keywords. It is not hard to realise that this extended SPARQL query can also be processed under the conjunctive query processing paradigm. Moreover, the storage model designed for the collection, following the relational perspective, permits to translate a SPARQL-FT query into an SQL query. However, for such a translation we have to take care of correct projections, preserve the join logic and at the same time do some basic optimisation so as to achieve a decent efficiency (though efficiency is not the main goal of the investigation). Our system-generated queries are essentially conjunctive queries over multiple instances of the DBpediaCore and Keywords relational tables (described later). In this chapter we shall discuss the storage model and the SPAR-Key engines, with their translation strategies, in details.

5.2 Data Storage in Relational-DBMS

Data management and storage forms a critical part in designing a query processor. Thinking of scalable data storage, the most common option is a Relational Data Management System (relational-DBMS). Though there are other approaches like file systems, graph-databases, etc., a relational-DBMS based approach opens up all the optimisation techniques presented by the database community, that aid a decent efficiency in query processing. Designing a clever database schema as our storage model suffices our need as we aim to investigate the effectiveness and not the efficiency of the query processing.

We propose the Wikipedia-LOD collection in Chapter 3 that represents a unique fusion of unstructured textual data and highly structured semantic or RDF data. We represent a basic unit of data of this entity-centric collection as WikiXML document, comprising of semantic data and textual data corresponding to Wikipedia entities separated with well formatted XML tags. Thus, due to the neat XML representation, a document (and hence the entire data collection) can easily be parsed into a structured part and an unstructured part. It is important to design a composite data storage model for these parts so as to efficiently process the new SPARQL-FT query over the data.

5.2.1 Storage Model for Structured Data

The RDF data can be commonly perceived as a collection of triples of the form Subject (S), Predicate (P) and Object (O) or sometimes also referred to as, in entity-relationship terminology, Entity, Property and Value. In the last decade, there have been many perspectives put forward by different research communities to vie RDF data. We identify
the three most important in order to find the best suited data storage model for our collection namely: a relation perspective, an entity perspective and a graph-based perspective [Luo12].

1. **The relational perspective** is brought by the database community and implies that a RDF graph is essentially a special form of relational data and can be stored in any relational-DBMS. Due to this, all the standard optimisation techniques developed for relational data can be reused on the relational representation of RDF data. Thus standard W3C SPARQL queries are translated into SQL queries and processed over the relational tables containing the data. As a result, these queries return relational tables containing data projected by the translated SQL queries. This perspective provides two methodologies to represent RDF data as relational data:

   - a *Vertical representation* where the entire data is stored in one single relational table with schema (Subject, Predicate, Object).
   - a *Horizontal representation* where every Predicate value forms a column in an a relational table.

Though the latter is seldom used, the vertical representation is adopted by some of the state-of-art RDF data stores.

2. **The entity perspective** presented by the IR community acknowledges RDF data as a graph (G) and each resource is deemed as an entity. These entities have a set of properties (which form the Predicates) with values (which form the Objects). Inverted indexes are traditionally used as primary data structure and many real-world systems have proved their scalability and effectiveness. These systems support keyword queries which are either simple (bag-of-words) or of more complex nature with constraints and conditions. As a result, these queries return a list of ranked entities with a notion of relevance to the issued query.

3. **The graph-based entity perspective** has emerged from graph databases and semi-structured databases. This perspective view of RDF data as a large graph (G) with entities as named nodes or vertexes (V) in the graph and relationships as labelled directed edges (E). This enables the theoretical graph algorithms to process basic queries like navigation (shortest-path) between two nodes etc.

### 5.2.1.1 Our Approach: Storing RDF Data as a Single Relational Table

The vertical representation under the relational perspective enables us to view it as a large collection of triples containing SPO components. These collections are sometimes known
Chapter 5 SPAR-Key : Rewriting SPARQL-FT to SQL

Figure 5.1: Graph representation of the sample RDF data.

<table>
<thead>
<tr>
<th>Column</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>N3ID</td>
<td>NUMBER</td>
</tr>
<tr>
<td>Subject</td>
<td>VARCHAR2(1024)</td>
</tr>
<tr>
<td>Predicate</td>
<td>VARCHAR2(1024)</td>
</tr>
<tr>
<td>Object</td>
<td>VARCHAR2(1024)</td>
</tr>
</tbody>
</table>

Table 5.1: Table schema the DBpediaCore table

as triple stores. This simply means that a given structured query in SPARQL has to be translated into a SQL query and issued to the relational-DBMS. Let us take an example to understand the relational representation of RDF data. Figure 5.1 shows a fragment of a RDF graph constructed over Wikipedia, and Figure 5.2 shows the corresponding relational table representation in the database.

From this example, it is evident that the entire RDF triple store can be realised as a single table over the relational schema (Subject, Predicate, Object). This route is pursued similarly by many triplet-stores like Jena [Kev03], Sesame [Bro02] and RDF-3X [Neu08]. Though one may claim to achieve better efficiency with more complex approaches, like vertical partitioning or the exploitation of property tables, but the key goal of this thesis, i.e., to investigate a unified collection, can be very well achieved by such a relatively simple approach.
As our storage back-end, we use the Oracle 11g relational-DBMS to store the RDF data we imported from the core dump of DBpedia (see http://downloads.dbpedia.org/3.7/en/). This means that a given structured query (like a SPARQL query) or the structured part of a given SPARQL-FT query, it is simply translated to a SQL query and processed over the relational schemas defined. Table 5.1 shows the schema of the table that stores the entire structured part of the collection. We call this table DBpediaCore table and from here, we refer to the table with this name.

5.2.1.2 Creation of the DBpediaCore Table

To parse the structured part of the collection constituting RDF facts, we make use of current Linked Open Data dumps for DBpedia (v3.7) and YAGO2, which are available from the following URLs:

- DBpedia v3.7 (created in July 2011):
  http://downloads.dbpedia.org/3.7/en/

- YAGO2 core and full dumps (created on 2012-01-09):
  http://www.mpi-inf.mpg.de/yago-naga/yago/

The N-Triple (.nt) format of the dumps are downloaded and are bulk-loaded into the Jena RDF engine. We make use of the Apache Jena TDB to bulk load the triples.
into the engine. Jena TDB build its own indexes over the data that can be used to efficiently process SPARQL queries over the ontologies. However, we use the JAVA interface provided by the Jena TDB to traverse over all the triples and further bulk load then into a relational table. Usage of Apache Jena TDB is not strictly necessary for building the triple store in the relational-DBMS as any standard RDF parser could do the job. Our use of Jena was solely to exploit its fast bulk loading mechanism. Table 5.2 shows some statistics of the DBpediaCore table.

### 5.2.2 Storage Model for Unstructured Data

Data, lacking a clear semantic structure, is deemed as "unstructured data". Traditionally, an unstructured data collection, or a corpus, is a collection of documents containing textual data. Such textual data corpus are considered as a collection of independent terms commonly expressed as bag-of-words. A document is defined as a basic unit of data on which retrieval systems are designed. To retrieve information from such document-centric corpus, we design systems to address an Ad-hoc retrieval task over it. The main goal is to retrieve documents containing information from the collection that would satisfy the information needs of a user. To express information needs, a user issues a query, which the system interprets as a bag-of-words, and retrieves a set of best-matched or relevant documents. A document is relevant if a user finds it as source of valuable information. Such a type of query issued by the user is called a keyword query as it contains a set of independent keywords.

Traditionally, inverted indices are used as core data structures for keyword-based retrieval systems. These term-document inverted indices efficiently identify the most relevant or find the best matched documents to a given keyword query. Essentially, inverted indices map every term (as keys) in the corpus to a ordered set of documents based on a ranking function. This ranking function models the similarity of a term to a document and generates a score, based on which the documents are ordered and presented to the user.
5.2.2.1 Our Approach : Storing textual data in a Single Relational Table

In traditional IR, a fulltext search performed with a keyword query retrieves a list of best-matched documents from the underlying corpus and optionally orders the retrieved documents on a score representing the similarity with the issued query. There are many state-of-art scoring statistical functions that attempt to model the relevance between the query and the retrieved documents. However, the most common approaches use fairly simple TF/IDF counts or Boolean retrieval to measure the content similarity of a fulltext keyword query to the documents in the corpus.

Commonly keyword-based retrieval systems at least store a map or inverted index mapping every term to the documents in the corpus. This can be viewed as relational data and can be stored in relational-DBMS. Realising this, we store the unstructured text contents of all the documents in the collection as another table in the relational-DBMS.

We define a keyword or term as a sequence of characters that occur together in some document and yield a useful semantic unit for processing. We extract keywords from the fulltext content of a Wikipedia article. In our approach, we create a relational table called **Keywords** table to store all the terms occurring in the unstructured Wikipedia fulltext collection and from here we refer to the table with this name. The schema of this table is shown in Table 5.3.

The **Entity_ID** column essentially stores the Uniform Resource Identifier (URI) of the DBpedia entities. Since in our entity-centric collection, every document corresponds to a DBpedia entity, we prefer to use the prefixes defined by DBpedia to represent these entities, for example [http://dbpedia.org/resource/entity](http://dbpedia.org/resource/entity). Every statement of the **Keywords** table represents a term mapped to DBpedia entity and a similarity score to the entity’s Wikipedia page.

5.2.2.2 Creation of the Keywords Table

We employ a regular SAX parser to parse the 3.1 Million XML articles whose general XML structure is still based on that of the original articles. That is, these articles contain a metadata header with information about the ID, authors, creation date and
Chapter 5 SPAR-Key: Rewriting SPARQL-FT to SQL

<table>
<thead>
<tr>
<th>Number Of Rows</th>
<th>715609000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blocks</td>
<td>6415636</td>
</tr>
<tr>
<td>Distinct_Rows_Entity_ID</td>
<td>135082</td>
</tr>
<tr>
<td>Distinct_Rows_Term</td>
<td>16353141</td>
</tr>
</tbody>
</table>

Table 5.4: Table Summary of the Keywords table

others, usually also an infobox with additional semi-structured information consisting of attribute-value pairs that describe the entity, and of course rich text contents consisting of unstructured information and more XML markup about the entity that is captured by such an article. Our keyword indexer uses the basic functionality of TopX 2.0 [Mar09], which includes Porter stemming, stopword removal and BM25 ranking, but stores the resulting inverted lists for keywords into the Keywords relational table instead of TopX proprietary index structures. Table 5.4 shows some statistics of the Keywords table.

5.2.3 Indexes on the Relational Tables

We note that solving a complex SPARQL query, with one ore more logical joins of triples, employs multiple self joins of DBpediaCore table and also Keywords table. This is a performance killer due to the colossal size of the tables. Relational-DBMS systems provide a standard optimisation by facilitating index creations over a relational table. This is in spirit similar to the inverted indices creation adopted by IR community, but with limited customisation.

Unclustered Index: A default index creation in Oracle is an unclustered index. In unclustered index a pointer to every triple is maintained separately. This map is essentially a B-Tree and drastically reduces look-up time on a large table. The idea is to create a fast access path to the data, in our case to each triple. It is to note that an index is an optional data structure built over the data and is physically and logically independent of the data itself. So we can create multiple indexes over the same table with different index schemas. Let us analyse some types of unclustered indexes that Oracle allows to be created:

1. Unique and Non-unique Index: A unique index guarantees the non existence of duplicate values in the key column. On the other hand, non-unique index allows the duplicate values in the key column. In our context all three columns of our DBpediaCore table has multiple entries with duplicate values. Keywords table also follows a similar story. Hence, we create the default or non-unique indexes for both the tables.
2. **Composite index or concatenated index**: This is an index with one or more columns of the table as the key of the index. This type of index can specially speed up the queries with *where* clauses. In addition, the arrangement of the columns in the table is not important but the order defined in the index definition is important. Thus we can create composite indexes with pairs SP or PO etc. as keys. This becomes really useful in our context.

3. **Visible and Invisible Indexes**: A visible index is one which is implicitly used by the optimiser to speed the query processing while an invisible index is one which is invisible to the optimiser. In our system we rely on the Oracle optimiser (described in details later) to speed up the query execution time.

In order to achieve maximum efficiency for any given query, we need to build six indexes, one for each permutation for the triple components (SPO) [Neu08]. However, we observe that we already achieve decent efficiency with three non-unique, visible and composite indexes on the three columns of **DBpediaCore** table. Also our goal is not to achieve maximum efficiency but to investigate the effectiveness of the query processing. So, we create the three indexes as shown in Table 5.5.

Similarly we create two non-unique, visible and composite indexes on the **Keywords** table. The two indexes are built with the consideration that queries are issued with conditions on the **Entity_ID** column and **Term** column while the **Score** column is used for internal purposes of ranking the result. Table 5.6 describes these two indexes built over the **Keywords** table.

### 5.2.4 Keywords Scoring: Okapi BM25

In the previous section, we discussed the storage model for unstructured data as a single relational table with schema as (**Entity_ID**, **Term**, **Score**). In this section, we present the scoring model used to generate the per term-entity scores stored in the third column
of the Keywords table. As already discussed in a fulltext search a list of documents (entities in our case) are returned, ordered with a notion of relevance to the query. This relevance is captured by ranking functions which provide a statistical measure for the same. This is a standard technique in IR and a well studied problem. There exist many state-of-art ranking functions proposed by many works that work well for a definite setting. The selection of a ranking function depends on the application and it would be difficult to claim a gold standard ranking function that is optimal given scenario or application. For our data, we select a variant of the Okapi BM25 [Cla05] scoring function and which works well for the fulltext searches involved in our query processing (discussed in details in the next chapter).

BM25 is a combinational statistical function with many functions that occur as components with tunable parameters. Each component independently captures a statistic and a final score is generated by their combination. We note that a BM25 ranking function assumes the independence of terms occurring in a document and hence does not take into account the distribution or any other special relationship of terms (like terms in phrases). The exact BM25 variant we used for ranking an entity \( e \) by a string of keywords \( S \) in an \texttt{FTContains} operator is given by the following formula:

\[
score(e, \text{FTContains}(e,S)) = \sum_{t_i \in S} \frac{(k_1 + 1) \cdot tf(e, t_i)}{K + tf(e, t_i)} \cdot \log \left( \frac{N - df(t_i) + 0.5}{df(t_i) + 0.5} \right)
\]

with
\[
K = k_1 \left( (1 - b) + b \frac{\text{len}(e)}{\text{avg}\{\text{len}(e') \mid e' \text{ in collection}\}} \right)
\]

where,

1) \( N \) is the number of XML articles in Wikipedia LOD collection.

2) \( tf(e, t) \) is the term frequency of term \( t \) in the Wikipedia LOD article associated with entity \( e \).

3) \( df(t) \) is the document frequency of term \( t \) in the Wikipedia LOD collection.

4) \( \text{len}(e) \) is the length (sum of \( tf \) values) of the Wikipedia LOD article associated with entity \( e \).

We used the values of \( k_1 = 2.0 \) and \( b = 0.75 \) as the BM25-specific tuning parameters (see also [Cla05] for tuning BM25 on earlier INEX settings).
5.2.5 Translating the Keyword-Scores to SPO Triples

Recently there has been a lot of work done to identify appropriate scoring in entity retrieval. Many techniques use language models [YCN10, Pe06, P08] while other approaches try to adopt more complex measures from IR. Ranking for structured queries have been intensively investigated for XML [Ame06b], and, to a small extent, also for restricted forms of SQL queries [Cha06]. While some of these approaches carry over to large RDF collections and expressive SPARQL queries as discussed earlier, we make a significant simplifying assumption: since every DBpedia or YAGO2 entity is associated (via an explicit link) with the Wikipedia article (in XML format) that describes this entity, the entities can directly be referred to and ranked by the keywords that are imposed over the corresponding Wikipedia article.

We thus believe that it is a good idea to translate the scores of the RDF entities from a keyword-based search to the fulltext part of the Wikipedia LOD collection. Thus, we perform a keyword search on this fulltext content by using the fulltext condition specified by the FTCContains operator. From this search, we get a ranked entity list containing the candidates answers to the given query. This can be best explained by an example as shown in Figure 5.3.

Figure 5.3(a) shows an example query containing a simple SPARQL query with one triplet and a fulltext condition that is bound to the Subject of the SPARQL query. Figure 5.3(b) shows fragments of top-100 results obtained by fulltext search on the unstructured part of the collection with keywords “Lucky Jim”. As illustrated, we already have the relevant
candidates for the answer from the keyword search due to the satisfactory performance of our BM25 scoring scheme applied to score the keywords. The scored entities in the candidate list is again checked for the graph pattern of the SPARQL query in Figure 5.3(a) and the final top-k entities are retrieved from the entities which qualify for the graph pattern. Figure 5.3(c) shows the retrieved results after searching for the pattern in the graph.

5.2.5.1 Special Case

As we describe a simple method to score triple entities based on the fulltext constraints, there may arise a special case where in a SPARQL-FT query, a triple pattern entities do not have a fulltext constraint. In such a case all the entities satisfy the structure defined by the triple become equi-likely to either to answer to the NL question from which the SPARQL-FT query is formulated, or for further processing. Thus to handle this we give a default score to all such entities. We choose to give a score of 1 to such entities. Let us take an example to understand this.

The example in Table 5.7 shows a Jeopardy-style NL question and translated SPARQL-Ft query. The query contains two triple patterns and one fulltext condition binding ?x, occurring as the Subject of the first triple ( ?x <http://dbpedia.org/property/data> ?s .). Thus the entities in the second triple receive a default score of 1 implying that all the entities (as Subject or Object) that follow the structure of the triple, become a candidate entities for further processing.

5.3 SPAR-Key Identity

SPAR-Key Identity is a basic translator with a straight-forward translation framework of SPARQL-FT queries into conjunctive SQL queries. The main goal is to check the compatibility and feasibility of SPARQL-FT queries with SQL queries.
A manually translated SPARQL-FT query:

```
SELECT ?sub WHERE {
    ?x ?r ?s .
    ?sub ?o ?s .
    ?sub rdf:type ⟨http://dbpedia.org/ontology/Place⟩ .
    FILTER FTContains (?x, "Canarian").
    FILTER FTContains (?sub, "place islands country") .
}
```

Table 5.8: A given SPARQL-fulltext query of the INEX 2012 Jeopardy task

<table>
<thead>
<tr>
<th>SELECT DISTINCT T1.SUBJECT AS SUB FROM</th>
</tr>
</thead>
<tbody>
<tr>
<td>DBPEDIACORE T2, DBPEDIACORE T1, DBPEDIACORE T3,</td>
</tr>
<tr>
<td>KEYWORDS K40, KEYWORDS K41, KEYWORDS K42, KEYWORDS K50</td>
</tr>
<tr>
<td>WHERE</td>
</tr>
<tr>
<td>T2.OBJECT = T1.OBJECT</td>
</tr>
<tr>
<td>AND T1.SUBJECT = T3.SUBJECT</td>
</tr>
<tr>
<td>AND T3.PREDICATE = '<a href="http://www.w3.org/1999/02/22-rdf-syntax-ns#type">http://www.w3.org/1999/02/22-rdf-syntax-ns#type</a>'</td>
</tr>
<tr>
<td>AND T3.OBJECT = '<a href="http://dbpedia.org/ontology/Place">http://dbpedia.org/ontology/Place</a>'</td>
</tr>
<tr>
<td>AND K40.TERM = 'place'</td>
</tr>
<tr>
<td>AND k41.TERM = 'island'</td>
</tr>
<tr>
<td>AND K42.TERM = 'country'</td>
</tr>
<tr>
<td>AND K50.TERM = 'canarian'</td>
</tr>
<tr>
<td>AND T1.SUBJECT = K40.ENTITY_ID</td>
</tr>
<tr>
<td>AND T1.SUBJECT = K41.ENTITY_ID</td>
</tr>
<tr>
<td>AND T1.SUBJECT = K42.ENTITY_ID</td>
</tr>
<tr>
<td>AND T2.SUBJECT = K50.ENTITY_ID</td>
</tr>
</tbody>
</table>

Figure 5.4: A basic conjunctive query from a SPARQL-FT query

5.3.1 Example 1

To capture the idea behind translating a given SPARQL-FT query into a conjunctive SQL query, let us take an example query $Q$ from the benchmark as shown in Table 5.8. $Q$ contains three triple patterns $T_1$, $T_2$ and $T_3$, and two fulltext conditions $K_1$ and $K_2$. Each $K_i$ contains a variable occurring in a triple pattern and is bound to an entity by an FTContains operator.

To begin translation, every attached string is tokenized and stemmed into the format of terms stored in the Term column of the Keywords table. For every generated token $k_j$ where $j \geq 0$ from each $K_i$, an instance of the Keywords table is taken, where the Term column of the instance is restricted to $k_j$. Similarly for every triple pattern $T_m$, we take an instance of the DBpediaCore table $t_i$, where $i \geq 0$. In the example, instance $t_1$ represents triple $\text{?sub ?o ?s}$, instance $t_2$ represents the triple $\text{?x ?r ?s}$, and instance $t_3$
represents the triple \(?sub\) \text{rdf:type} (http://dbpedia.org/ontology/Place). These instances \(t_i\) are further restricted on their \text{Subject} or \text{Object} by the \text{Entity_ID} of the \(k_j\) instance of the \textbf{Keywords} table as specified by the fulltext condition. Finally, the instances \(t_i\) are restricted by each other on \text{Subject}, \text{Predicate} or \text{Object} as per the logic of the joins in the SPARQL query. The translation of \(Q\) into a conjunctive SQL query, as described above, is shown in Table 5.4.

\subsection{5.3.2 Bottlenecks and Solutions}

We observe that SPAR-Key Identity represents a naive translation framework. Anyone acquainted with query processing in relational-DBMS, would argue that such queries generated by SPAR-Key Identity are simplest possible queries and would be efficiently executed. On the contrary, the translated SQL queries are often inefficient and mostly return a null result. A thorough analysis leads to the identification of the following bottlenecks.

- \textbf{The database optimizer frequently fails to find the best query plan.}
  Relying on the Oracle was the initial plan to find the best query execution plan. However, even a state-of-art query optimiser, like that of Oracle, fails due to the following reasons:

  - \textit{Insufficient Column and Table statistics of DBpediaCore table and Keywords table}: Each column of \textit{DBpediaCore} table and \textit{Keywords} table store data of \textit{varchar} type. Moreover, each column represents though diverse, yet largely reoccurring data. Broadly, \textit{table statistics} gathered for a relational table by the optimiser, includes the \textit{total number of rows} in a table, and \textit{average row length} of a row. \textit{Column statistics} include information on the \textit{number of distinct values in a column} (NDV) as well as the \textit{minimum and maximum value found in the column}. Figure 5.5 illustrates an overview of the optimizer statistics.

  - \textit{Insufficient additional column statistics} : Apart from the \textit{general statistics} (Column and Table statistics), the optimiser also builds additional column statistics, like \textit{frequency histograms}, which are used for cardinality estimation and hence leads to formulation of the best query plan. The colossal size of the tables and the highly diverse, yet recurring, data stored in the columns make such statistic generation even more difficult. One specific case could be where the optimiser fails to estimate the number of documents for every term stored in the \textit{Keywords} table. Moreover it is difficult for the optimiser to estimate the number of entities per term, for every term.
Oracle 11g handles the statistics gathering automatically, however, it does provide functionality to explicitly initiate this process. We highlight that we observe insufficient statistics collected by the optimiser even after explicitly initiating the process. Due to the insufficient column and table statistics, the optimiser mostly selects inefficient join orders for joining the individual Keywords table and DBpediaCore table instances. Without losing generality, we can argue that the best order should be in the increasing cardinality of join candidates or in other words, lists with the lowest cardinality should be joined first.

Solution: We need to perform some sort of selectivity estimation by building our own statistics and force the optimiser to formulate the right joining order, i.e., joining the tables (or intermediate result sets) with lowest cardinality first. We note that we can force the optimiser to join in a specified order by adding appropriate Oracle Optimiser Hints. In addition, materialising temporary tables forms an additional hint for the optimiser (described later in the chapter).

- The entire query returns an empty result set if a single keyword condition fails. This is a clearly observed problem in large conjunctive keyword queries.
In the predicament of occurrence of a keyword which is absent from the Keywords table, a fulltext constraint, containing that keyword, returns an empty result set for such projected entities in a sub-query. In such a case, the overall result of the conjunctive query would be an empty result set. Thus we need relaxation of queries to avoid such situation where the entire query returns NULL even with only one (or few) condition failing.

Solution: We see, using some special join strategies like INNER joins and OUTER joins we can avoid such a case.

5.4 SPAR-Key Supremacy

In the Section 5.3, we showed a technique to translate a SPARQL-FT query into a conjunctive SQL query. However, we identify bottlenecks in processing large conjunctive SQL queries which are a direct logical translation of SPARQL-FT queries.

In this section, we introduce a new translator, coined SPAR-Key Supremacy, that overcome the bottlenecks of the SPAR-Key Identity. To achieve this, the translator 1) uses SQL Joins over simple AND conditions, 2) materialises temporary tables and sub queries to represent intermediate results, 3) uses a simple selectivity estimation technique to formulate join order of tables, and 4) includes Oracle optimiser hints to force the optimiser to follow the formulated join order.

5.4.1 Exploiting SQL Joins

Almost all modern relational-DBMS provide joining of two tables as a basic functionality. We observe that by formulating some of the clever joining strategies provided by Oracle, we can avoid the NULL result problem (Section 5.3.2). Our main goal is to translate a SPARQL-FT query into a SQL query with INNER join and OUTER join conditions.

A join query contains at least one join condition, either in the FROM (for OUTER joins) or in the WHERE clause. Conceptually, a join condition compares two columns, each from the tables specified by the join condition and combines pairs of rows, one row from each table, for which the join condition evaluates to true.

Joining of three tables is performed by joining first two tables as per the join condition and then the result to the third table based on the join conditions containing columns of the previously joined tables and the third table. A similar strategy is followed if joining of more than three tables is performed i.e., Oracle joins intermediate result set (of a join) to the new table to obtain another set of intermediate results and this follows until all
the tables in the join condition are joined. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and available statistics for the tables.

### 5.4.1.1 FULL OUTER JOIN

In a FULL OUTER JOIN, each record in the two joined tables does not require a matching record. The joined table retains each record even if there exists no other matching record. Where records in the FULL OUTER JOIN’ed tables do not match, for every column of the table, that lacks a matching row, the result set will have NULL values. For non matching records, a single row will be produced in the result set (containing fields populated from both tables). This join is also used to solve the second bottleneck described in Section 5.3.2. All instances of the **Keywords** table representing a fulltext condition $K_i$ can undergo a FULL OUTER JOIN on the **Entity_ID** attribute.

### 5.4.1.2 INNER JOIN

An INNER JOIN is the most common join operation and is also regarded as the default join type. This join combines the column values of two tables ($X$ and $Y$) based upon the join predicate. To find all pairs of rows which satisfy the join predicate, the query compares each row of $X$ with each row of $Y$. The column values for each matched pair of rows of $X$ and $Y$ are combined into a result row whenever the join predicate is satisfied. The result of the join can be calculated by taking the Cartesian product (or cross join) of all records in the tables (combining every record in table $X$ with every record in table $Y$). It returns all records which satisfy the join predicate. This join returns the same result as the “equality” join but gives more flexibility to the optimizer.

### 5.4.2 Materializing Temporary Tables

One big conjunctive query forces the Oracle optimizer to choose from a very large number of possible query execution plans, and it often chooses an inefficient plan (Section 5.3.2). Thus, to prevent the optimizer from taking such inappropriate decisions, we materialize temporary tables by separately joining the **Keywords** table instances and the **DBpediaCore** table instances. This strategy acts as a strong hint for the optimizer. The optimiser selects better query plans for the smaller intermediate queries and store their results into temporary tables which are later joined together to retrieve the final result.
5.4.3 Evaluating the Join Order and Forcing Orders via Optimizer Hints

There are some simple techniques by which we can determine the join order of the tables. One such technique is to maintain a feature index or inverse document frequency (IDF) index containing the most frequent terms that occur in the collection. This index preserves a very simple schema, i.e., Features(Term, IDF). The first column represents a term and the second column represents its IDF. A frequent term will have lower IDF and a select query on the Keywords table, with the term as constraint, will return a larger result set. At the same time, if a term is absent in the feature index, it can be assumed to be infrequent. Every instance of the Keywords table can now be joined in increasing order of the IDF values of their respective terms, thus ensuring the smaller tables to be joined first. This order of joining can be enforced on the Oracle optimizer by adding optimizer hints to the queries.

A hint is a code snippet that is embedded into a SQL statement to suggest Oracle how the statement should be executed. There are many hints provided to assist the optimizer. In our case, we identify the Ordered hint to force the optimizer to join tables in the specified order written in the FROM clause of the query. So our translator algorithm writes the Keywords table instances in appropriate order in the FROM clause of the translated SQL query.

A similar analysis is done for joining the temporary tables representing each triples. To estimate the cardinality of the results from individual triples in a SPARQL query, we follow a assumption that a pattern with more constraints on the components (SPO) will result in smaller result set. However, one may argue that such an assumption may not give a close estimate but is enough in our case to decide the order of joining the triples. A more accurate method is presented by RDF-3X [Neu08].

These constraints include the literals specified in the triple and also the fulltext contains. This forms a simple method to determining the join order of the triples. We use the Ordered hint to force the joining the tables representing triples in the determined order while preserving the logics of the join condition in the original query.

5.4.4 Example 2

In this sub-section, we try to capture the above analysed techniques to translate a SPARQL-FT query by taking an example. Let us take the SPARQL-FT query described in Section 5.3.1. We translate this query into the following SQL queries by exploiting SQL joins, temporary tables, sub queries and Oracle optimiser hints.
Figure 5.6 shows the SQL query for the creation of the Keys\text{0} tables. These relational tables hold intermediate results which are entities satisfying a fulltext condition $K_1$. For example, in Figure 5.6 Keys\text{0} represents the results from a fulltext condition ’country place island’ bounded to \( ?\text{sub} \). To create the Keys\text{0} table, a FULL OUTER JOIN is performed on the result sets of a fulltext search with individual terms. These result sets are represented with aliases $K_i$ in Figure 5.6. The final score of the entities in the Keys\text{i} tables are aggregate of their scores from individual terms. Thus entities retrieved by more terms have a higher score compared to entities that are retrieved by fewer terms. For example, in this query, ’Canary Islands’ will have a higher score compared to ’India’ as it will be retrieved by all the terms (’country’, ’place’ and ’island’) while the later will be retrieved by only two terms (’country’ and ’place’).

Figure 5.7 shows the SQL query for creation of the Tab\text{o} tables. Each Tab\text{o} table stores intermediate results corresponding to each triple pattern $T_o$. For example, in Figure 5.7, Tab\text{3} table represents the intermediate results for the triple " ?sub rdfs:type <http://dbpedia.org/ontology/Place>". Note that it performs an INNER JOIN on the Keys tables and the triples in the DBpediaCore table that follow defined structure. This filters out the entities that are retrieved by a fulltext search but do not follow the defined structure.
Chapter 5 SPAR-Key : Rewriting SPARQL-FT to SQL

Figure 5.7: Queries to create the TAB temporary tables

CREATE TABLE TAB3 AS
SELECT SUBJECT, PREDICATE, OBJECT, ( NVL(KEYS0.FINAL_SCORE,0) ) AS FINAL_SCORE FROM

(SELECT * FROM DBPEDIACORE3 T1
WHERE T1.PREDICATE='http://www.w3.org/1999/02/22-rdf-syntax-ns#type'
AND T1.OBJECT='http://dbpedia.org/ontology/Place'temp
INNER JOIN
KEYS0
ON TEMP.SUBJECT=KEYS0.ENTITY_ID

CREATE TABLE TAB2 AS
SELECT SUBJECT, PREDICATE, OBJECT, ( NVL(KEYS0.FINAL_SCORE,0) ) AS FINAL_SCORE FROM

(SELECT * FROM DBPEDIACORE3 T3 )TEMP
INNER JOIN
KEYS0
ON TEMP.SUBJECT=KEYS0.ENTITY_ID

CREATE TABLE TAB1 AS
SELECT SUBJECT, PREDICATE, OBJECT, ( NVL(KEYS1.FINAL_SCORE,0) ) AS FINAL_SCORE FROM

(SELECT * FROM DBPEDIACORE3 T2 )TEMP
INNER JOIN
KEYS1
ON TEMP.SUBJECT=KEYS1.ENTITY_ID

Figure 5.8: The final SELECT query to obtain the answer

SELECT /*+ORDERED*/ SUB, MAX(AGGR_SCORE) AS FINAL_SCORE FROM

(SELECT /*+ORDERED*/ DISTINCT TAB3.SUBJECT AS SUB, ( NVL(TAB3.FINAL_SCORE,0) + NVL(TAB1.FINAL_SCORE,0) ) + NVL(TAB2.FINAL_SCORE,0) ) AS AGGR_SCORE FROM

TAB3, TAB1, TAB2
WHERE TAB1.OBJECT = TAB2.OBJECT
AND TAB2.SUBJECT = TAB3.SUBJECT
)
GROUP BY SUB ORDER BY FINAL_SCORE DESC

Figure 5.8 shows the formulation of the final Select query that joins the temporary Tabo tables based on the logic given by the structured part of the SPARQL-FT query and projects the correct entities. Following the conjunctive query processing paradigm for structured queries, this translator performs an EQUI JOIN over the Tab tables to project the desired entities.

5.4.5 SPAR-Key Supremacy : The Rewriting Algorithm

We can now develop an overall rewriting algorithm by putting together all the aforementioned steps. Figure 5.9 illustrates a block diagram showing the major components
Chapter 5 SPAR-Key: Rewriting SPARQL-FT to SQL

5.4.6 Bottlenecks and Solutions

The algorithm discussed in the previous section forms the core of the SPAR-Key Supremacy translator. This algorithm counters the bottlenecks described in section 5.4.6.
and achieves decent efficiency in processing the queries. However, we observe a lot of false-positives in the final result set for a query. We identify the bottlenecks that cause this inclusion of noisy results and affect the efficiency of the query processing as follows:

- **The keyword search is always performed on the entire Wikipedia collection.** For every term in a fulltext, condition an instance of the Keywords table is constraint with at term on the Terms column. The qualifying entities from each instance, are OUTER-joined to form the temporary Keys table. This essentially signifies a search on the entire unstructured part of the collection for every term in a fulltext condition. In other words, each Keys table would contain entities with boosted score that qualify multiple constraints along with entities that qualify for only one term with very low score. It can be very easily seen that these entities with very low score mostly represent noise and increase the cardinality of the result set. Arguably, an INNER join on these entities in the Keys tables may not exactly represent a precise set of candidate entities for further processing.

  **Solution**: We need to reduce the search to a smaller candidate entity set for each term, rather than considering every entity in the Keywords table as a candidate. This can be achieved by adding contextual information to the entities bounded by a fulltext conditions and using the contextual information to prune out entities before performing a search.

- **For every fulltext condition there is an INNER join of temporary Keys tables with the entire DBpediaCore table.** The temporary Keys tables representing results from a fulltext condition, are INNER joined to the instances of DBpediaCore table considering every entity as a candidate entity to create the Tab tables. Furthermore, such INNER joins mostly lead to a nested loop join by the Oracle optimiser.

  **Solution**: To join with the Keys tables, we need to reduce the candidate entity set, rather than considering every entity in the DBpediaCore table as a candidate. This can also be achieved by adding contextual information to prune out entities before performing the INNER join.

- **For sub queries containing only variables, entire DBpedia is considered relevant.** In other words, for a triple pattern without any constricted Subject or Object by a fulltext condition or a literal, and also where the Predicate is a variable, a Tab table corresponding that triple will represent the entire DBpediaCore table.

  **Solution**: We observe that the structure of a query, chain structure or a star structure, can be exploited to derive contextual information for entities in a triple
consisting of only variables, for example, ?x ?y ?z. This additional information can be used to reduce the candidate entity set in a Tab table.

5.5 SPAR-Key Ultimatum

The goal at hand is to further improve the quality of the final results without affecting efficiency of query processing. In this section we introduce a third translator, called **SPAR-Key Ultimatum**, that aims to achieve the desired goals by further implementing a form of class-based logical partitioning of data. A common observation is that the keywords that describe an entity, in most of the benchmark queries, are their surface forms. So, this translator facilitates an additional search over the entity URIs to disambiguate entities and capture the false-negatives in the results.

It can be seen that additional contextual information can be used to prune out irrelevant entities before performing intermediate INNER joins or OUTER joins, to create temporary Tab or Keys tables (described in the 5.4.5). In many approaches like, [Neu08], introduce context columns in a statement table generally stores the Internationalized Resource Identifier (IRI) of the named graph in which the RDF triples occurs. However in our case, by context we refer to the type of an entity. For example, entity <Aircraft> has a type <MeansOfTransportation>. It is not hard to see that context or type of an entity can be derived from the DBpedia class to which it belongs. Due to neat hierarchy of classes defined in DBpedia, we can safely obtain the classification of entities. Using this knowledge we can logically partition the RDF data graph based on the classes so as to reduce the search space in the intermediate steps. Figure 5.10 shows a snap-shot of the class structure defined in DBpedia found at http://mappings.dbpedia.org. We are only concerned with the DBpedia and YAGO2 classes identified by prefix <http://dbpedia.org/ontology/class> and <http://dbpedia.org/ontology/yagoclasses/class>.

This translator also performs an initial search on the URI index which essentially stores a map of the entities to their surface forms. Even though this seems like an overhead but this actually helps to achieve higher recall and precision.

5.5.1 Entity-Classes from the Predicates

We note that DBpedia defines two kinds of properties,

1. that are neatly defined in a class identified with the prefix <http://dbpedia.org/ontology/> and,
Ontology Classes

- owl:Thing
  - Activity (edit)
    - Game (edit)
    - Sport (edit)
  - Agent (edit)
    - Organisation (edit)
      - Band (edit)
      - Broadcaster (edit)
        - BroadcastNetwork (edit)
        - RadioStation (edit)
        - TelevisionStation (edit)
      - Company (edit)
        - Airline (edit)
        - LawFirm (edit)
        - RecordLabel (edit)
      - EducationalInstitution (edit)
        - College (edit)
        - Library (edit)
        - School (edit)
        - University (edit)
      - GeopoliticalOrganisation (edit)
      - GovernmentAgency (edit)
      - Group (edit)

Figure 5.10: Snapshot of DBpedia class hierarchy [http://mappings.dbpedia.org]

2. raw properties identified with prefix <http://dbpedia.org/property/>

Since the main characteristic of this translator is to reduce the search space by considering the context or the class information of entities as additional feature, it can only handle queries with Predicates (i.e., when predicates are specified with literals) that are defined in a DBpedia class (type 1 mentioned above).

The property definition in a class specifies a signature. To understand this, let us consider an example shown in Figure 5.11. This example shows the properties defined in the <Aircraft> class with their signature. In the above example, one can see a property <aircraftUser> defined with a Domain, <Aircraft> and Range <Organisation> which are DBpedia classes. Thus if this property should occur as a Predicate in a triple pattern then all the entities classified as <Aircraft> should occur as the Subject of the triple and all the entities classified as <Organisation> should occur as the Object of the triple.

This forms an important observation to derive the classes of the entities that can occur in a triple pattern of a given query. We can therefore reduce the search space only to those entities that belong to the marked classes. Figure 5.12 shows an example SPARQL-FT query from the benchmark and the derived classes for the unknowns from the Predicates.
### Table 5.1: SPAR-Key Column Types

<table>
<thead>
<tr>
<th>Name</th>
<th>Label</th>
<th>Domain</th>
<th>Range</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>aircraftType (edit)</td>
<td>aircraft type</td>
<td>Aircraft</td>
<td>xsd:string</td>
<td></td>
</tr>
<tr>
<td>aircraftUser (edit)</td>
<td>aircraft user</td>
<td>Aircraft</td>
<td>Organisation</td>
<td></td>
</tr>
<tr>
<td>assembly (edit)</td>
<td>assembly</td>
<td>MeanOfTransportation</td>
<td>owl:Thing</td>
<td></td>
</tr>
<tr>
<td>class (edit)</td>
<td>class</td>
<td>MeanOfTransportation</td>
<td>owl:Thing</td>
<td></td>
</tr>
<tr>
<td>designerCompany (edit)</td>
<td>designer company</td>
<td>MeanOfTransportation</td>
<td>Company</td>
<td></td>
</tr>
<tr>
<td>engineType (edit)</td>
<td>engine type</td>
<td>MeanOfTransportation</td>
<td>owl:Thing</td>
<td></td>
</tr>
<tr>
<td>gun (edit)</td>
<td>aircraft gun</td>
<td>Aircraft</td>
<td>xsd:string</td>
<td></td>
</tr>
<tr>
<td>introductionDate (edit)</td>
<td>introduction date</td>
<td>MeanOfTransportation</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>modelEndDate (edit)</td>
<td>model end date</td>
<td>MeanOfTransportation</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>modelEndYear (edit)</td>
<td>model end year</td>
<td>MeanOfTransportation</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>modelStartDate (edit)</td>
<td>model start date</td>
<td>MeanOfTransportation</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>modelStartYear (edit)</td>
<td>model start year</td>
<td>MeanOfTransportation</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>numberBuilt (edit)</td>
<td>number built</td>
<td>Aircraft</td>
<td>xsd:nonNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>numberOfBombs (edit)</td>
<td>number of bombs</td>
<td>Aircraft</td>
<td>xsd:nonNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>numberOfCrew (edit)</td>
<td>number of crew</td>
<td>MeanOfTransportation</td>
<td>xsd:nonNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>numberOfLaunches (edit)</td>
<td>number of launches</td>
<td>MeanOfTransportation</td>
<td>xsd:nonNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>numberOfRockets (edit)</td>
<td>number of rockets</td>
<td>Aircraft</td>
<td>xsd:nonNegativeInteger</td>
<td></td>
</tr>
<tr>
<td>powerType (edit)</td>
<td>power type</td>
<td>MeanOfTransportation</td>
<td>owl:Thing</td>
<td></td>
</tr>
<tr>
<td>productionYears (edit)</td>
<td>production years</td>
<td>Aircraft</td>
<td>xsd:date</td>
<td></td>
</tr>
<tr>
<td>programCost (edit)</td>
<td>program cost</td>
<td>Aircraft</td>
<td>Currency</td>
<td></td>
</tr>
<tr>
<td>rebuilder (edit)</td>
<td>rebuilder</td>
<td>MeanOfTransportation</td>
<td>owl:Thing</td>
<td></td>
</tr>
<tr>
<td>relatedMeanOfTransportation (edit)</td>
<td>related mean of transportation</td>
<td>MeanOfTransportation</td>
<td>MeanOfTransportation</td>
<td></td>
</tr>
<tr>
<td>unitCost (edit)</td>
<td>unit cost</td>
<td>Aircraft</td>
<td>Currency</td>
<td></td>
</tr>
<tr>
<td>wingArea (edit)</td>
<td>wing area</td>
<td>Aircraft</td>
<td>Area</td>
<td></td>
</tr>
<tr>
<td>wingspan (edit)</td>
<td>wingspan</td>
<td>Aircraft</td>
<td>Length</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 5.11:** Snapshot of DBpedia class hierarchy [http://mappings.dbpedia.org]

```sql
SELECT ?s ?c WHERE {
  ?s rdf:type <http://dbpedia.org/ontology/Station>.
  FILTER FTContains (?s, "most beautiful railway station").
}
```

**Figure 5.12:** Marking the classes from the *Predicates*
### Index Name | Attributes
---|---
PredicateDomainIDX | (Predicate, Domain)
PredicateRangeIDX | (Predicate, Range)

**Table 5.9:** Schema of the `PredicateDomainIDX` and `PredicateRangeIDX` indexes

In the above example, the SPARQL-FT query is shown that contains two triples and one fulltext condition. The fulltext condition binds the *Object* of the first and the second triple (specified by `?a`) to a set of keywords 'most beautiful railway station'. In the first triple, the *Subject* is a variable (`?s`) and in the second triple pattern both *Subject* and *Object* are variables. It can be clearly seen that if we tokenise the fulltext condition, we get a set of terms ('beautiful', 'railway', 'station') that are most likely to return a wide range of results, i.e., ranging from cities, railway organisations, police stations etc., while it is clear to a human that the query is intended to look for railway stations only. If we analyse the *Predicate* specified in the first triple i.e., `<rdf:type>` we can straight away mark the class of *Subject* (`?s`) entities as `<Station>` and from the second triple, *Predicate* `<location>` specifies that the class of *Object* (`?c`) entities is `<Place>`. By marking these classes, we reduce the search space from millions of entities to a few thousands, i.e., only to those entities classified into these classes.

To implement this, we create two indexes that store the *Predicates* along with their *Domains* and *Ranges* separately. The first index is called the `PredicateDomainIDX` and the second index is called the `PredicateRangeIDX`. Table 5.9 depicts the schema of these indexes. These indexes facilitate the class markings of the *Subjects* and *Objects* on-the-fly while processing a query. These indexes prove not to be very large in size and can hence easily be loaded into the main memory. This makes the look-up on these indexes very fast.

#### 5.5.2 Exploiting the Query Structure

Interpreting a structured query as a basic graph pattern, leads us to observe two common kinds of query patterns. They are commonly known as a chain pattern query and a star pattern query [Neu08].

- **Chain pattern** is where the *Object* of the first triple pattern is the *Subject* of the next triple pattern, again with given *Predicates*. Figure 5.13 shows a generic illustration of a chain pattern.

- **Star pattern** is where multiple triple patterns with different *Predicates* share the same *Subject*. These are used to select specific subjects. Figure 5.14 shows a generic illustration of a star pattern.
The query pattern can sometimes be used to derive classes of entities even though in a triple pattern a Predicate is unknown or specified by a variable. In Figure 5.14 by analysing the Predicate of the first triple pattern (\texttt{<museum> located ?a.}), i.e., \texttt{located} we can mark the Object (?a) of to be class \texttt{<City>} which is specified by the Range of the Predicate signature ; and then by analysing the second Predicate \texttt{<partOf>}, we can mark the Subject (?c) of the second triple pattern (?b \texttt{<partOf> ?a.}) to be of class \texttt{<Country>}. Similar analogies can be drawn for chain query patterns also.

5.5.3 Entity Selectivity Estimation

The strategy so far described, marks the classes of the entities of a triple by analysing the Predicate. But there can be cases where two or more triples are joined on Subject or Object, with multiple classes marked. This brings us to the problem of finding the most selective class that would reduce the search space to its maximum without affecting the recall. To understand this, let us take the example query given in Figure 5.12. The variable ?s is marked with two classes namely, \texttt{<Thing>} and \texttt{<Station>}. This marking is done by analysing the first Predicate \texttt{<rdf:type>}, which marks ?s with \texttt{<Station>}, and the second Predicate \texttt{<location>} marks it with \texttt{<Thing>}. In order to resolve this, we maintain a separate index called the class selectivity index, comprising the classes and the entities classified under them. This statistic is not hard to find and could be found by issuing a single SQL query. Table 5.10 shows the schema of the class selectivity index.
5.5.4 Search on the URI Index

Document titles are considered as an important feature by IR systems among other features like content, context, etc.[Che09]. These document titles tend to summarise the major context of the articles. Following this paradigm, we also find that most of the entity descriptions or key-concepts in a fulltext condition map to the surface forms of the entity. In our collection these surface forms tend to occur as document titles. Following this heuristics, we create an additional URI index other than the above described keywords indexes that mainly stores the surface forms of the articles. For a fulltext condition, SPAR-Key Ultimatum performs an additional look-up on the URI index and then performs an OUTER join with the results of a fulltext search on the article content. By doing this, we include the entities that are missed by a fulltext search on the content. Also the scores of the entities that are found by both the searches, are boosted. Table 5.11 shows the schema of the URI index used by SPAR-Key Ultimatum.

5.5.5 Example 3

Earlier we mentioned that SPAR-Key Ultimatum works on the queries with Predicates that are defined in DBpedia classes (identified with prefix: <http://dbpedia.org/ontology/>). The query given in example 1 (Section 5.3.1) cannot be handled by this translator. In order to capture the idea behind this translator, let us take another example query \( Q_2 \) as shown in Table 5.12.

\( Q_2 \) contains two triple patterns \( T_1 \) and \( T_2 \), and two fulltext conditions \( K_1 \) and \( K_2 \). Each \( K_i \) contains a variable occurring in a triple pattern and is bound to an entity by an
A manually translated SPARQL-FT query:

```sql
SELECT ?s WHERE {
  FILTER FTContains (?x,"Johnny depp").
  FILTER FTContains(?y,"Tim Burton").
}
```

Table 5.12: A given SPARQL-fulltext query of the INEX 2012 Jeopardy task

FTContains operator. This example will be used throughout the chapter to demonstrate and compare SPAR-Key Ultimatum translation strategies.

Applying the concepts described for the SPAR-Key Ultimatum, let us now try to translate the SPARQL-FT query in Table 5.12 to SQL queries.

Figure 5.15 shows the SQL query for creation of `Keys_Search` tables. These `Keys_Search` tables holding intermediate result from a fulltext search for each term occurring in fulltext condition $K_i$. For example, in the above figure, `Key_Search0` represents the results from the fulltext condition 'Johnny depp' bounded to ?x. Note that the entities projected by a fulltext search for a term is further constricted by a class. This is realised by an Equi join on the `Entity_ID` columns of the `Entity_class` table and the `Keywords` table.

For example, in `Key_Search0`, the entities found relevant in a fulltext search for term 'johnni' (this is a result of stemming the original term which is 'Johnny') are restricted to the class 'http://dbpedia.org/ontology/Actor'. This forms a simple method to filter out noise entities from the intermediate results which also helps to achieve better efficiency by reducing the cardinality of the intermediate result sets to be joined later.

Figure 5.16 shows the SQL query for the creation of the `Keys_Final` tables. These tables hold results by combining the data from the `Keys_Search` tables and `Keys_URI` tables.

As described earlier, we create the `Keys_URI` table to represent `Entity_IDs` that are found relevant by a simple fulltext on the URIs of all the DBpedia entities. Note that we also apply the class constraint to the entities obtained from a simple fulltext search on the entity URIs. By doing this we remove the irrelevant noisy entities and greatly reduce the cardinality of the result sets. Thus each `Keys_Final` table represents the final fulltext search results for a fulltext condition $K_m$.

Figure 5.17 shows the SQL query for the creation of the `Tab` tables. This step is similar to the one described for the SPAR-Key Supremacy, i.e., each `Tab_o` table stores the intermediate result corresponding every triple pattern $T_o$.

Finally, we formulate a `Select` query that joins the intermediate results in the `Tab` tables following the join logic specified in the structured part of the SPARQL-FT query.
CREATE TABLE KEYS_SEARCH0 AS SELECT * FROM

(SELECT TEMP.ENTITY_ID AS ENTITY_ID, MAX(TEMP.AGGR_SCORE) AS FINAL_SCORE FROM
 (SELECT /*+ORDERED*/
 CASE WHEN K1.ENTITY_ID IS NULL
 THEN K2.ENTITY_ID
 ELSE K1.ENTITY_ID
 END AS ENTITY_ID,
 (NVL(K1.SCORES,0)+NVL(K2.SCORES,0)) AS AGGR_SCORE FROM
 (SELECT K.ENTITY_ID, K.SCORE AS SCORES FROM KEYWORDS K, ENTITY_CLASS C WHERE
 K.TERM='johnni'
 AND C.CLASS = 'http://dbpedia.org/ontology/Actor'
 AND C.ENTITY_ID = K.ENTITY_ID) K1
 FULL OUTER JOIN
 (SELECT K.ENTITY_ID, K.SCORE AS SCORES FROM KEYWORDS K, ENTITY_CLASS C WHERE
 K.TERM='depp'
 AND C.CLASS = 'http://dbpedia.org/ontology/Actor'
 AND C.ENTITY_ID = K.ENTITY_ID) K2
 ON K1.ENTITY_ID = K2.ENTITY_ID
) TEMP GROUP BY ENTITY_ID ORDER BY FINAL_SCORE DESC )

CREATE TABLE KEYS_SEARCH1 AS SELECT * FROM

(SELECT TEMP.ENTITY_ID AS ENTITY_ID, MAX(TEMP.AGGR_SCORE) AS FINAL_SCORE FROM
 (SELECT /*+ORDERED*/
 CASE WHEN K1.ENTITY_ID IS NULL
 THEN K2.ENTITY_ID
 ELSE K1.ENTITY_ID
 END AS ENTITY_ID,
 (NVL(K1.SCORES,0)+NVL(K2.SCORES,0)) AS AGGR_SCORE FROM
 (SELECT K.ENTITY_ID, K.SCORE AS SCORES FROM KEYWORDS K, ENTITY_CLASS C WHERE
 K.TERM='tim'
 AND C.CLASS = 'http://dbpedia.org/ontology/Person'
 AND C.ENTITY_ID = K.ENTITY_ID) K1
 FULL OUTER JOIN
 (SELECT K.ENTITY_ID, K.SCORE AS SCORES FROM KEYWORDS K, ENTITY_CLASS C WHERE
 K.TERM='burton'
 AND C.CLASS = 'http://dbpedia.org/ontology/Person'
 AND C.ENTITY_ID = K.ENTITY_ID) K2
 ON K1.ENTITY_ID = K2.ENTITY_ID
) TEMP GROUP BY ENTITY_ID ORDER BY FINAL_SCORE DESC )

Figure 5.15: Creation of the KEYS_SEARCH_i tables
Figure 5.16: Creation of the $KEYS_{FINAL0_i}$ tables

Figure 5.18 shows the final select query formulated for the example. Note that we add the class constraints to the entities to clean up any noise that have endured so far. We also note that though the constraints at this step may seem redundant and hence negatively affect the efficiency, yet by experiments we note that this step helps in obtaining an increased precision.

5.5.6 SPAR-Key Ultimatum: The Rewriting Algorithm

We can now develop an overall rewriting algorithm by putting together all the aforementioned steps. Figure 5.19 illustrates a block diagram showing the major components.
CREATE TABLE TAB1 AS

SELECT SUBJECT, PREDICATE, OBJECT, ( NVL(KEYS_FINAL0.FINAL_SCORE,0) ) AS FINAL_SCORE FROM

(SELECT * FROM DBPEDIACORE T1 WHERE T1.PREDICATE='http://dbpedia.org/ontology/starring') TEMP
INNER JOIN
KEYS_FINAL0
ON TEMP.OBJECT=KEYS_FINAL0.ENTITY_ID

CREATE TABLE TAB2 AS

SELECT SUBJECT, PREDICATE, OBJECT, ( NVL(KEYS_FINAL1.FINAL_SCORE,0) ) AS FINAL_SCORE FROM

(SELECT * FROM DBPEDIACORE T2 WHERE T2.PREDICATE='http://dbpedia.org/ontology/director') TEMP
INNER JOIN
KEYS_FINAL1
ON TEMP.OBJECT=KEYS_FINAL1.ENTITY_ID

Figure 5.17: Creation of the $TAB_i$ tables

SELECT S, MAX(AGGR_SCORE) AS FINAL_SCORE FROM

(SELECT TAB2.SUBJECT AS S, ( NVL(TAB1.FINAL_SCORE,0) + NVL(TAB2.FINAL_SCORE,0) )) AS AGGR_SCORE FROM
TAB1, TAB2
WHERE TAB1.SUBJECT = TAB2.SUBJECT AND TAB1.SUbject IN
(SELECT ENTITY_ID FROM ENTITY_CLASS WHERE CLASS = 'http://dbpedia.org/ontology/Work')
AND TAB1.OBJECT IN
(SELECT ENTITY_ID FROM ENTITY_CLASS WHERE CLASS = 'http://dbpedia.org/ontology/Actor')
AND TAB2.SUBJECT IN
(SELECT ENTITY_ID FROM ENTITY_CLASS WHERE CLASS = 'http://dbpedia.org/ontology/Work')
AND TAB2.OBJECT IN
(SELECT ENTITY_ID FROM ENTITY_CLASS WHERE CLASS = 'http://dbpedia.org/ontology/Person')
) GROUP BY S ORDER BY FINAL_SCORE DESC

Figure 5.18: Final Select query by joining the Tab tables.

Figure 5.19: Block diagram illustrating the SPAR-Key Ultimatum Engine
1. Load the features index containing frequent terms and their IDF values into main memory.

2. Tokenize and stem the `FTContains` fulltext conditions and decide the order of joins among the keywords from the features index.

3. Analyse the `Predicates` in each triple and mark the bounded variables with their class. The domain of the `Predicate` will be marked for the `Subject` and Range will be marked for the `Object`. Thus we obtain a set:

   \[
   \text{classSet} = \{\{\text{variable}_1, \text{class}_1\}, \{\text{variable}_2, \text{class}_2\}, \{\text{variable}_3, \text{class}_3\}, \ldots\}.
   \]

4. Exploit the query structure to find most selective class corresponding a variable and update the `classSet`.

5. Create `Keys_SearchUri_i` tables containing results form a search on the entity URI for each term.

6. Create temporary `Keys_i` tables for each fulltext condition: these contain the results of the OUTER join over the `Keywords` table instances constrained by the terms. Also add the class constraint on the bounded entity with the corresponding class value in `classSet`.

7. Create temporary `Keys_Search_Final_i` tables by performing a FULL OUTER JOIN on `Keys_SearchUri_i` and `Keys_i`.

8. Create temporary `Tab_i` tables for each triplet pattern. These contain the results of the INNER join over the `DBpediaCore` table instances which are additionally joined with `Keys_Search_Final_i` temporary tables for each `FTContains` fulltext condition in the query. Also add the class constrains to the variables by selecting the class values from the `classSet`.

9. Assign a default score of 1 to all triples in absence of a fulltext condition.

10. Formulate the main select query that combines the `Tab_i` temporary tables via an INNER join; the join logic is based on the joins given in the original SPARQL query.

11. Finally, drop the temporary tables `Keys_SearchUri_i`, `Keys_i`, `Keys_Search_Final_i` and `Tab_i`.
5.6 SPAR-Key Online Interface

To visualize the methods presented, we built a simple interface for the SPAR-Key engines in JSP. The goal is to present to the users a simple yet user-friendly interface to process SPARQL-FT queries in the SPAR-key engines. The interface follows the simple Google search design, i.e., with one text area to enter a query and one ‘Execute’ button to initiate the query processing. We try to make using the interface self-explanatory by adding instructions and help in each page. We design the online system with the Model View Controller (MVC) architecture, which is a standard for web applications.

The interface starts by displaying the 'Home' page of the system. This page contains the text area to input a SPARQL-FT query into the system. In addition, this page contains some example queries that a user can try out in order to gain experience with the system. This page also displays some images displaying conference presentations and publications related to the system. After issuing a query, a user is directed to a 'Results' page containing the results form SPAR-Key Supremacy and Ultimatum engines. There instruction on the page that help to interpret the returned results. This page also contains a link to the 'Statistics' page that displays the intermediate results from fulltext search obtained from both the engines. In addition, this page displays the execution statistics, i.e., the run times, of both the engines at each step. This page also displays the intermediate SQL queries that are created from the issued SPARQL-FT query in a dynamic box. The following figures present screen shots of the interface.

\footnote{www.google.com}
Combining Knowledge and Document Retrieval
Keyword retrieval on unstructured (and even semistructured) documents.

Fails to answer queries which need to combine information that is located across two or more documents. Difficult to interpret and disambiguate into a structured form. Full-text ranked documents based on a notion of relevance.

**Figure 5.20:** Home page of the interface.
Chapter 5 SPAR-Key: Rewriting SPARQL-FT to SQL

Figure 5.21: Results page of the interface.
Figure 5.22: Statistics page of the interface, showing the intermediate fulltext results.
Chapter 5 SPAR-Key: Rewriting SPARQL-FT to SQL

Execution Statistics

<table>
<thead>
<tr>
<th>Translator Supremacy</th>
<th>Translator Ultimatum</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executing FINAL SELECT</td>
<td>Creating KEYS_FINAL1</td>
</tr>
<tr>
<td>Creating KEYS5</td>
<td>Creating KEYS_FINAL2</td>
</tr>
<tr>
<td>Creating KEYS2</td>
<td>Executing FINAL SELECT</td>
</tr>
<tr>
<td>Dropping TAB3</td>
<td>Dropping KEYS_FINAL3</td>
</tr>
<tr>
<td>Creating KEYS1</td>
<td>Dropping KEYS_FINAL2</td>
</tr>
<tr>
<td>Dropping KEYS5</td>
<td>Dropping KEYS_FINAL1</td>
</tr>
<tr>
<td>Creating KEYS0</td>
<td>Creating KEYS_FINAL0</td>
</tr>
<tr>
<td>Dropping KEYS4</td>
<td>Dropping KEYS_FINAL3</td>
</tr>
<tr>
<td>Creating KEYS5</td>
<td>Dropping KEYS_FINAL1</td>
</tr>
<tr>
<td>Dropping KEYS3</td>
<td>Dropping KEYS_FINAL2</td>
</tr>
<tr>
<td>Dropping KEYS1</td>
<td>Dropping KEYS_FINAL3</td>
</tr>
<tr>
<td>Creating TAB2</td>
<td>Creating KEYS_SEARCH0</td>
</tr>
<tr>
<td>Creating TAB3</td>
<td>Creating KEYS_SEARCH2</td>
</tr>
<tr>
<td>Creating TAB2</td>
<td>Creating KEYS_SEARCH1</td>
</tr>
<tr>
<td>Creating TAB3</td>
<td>Creating KEYS_SEARCH2</td>
</tr>
<tr>
<td>Creating TAB1</td>
<td>Creating KEYS_SEARCH3</td>
</tr>
<tr>
<td>Creating TAB3</td>
<td>Creating KEYS_SEARCH0</td>
</tr>
</tbody>
</table>

The Queries

Queries from Sparkey-Ultimatum

```sql
CREATE TABLE KEYS_SEARCH AS
SELECT * FROM (SELECT TEMP.ENTITY_ID AS ENTITY_ID,
MAX(TMP.SCORES) AS SCORE FROM (SELECT * AS ORDERED) AS CASE WHEN K.ENTITY_ID IS NULL THEN
K.ENTITY_ID ELSE K.ENTITY_ID END AS ENTITY_ID,
(NVL(K1.SCORES,0)+1) + NVL(K2.SCORES,0)+1 ) AS SCORES FROM (SELECT K.ENTITY_ID, K.SCORE AS SCORES FROM KEYWORDS K, ENTITY_CLASS C WHERE
```

Queries from Sparkey-Supremacy

```sql
CREATE TABLE KEYS AS
SELECT * AS ORDERED FROM (SELECT * AS ORDERED
FROM (SELECT * AS ORDERED
DISTINCT ENTITY_ID, MAX(SCORES) AS SCORE FROM (SELECT DISTINCT K.ENTITY_ID AS ENTITY_ID,
(NVL(K1.SCORES,0)+1) + NVL(K2.SCORES,0)+1 ) AS SCORES FROM (SELECT DISTINCT ENTITY_ID, SCORE AS SCORES FROM KEYWORDS WHERE TERM='johny') AS K1 FULL OUTER JOIN (SELECT DISTINCT ENTITY_ID, SCORE AS SCORES FROM KEYWORDS WHERE TERM='dep') AS K2 ON K.ENTITY_ID = K2.ENTITY_ID ) ORDER BY SCORE DESC) GROUP BY ENTITY_ID ORDER BY SCORE DESC
```

Figure 5.23: Statistics page of the interface, showing the intermediate SQL Queries.
Chapter 6

INEX 2012 Linked Data Track

6.1 Introduction

The Initiative for the Evaluation of XML retrieval, or popularly just known as INEX, is a unique platform to evaluate search engines for focused retrieval tasks. For this, INEX provides a large test collection, uniform evaluation measures and a forum for participants or organisations to compete and compare their results. Last year for INEX’12, we introduced a new track, called the Linked Open Data (LOD) track, to investigate retrieval techniques over a combination of textual and highly structured data. The idea is RDF properties carry additional key information about semantic relations among data objects that cannot be captured by keywords alone.

Through this track we intend to investigate:

- The advantages of exploiting semantic information to improve Ad-hoc retrieval.
- Effectiveness of newly introduced SPARQL-FT queries, representing a coalition of semantic (SPARQL) and keyword queries, in helping users navigate or explore large sets of results.
- Effectiveness of newly introduced SPARQL-FT queries, in addressing Jeopardy-style natural-language questions.

The new Linked Data track at INEX 2012 thus aims to close the gap between IR-style keyword search and Semantic-Web-style reasoning techniques. Our goal is to bring together different communities and to foster research at the intersection of Information Retrieval, Databases, and the Semantic Web. As its core collection, the Linked Data track uses the Wikipedia-LOD collection discussed in Chapter 3.
For INEX 2012, we explored three different retrieval tasks:

- The classic **Ad-hoc Retrieval task** investigated informational queries to be answered mainly by the textual contents of the Wikipedia articles.

- The **Faceted Search task** employed a hand-crafted hierarchy of facets and facet-values obtained from DBpedia that aim to guide the searcher toward relevant information.

- The brand-new **Jeopardy task** employed NL Jeopardy clues which are manually translated into a semi-structured query format based on SPARQL with fulltext conditions.

### 6.2 Data Collection

The Wikipedia-LOD (v1.1) collection is hosted by the Max Planck Institute for Informatics and has been made available for download in May 2012 from the following link: [http://www.mpi-inf.mpg.de/inex-lod/wikipedia-lod-2012/](http://www.mpi-inf.mpg.de/inex-lod/wikipedia-lod-2012/)

The collection consists of three compressed tar.gz files and contains an overall amount of 3.1 Million individual XML articles. The uncompressed size of the collection is 61 GB. The detailed DTD file for the XML collection is also available from the above URL.

Each Wikipedia-LOD article consists of a mixture of XML tags and CDATA sections, containing infobox attributes, free-text contents, describing the entity or category that the article captures, and a section with both DBpedia and YAGO2 properties that are related to the article’s entity. All sections contain links to other Wikipedia articles (including links to the corresponding DBpedia and YAGO2 resources), Wikipedia categories, and external Web pages.

In addition to the new core collection, which is based on XML-ified Wikipedia articles, the Linked Data track explicitly encourages (but does not require) the use of current Linked Open Data dumps for DBpedia (v3.7) and YAGO2, which are available from the following URLs:

- DBpedia v3.7 (created in July 2011):

- YAGO2 core and full dumps (created on 2012-01-09):
Table 6.1: Organisers of INEX’12 LOD Ad-hoc Search Task

<table>
<thead>
<tr>
<th>Name</th>
<th>Organisation</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Qiuyue Wang</td>
<td>Renmin University of China</td>
<td><a href="mailto:qiuyuew@ruc.edu.cn">qiuyuew@ruc.edu.cn</a></td>
</tr>
<tr>
<td>Jaap Kamps</td>
<td>University of Amsterdam</td>
<td><a href="mailto:kamps@uva.nl">kamps@uva.nl</a></td>
</tr>
<tr>
<td>Georgina Ramirez Camps</td>
<td>Universitat Pompeu Fabra</td>
<td><a href="mailto:georgina.ramirez@upf.edu">georgina.ramirez@upf.edu</a></td>
</tr>
</tbody>
</table>

Table 6.2: Organisers of INEX’12 LOD Faceted Search Task

<table>
<thead>
<tr>
<th>Name</th>
<th>Organisation</th>
<th>Email</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maarten Marx</td>
<td>University of Amsterdam</td>
<td><a href="mailto:maartenmarx@uva.nl">maartenmarx@uva.nl</a></td>
</tr>
<tr>
<td>Anne Schuth</td>
<td>University of Amsterdam</td>
<td><a href="mailto:anne.schuth@uva.nl">anne.schuth@uva.nl</a></td>
</tr>
</tbody>
</table>

The Linked Data track was explicitly intended to be an “open track” and thus invited participants to include more Linked Data sources (see, for example, linkeddata.org) or other sources that go beyond “just” DBpedia and YAGO2. Any inclusion of further data sources was welcome, however, workshop submissions and follow-up research papers should explicitly mention these sources when describing their approaches.

6.3 Retrieval Tasks and Topics

6.3.1 Ad-hoc Task and Faceted Search Tasks

The main goal of the Ad-hoc Task is to report a ranked list of results (Wikipedia pages or equivalently DBpedia resources) relevant to the user’s information need. This information need is expressed by formulated keyword queries on a broad set of topics. A search system participating in the task may return a large result set for any query in the benchmark.

Faceted search is a way to help users navigate through the large set of results to quickly identify the results of interest. It presents the user a list of facet-values to refine the query. After the user choosing from the suggested facet-values, the result list is narrowed down and then the system may present a new list of facet-values for the user to further refine the query. The interactive process continues until the user finds the items of interest. One of the key issues in faceted search systems is to recommend appropriate facet-values to help the user quickly identify what he/she really wants in the large set of results. The task aims to investigate different techniques of recommending facet-values.

We generated and collected the topics from the following three sources.

Firstly, we built a three-level hierarchy of topics. For example,
Vietnam

Vietnam war
  Vietnam war movies
  Vietnam war facts
Vietnam food
  Vietnam food recipes
  Vietnam food blog
Vietnam travel
  Vietnam travel national park
  Vietnam travel airports

The top level represents general topics, e.g., "Vietnam". We similarly select 5 general topics, i.e., "Vietnam", "guitar", "tango", "bicycle", and "music". For each topic, we use the Google online search\(^1\) suggestion to further select three sub topics. For example, "Vietnam" typed into the Google search box may pop up the suggestions as: "Vietnam war", "Vietnam food" or "Vietnam travel", and so on. These suggestions can be viewed as subtopics to "Vietnam". A similar process was followed for further selecting the sub-subtopics. Thus in total for 5 general topics we decide 15 subtopics and 30 sub-subtopics which essentially represents the desired three-level hierarchy of topics. It can be presumed that the union of relevant documents of the sub-subtopics i.e., the leaves in the topic hierarchy tree, represents the relevant documents for the subtopics, i.e., one level higher in the topic hierarchy tree and so on. Thus for each topic only the 30 sub-subtopics need to be assessed. So we put the 30 sub-subtopics to the Ad-hoc Task and 20 non-leaf level topics to the Faceted Search Task. The relevance results for the Ad-hoc topics will serve as the relevant results to their corresponding faceted search topics.

Secondly, to compare the performance of different data collections, we select 20 topics from INEX 2009 and 2010 Ad-hoc Tracks. We select 40 worst performed topics from the INEX 2009 Ad-hoc Track and 30 worst performed topics from the INEX 2010 Ad-hoc Track( so as to select challenging topics).Then we select 10 topics from each sets randomly. In this process of selection, we found some natural general topics, "Normandy", "museum" and "social network". These general topics have multiple subtopics. So we added the 3 topics to the set of faceted search topics.

Thirdly, we add all the 90 keyword titles of Jeopardy topics (Appendix B) into the set of Ad-hoc topics so as to compare the performance of structured queries used in Jeopardy Task and unstructured queries. Finally, we collected 140 Ad-hoc topics and 23 faceted search topics, which are in the same format as that in previous years.

\(^1\)www.google.com
Table 6.3: Organisers of INEX’12 LOD Jeopardy Task

Table 6.1 gives details of the organisers of the Ad-hoc search track and Table 6.2 gives the details of the organisers of the Faceted search track.

### 6.3.2 Jeopardy Task

INEX 2012 introduced a new track called Linked Open Data (LOD) track which had three task defined over the core collection. One of them is called the Jeopardy task [Qiu12]. The goal of this track was to investigate retrieval techniques over a benchmark of 90 SPARQL-FT queries, included in Appendix B (described in Chapter 4). The queries were translated manually from Jeopardy-style natural language (NL) questions or Jeopardy! clues. An XML file with the benchmark queries was made available available for download in June 2012 under the following URL: [http://www.mpi-inf.mpg.de/inex-lod/LDT-2012-jeopardy-topics.xml](http://www.mpi-inf.mpg.de/inex-lod/LDT-2012-jeopardy-topics.xml)

For example, topic no. 2012301 from the current set of Jeopardy topics looks as follows:

```xml
<topic id="2012301" category="LAKES">
    <jeopardy_clue>
        Niagara Falls has its source of origin from this lake.
    </jeopardy_clue>
    <keyword_title>
        Niagara Falls source lake
    </keyword_title>
    <sparql_ft>
        Select ?q Where {
            <http://dbpedia.org/resource/Niagara_Falls>
            Filter FTContains(?o, "river water course niagara") .
            Filter FTContains(?q, "lake origin of")
        }
    </sparql_ft>
</topic>
```
In the above query the `<topic>` element specifies the meta-data of every query i.e., the id attribute gives a unique id to every query and the category attribute gives a broad category the query belongs to. The above query belongs to a category "Lakes". This information could be used to for fine-tuning the retrieval techniques. The `<jeopardy_clue>` element contains the original NL question or Jeopardy! clue; `<keyword_title>` element contains manually extracted set of keyword from the given natural language query ; and `<sparql_ft>` element contains the manually translated Jeopardy clue into a SPARQL-FT query. Let us analyse the SPARQL-FT query given in the above example.

We see that the query contains two triple patterns where the Subject is a literal i.e., a known DBpedia entity specified with the URI, `<http://dbpedia.org/resource/Niagara_Falls>`, the Predicate is also a literal specified with the URI, `<http://dbpedia.org/property/watercourse>` and the Object is an unknown represented with a variable `?o`. In the second triple pattern, only the Predicate is a known literal marked by `<http://dbpedia.org/ontology/origin>` and the Subject and the Object are unknown represented with `?o` and `?q` respectively. As described in the previous section a join logic in SPARQL is defined by the shared variables in the triples, in this example the two triples are joined by Object of the first triple and the Subject of the second triple as represented by a common variable (`?o`). In addition the query contains two fulltext conditions as specified by the FTContains operator. The first fulltext condition binds entities that occur in place of variable `?o` to the set of keywords "river water course niagara" and the second condition binds the entities that occur in place of variable `?q` to the set of keywords "lake origin of". This is discretely represented by the FTContains operator as per the semantics defined in the previous section. Thus this set of 90 SPARQL-FT queries formed the benchmark queries for this task.

Table 6.3 gives details of the organisers of the Jeopardy task.

### 6.4 Participations and Results

In total 20 Ad-hoc search runs were submitted by 7 participants, i.e., *Ecole des Mines de Saint-Etienne (EMSE), Kasetsart University, Renmin University of China, University of Otago, Oslo University College, University of Amsterdam, Norwegian University of Science and Technology (NTNU)*, and 5 valid Jeopardy runs were submitted by 2 participants, i.e., *Kasetsart University and Max-Planck Institute for Informatics (MPI)*.

The official results for INEX’12 Jeopardy task are discussed in Chapter 7, Section 7.3.1.
Chapter 7

Experimental Evaluations and Results

In this chapter we provide experimental evaluation of our SPAR-Key query processor over the Wikipedia-LOD collection. The evaluation studies the effectiveness of answering a Jeopardy-style Natural Language question translated into a SPARQL-FT query. In the previous chapter, we described in detail the translation mechanism for a SPARQL-FT query into a conjunctive SQL query. We also described the relational storage schema for the collection in a relational-DBMS. In this chapter, we evaluate the system by processing different SQL queries formulated by combination of translation strategies over the relational database schema and test their effectiveness.

7.1 Experimental Setup

Preprocessing of the data collection before storing into Oracle 11g relational-DBMS, is done on a machine with Intel Xeon processor at 2.79 GHz. The machine has a main memory of 64 GB and secondary memory of 1 TB. We host the demo with a Apache Tomcat server also running in the same machine. This machine is running a 64bit-Windows operating system.

To generate runs for the benchmark queries, we use a personal computer with Intel Core i3 processor at 3.30 GHz. This machine has a main memory of 8 GB and secondary memory of 200 GB. This machine is running a 64bit-Windows operating system.
Chapter 7 Experimental Evaluations and Results

7.2 Measures

In this section, we present the different measures used for building a comparison between results from the engines that took part in INEX’12 Jeopardy Task as well as the engines proposed in this thesis. We use standard TREC metrics to measure the performance of all the runs. To compare our SPAR-Key engines with Ad-hoc search engines, we use the TREC MAP metric, \( P@5, P@10, P@15 \) and show plotting of the interpolated precision values at 11 standard points. We also perform a QA style evaluation with the TREC Inverse Rank, NDCG@5, NDCG@10 and NDCG@15 of the runs generated from different combinations of translation strategies, described in this thesis, to conclude the best combination.

\[
\text{Precision (P)} = \frac{\text{Number of relevant target entities returned}}{\text{Total number of entities returned}} \tag{7.1}
\]

\[
\text{Precision at K (P@K)} = \frac{\text{Number of relevant target entities at rank K}}{K} \tag{7.2}
\]

\[
\text{Recall (R)} = \frac{\text{Number of relevant target entities returned}}{\text{Total number of relevant entities}} \tag{7.3}
\]

\text{Precision (P)} is defined as the ability of a system to present all relevant items. It is a simple statistical set-based measure calculated as shown by Equation 7.1. To measure precision, we take top-1000 entities, for each query, generated by an engine and compute the relevant target entities returned. This number is then divided by the total number of retrieved entities. To determine relevant entities we use the relevant assessment generated for evaluation of the INEX'12 LOD track. \text{Precision at rank K (P@K)} is the portion of the relevant documents in the first K positions and is calculated as shown by Equation 7.2.

\[
\text{Recall (R)} \text{ is also a set-based measure that can be perceived as the probability of a system to return correct entities. It can be computed as shown in Equation 7.3. In our case, we have two kind of queries – first, queries that look for one target entity, and second, queries that look for a ranked list of target entities. It is clear that for the second kind of queries, it is hard to build a relevant list of all the possible entities or entity-pairs. Thus in this case, we consider the entities marked relevant by the relevance assessment done for INEX’12 to compute the recall for each query.} \]
Interpolated Precision at \( r = \max_{r' \geq r}(P(r')) \), \( (7.4) \)

where, \( r \) and \( r' \) are recall levels.

To measure average performance of a system over a set of queries, each with different number of relevant entities, we compute the interpolated precision at a set of 11 standard recall levels (specifically, 1\%, 10\%, 20\%, 30\%, 40\%, 50\%, 60\%, 70\%, 80\%, 90\% and 100\%). A standard technique to compute interpolated precision at a given recall level is to use the maximum precision for any actual recall level greater than or equal to the recall level in question. This is modelled by Equation 7.4.

\[
\text{Mean Average Precision (MAP)} = \frac{1}{|Q|} \sum_{j=1}^{|Q|} \frac{1}{m_j} \sum_{k=1}^{m_j} \text{Precision}(RL_j) \tag{7.5}
\]

where, \(|Q|\) is the total number of queries,

\( m_j \) is the total number of relevant entities for a \( q_j \),

\( RL_j \) is the Ranked lst of entities returned for a query \( q_j \).

Average Precision (AP) is a single-valued measure that reflects the performance of an engine over all the relevant entities. To calculate the average precision we do not perform the averaging at each recall level, rather at each returned relevant document. AP is obtained for each query by averaging over top-1000 reported result by a system. We report the Mean Average Precision (MAP) that reflects the performance of a system over all the queries. This is simply the mean of the AP for each query. Mathematically this is shown by Equation 7.5.

The Reciprocal Rank \((1/R)\) of a query can be defined as the rank at which a system returns the first relevant entity. We set a threshold of 5 i.e., a system gets a score of zero for a query if no correct entity was returned in the top 5 results. This metric is used for evaluation in question-answering setting, rather than tradition document retrieval setting. In our case, we report an average of reciprocal rank score, also known as Mean Reciprocal Rank (MRR), obtained for all the queries for a system.
Finally we present the Normalised Discounted Cumulative Gain (NDCG) at top 5, 10 and 15 results to evaluate systems in a QA setting. Discounted Cumulative Gain (DCG) uses a graded relevance scale to measure the gain of a system based on the positions of the relevant entities in the result set. This measure gives a lower gain to relevant entities returned in the lower ranks to that of the higher ranks. This makes a sensible measure to use for our task as we reward engines that retrieve relevant entities as top results. NDCG reports a single valued score by normalising the DCG accounting for differently sized output lists. \( \text{NDCG}(Q, K) \), i.e., NDCG at \( K \) for a set of queries \( Q \) is computed as shown in Equation 7.6.

\[
\text{NDCG}(Q, k) = \frac{1}{|Q|} \sum_{j=1}^{|Q|} Z_{kj} \sum_{m=1}^k \frac{2^{R(j,m)} - 1}{\log_2(1 + m)}
\]

where, \( |Q| \) is the total number of queries, 
\( R(j, e) \) is the relevance score obtained for an entity for a query \( j \), 
\( Z_{jk} \) is the normalization factor, 
\( k \) is the rank at which NDCG is calculated.

### 7.3 Experimental Runs

In this section we provide evaluations in the traditional document retrieval settings and QA settings. In the first setting, we make an assessment of the relevant entities by crowd sourcing the top-100 run results from the run pool of submitted results by all the systems. These evaluations also represent the official results of the INEX’12 LOD Jeopardy Task. For the second setting, we carefully create a gold result set of entities as answers to the benchmark queries and use this for evaluation.

### 7.3.1 Ad-hoc Search style Evaluation

For the INEX’12 Jeopardy task, from the benchmark queries, 50 (Appendix A, Section A.1) were selected for assessment of the systems that participated in the task. To generate the relevant entity list, we pooled all the submitted runs for each query in a round-robin manner, and then picked up the top-100 results to be assessed. The idea was to view the Jeopardy Task as a known item search task. For the assessment, the popular on-line crowd sourcing portal Amazon Mechanical Turk (AMT) was used. For
### Table 7.1: Official INEX’12 Jeopardy Task evaluations for 50 Jeopardy topics

<table>
<thead>
<tr>
<th>Run</th>
<th>MAP</th>
<th>1/rank</th>
<th>P@5</th>
<th>P@10</th>
<th>P@20</th>
<th>P@30</th>
</tr>
</thead>
<tbody>
<tr>
<td>Renmin-LDT2012_adhoc_ruc_comb07</td>
<td>0.3195</td>
<td>0.7655</td>
<td>0.416</td>
<td>0.306</td>
<td>0.231</td>
<td>0.188</td>
</tr>
<tr>
<td>Amsterdam-inex12LDT.adhoc.baseline_LM</td>
<td>0.2704</td>
<td>0.7615</td>
<td>0.4</td>
<td>0.294</td>
<td>0.208</td>
<td>0.1673</td>
</tr>
<tr>
<td>Otago-on2012pr09</td>
<td>0.3264</td>
<td>0.741</td>
<td>0.396</td>
<td>0.318</td>
<td>0.233</td>
<td>0.188</td>
</tr>
<tr>
<td>NTNU-run1</td>
<td>0.3014</td>
<td>0.7099</td>
<td>0.42</td>
<td>0.316</td>
<td>0.228</td>
<td>0.1733</td>
</tr>
<tr>
<td>Kasetsart-kas16-PHR</td>
<td>0.1434</td>
<td>0.7</td>
<td>0.18</td>
<td>0.16</td>
<td>0.085</td>
<td>0.0633</td>
</tr>
<tr>
<td>EMSE-run-085</td>
<td>0.3157</td>
<td>0.6979</td>
<td>0.421</td>
<td>0.316</td>
<td>0.235</td>
<td>0.186</td>
</tr>
<tr>
<td>MPI-submission</td>
<td>0.1618</td>
<td>0.5991</td>
<td>0.2732</td>
<td>0.1829</td>
<td>0.1061</td>
<td>0.0772</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recall</th>
<th>ou2012pr09</th>
<th>LDT2012</th>
<th>run-085</th>
<th>run1</th>
<th>inex12LDT</th>
<th>MPI</th>
<th>kas16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.7646</td>
<td>0.7867</td>
<td>0.7278</td>
<td>0.737</td>
<td>0.7774</td>
<td>0.6096</td>
<td>0.7</td>
</tr>
<tr>
<td>10%</td>
<td>0.6998</td>
<td>0.6869</td>
<td>0.6586</td>
<td>0.6792</td>
<td>0.6776</td>
<td>0.4552</td>
<td>0.3445</td>
</tr>
<tr>
<td>20%</td>
<td>0.5634</td>
<td>0.5627</td>
<td>0.5663</td>
<td>0.5595</td>
<td>0.542</td>
<td>0.2801</td>
<td>0.206</td>
</tr>
<tr>
<td>30%</td>
<td>0.4545</td>
<td>0.4509</td>
<td>0.4531</td>
<td>0.4508</td>
<td>0.4022</td>
<td>0.2138</td>
<td>0.206</td>
</tr>
<tr>
<td>40%</td>
<td>0.3737</td>
<td>0.3603</td>
<td>0.369</td>
<td>0.3429</td>
<td>0.2985</td>
<td>0.1538</td>
<td>0.1755</td>
</tr>
<tr>
<td>50%</td>
<td>0.3272</td>
<td>0.3114</td>
<td>0.3245</td>
<td>0.2608</td>
<td>0.2296</td>
<td>0.1063</td>
<td>0.1574</td>
</tr>
<tr>
<td>60%</td>
<td>0.2305</td>
<td>0.2113</td>
<td>0.2181</td>
<td>0.1792</td>
<td>0.1312</td>
<td>0.0381</td>
<td>0</td>
</tr>
<tr>
<td>70%</td>
<td>0.1613</td>
<td>0.1538</td>
<td>0.1546</td>
<td>0.1317</td>
<td>0.0838</td>
<td>0.0295</td>
<td>0</td>
</tr>
<tr>
<td>80%</td>
<td>0.1176</td>
<td>0.1057</td>
<td>0.1125</td>
<td>0.0948</td>
<td>0.0514</td>
<td>0.0294</td>
<td>0</td>
</tr>
<tr>
<td>90%</td>
<td>0.0787</td>
<td>0.0713</td>
<td>0.0733</td>
<td>0.0619</td>
<td>0.0269</td>
<td>0.0278</td>
<td>0</td>
</tr>
<tr>
<td>100%</td>
<td>0.0667</td>
<td>0.0502</td>
<td>0.0564</td>
<td>0.0498</td>
<td>0.0223</td>
<td>0.0249</td>
<td>0</td>
</tr>
</tbody>
</table>
each query, a Mechanical Turk Worker (MTW) was presented the Natural Language (NL) question and a set containing 5 links to Wikipedia articles. These articles correspond to the DBpedia entities returned by a system in top 100 results. The goal of a MTW was to decide the relevance of the articles to the NL question. The relevance assessments were conducted by Renmin University of China. Such a relevant assessment is commonly used for evaluating IR engines for Ad-hoc search tasks.

Table 7.1 shows the official INEX results over the Jeopardy topics. In total, 7 groups submitted results to the Jeopardy task. We observe that Renmin University of China (0.7655) runs the first in terms of the mean reciprocal rank (1/rank), but University of Otago (0.741) has the best MAP. The second best scoring team in terms of reciprocal rank is University of Amsterdam. It is worthwhile to mention that of all the groups that submitted run results, we were the only team that exploited both the structured and the unstructured part of the query. The other groups exploited only the keyword titles to perform an Ad-hoc search over the topics.

### 7.3.2 Creation of a Gold Result Set

We find that such a relevant assessment and evaluation, performed for the INEX’12 Jeopardy task, does not truly apply to SPAR-Key engine. Our system focuses on retrieving the entities that represent the correct answers to a NL question by processing the translated SPARQL-FT query. This essentially means that we select only those entities that follow a specific structure in the data graph and are also associated to a set of key-concepts. To clearly understand this, let us take a query from the benchmark and analyse its assessment results.

```xml
<topic id="2012305" category="Flowers">
    <jeopardy_clue>
        North Dakota’s highest point is White Butte; its lowest is on this river of another colour.
    </jeopardy_clue>
    <keyword_title>
        North Dakota’s lowest river of another colour
    </keyword_title>
    <sparql_ft>
        <escape>
            SELECT DISTINCT ?o1 WHERE
            {
```
<http://dbpedia.org/resource/North_Dakota>
FILTER
   FTContains (?o1, "river colour lowest point north Dakota") .
}
...
</topic>

The above Jeopardy-style NL question is looking for only one target entity, i.e.,

However, a Ad-hoc search style relevance assessment marks the following entities as the relevant entities:

http://dbpedia.org/resource/geography_of_north_dakota
http://dbpedia.org/resource/pembina_county,_north_dakota
http://dbpedia.org/resource/north_dakota
http://dbpedia.org/resource/1997_red_river_flood_in_the_united_states

This style of relevance assessment is is fine for an Ad-hoc search task. This is because, the Wikipedia article corresponding to the marked relevant entities contain valid (or relevant) information for the given keyword titles, "North Dakota’s lowest river of another colour". However, they fail to represent the answer to the NL question. Moreover, they fail to qualify the structural constraint imposed by the SPARQL part of the SPARQL-FT query. To tackle this, we create a gold result set or ground truth, and used it to evaluate our system.

This gold result set comprises of the correct answers to the original Jeopardy questions taken as the NL questions for building the benchmark. The idea is to have the exact answers of the Jeopardy question correctly mapped to DBpedia entities. Identification of correct entities to a question and building of the gold set is done by manual assessment by our research group.
7.3.3 QA style Evaluation with Gold Result Set

We set up a QA style evaluation for our SPAR-Key as well as for the groups that submitted their runs to INEX’12 Jeopardy Task. For this new evaluation, we use the gold result set instead of the official INEX assessments for the Jeopardy task. In this sub-section we re-evaluate the official INEX run submission for the Jeopardy Task and present the new results.

<table>
<thead>
<tr>
<th>Runs</th>
<th>1/rank</th>
<th>MAP</th>
<th>P@5</th>
<th>P@10</th>
<th>P@15</th>
<th>ndcg@5</th>
<th>ndcg@10</th>
<th>ndcg@15</th>
</tr>
</thead>
<tbody>
<tr>
<td>kas16</td>
<td>0.0916</td>
<td>0.0249</td>
<td>0.0353</td>
<td>0.0294</td>
<td>0.0314</td>
<td>0.0376</td>
<td>0.0336</td>
<td>0.0543</td>
</tr>
<tr>
<td>LDT2012</td>
<td>0.2484</td>
<td>0.1744</td>
<td>0.1077</td>
<td>0.0808</td>
<td>0.0692</td>
<td>0.1994</td>
<td>0.2029</td>
<td>0.2176</td>
</tr>
<tr>
<td>Supremacy</td>
<td>0.5135</td>
<td>0.3835</td>
<td>0.1826</td>
<td>0.1087</td>
<td>0.0783</td>
<td>0.4337</td>
<td>0.435</td>
<td>0.4331</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recall</th>
<th>kas16</th>
<th>LDT2012</th>
<th>Supremacy</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.093</td>
<td>0.2643</td>
<td>0.521</td>
</tr>
<tr>
<td>10%</td>
<td>0.0798</td>
<td>0.2622</td>
<td>0.4764</td>
</tr>
<tr>
<td>20%</td>
<td>0.0581</td>
<td>0.2543</td>
<td>0.4609</td>
</tr>
<tr>
<td>30%</td>
<td>0.028</td>
<td>0.1947</td>
<td>0.4408</td>
</tr>
<tr>
<td>40%</td>
<td>0.0243</td>
<td>0.1878</td>
<td>0.4304</td>
</tr>
<tr>
<td>50%</td>
<td>0.018</td>
<td>0.1741</td>
<td>0.3605</td>
</tr>
<tr>
<td>60%</td>
<td>0.0058</td>
<td>0.1515</td>
<td>0.3492</td>
</tr>
<tr>
<td>70%</td>
<td>0.0053</td>
<td>0.1357</td>
<td>0.3466</td>
</tr>
<tr>
<td>80%</td>
<td>0.0053</td>
<td>0.1234</td>
<td>0.3162</td>
</tr>
<tr>
<td>90%</td>
<td>0.0053</td>
<td>0.1221</td>
<td>0.3162</td>
</tr>
<tr>
<td>100%</td>
<td>0.0053</td>
<td>0.1221</td>
<td>0.3162</td>
</tr>
</tbody>
</table>

Table 7.2: Jeopardy Task re-evaluations with gold result set for 26 queries
7.3.3.1 Comparison to the INEX Competitors

Table 7.2 shows the new results obtained with the gold set created by us. The table compares the best performing group as per official results, LDT2012, to the two groups that submitted runs for the Jeopardy task, kas16 and MPI. We evaluate the engines for 26 queries (Appendix A, Section A.2) selected from the benchmark. We observe that with the new gold set used for evaluations, the precision values at recall levels of the Ad-hoc search engines reduces drastically. This is primarily due the exclusion of 'relevant documents' from the gold set that may not represent target entities. We also observe that the inverse rank scores of the Ad-hoc search engines also reduce and our engine stands out as the top one. From these results, one may conclude that traditional keyword-based document retrieval techniques may not perform well in an Entity-retrieval or QA tasks. On the other hand, the addition of structured information to the queries becomes useful to retrieve correct entities as top results.

7.3.3.2 Extended evaluation of Supremacy Components

Earlier we presented a heuristic that most of the entity descriptions or key-concepts in fulltext conditions map to the surface forms of the entity. In our collection these surface forms tend to occur as document titles (Chapter 5, Section 5.5.4). Due to this, we designed a strategy to perform an additional fulltext search on the entity URIs and adding the entities as candidate entities. In this sub-section, we attempt to study this through experiments by adding an URI search component to the SPAR-Key Supremacy. For performing these set of experiments, we select 26 queries from the benchmark and perform the evaluation using the gold result set created by us.

Table 7.3 shows the evaluation results of the experiments performed to study the effect of URI-search on 26 queries (Appendix A, Section A.2). We observe that there is no significant change to the scores. However these scores represent average scores over all the queries. These scores indicate that even though for some queries, we observe rise in precision with the inclusion of false negatives, we also observe a decrease in precision of queries that have fulltext conditions with keywords (concepts) that do not describe entities. Due to this observation, we do not include this component to the translation strategy designed for SPAR-Key Supremacy in Chapter 5, Section 5.4.5.

7.3.3.3 Comparison of Supremacy and Ultimatum

In this thesis, we presented two translation algorithms for translating SPARQL-FT queries into SQL queries namely, SPAR-Key Supremacy and Ultimatum. We highlight
Chapter 7 Experimental Evaluations and Results

<table>
<thead>
<tr>
<th>Runs</th>
<th>1/R</th>
<th>MAP</th>
<th>P@5</th>
<th>P@10</th>
<th>P@15</th>
<th>ndcg@5</th>
<th>ndcg@10</th>
<th>ndcg@15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Supremacy</td>
<td>0.5135</td>
<td>0.5135</td>
<td>0.1826</td>
<td>0.1087</td>
<td>0.0783</td>
<td>0.4337</td>
<td>0.435</td>
<td>0.4331</td>
</tr>
<tr>
<td>Supremacy + URI Search</td>
<td>0.4923</td>
<td>0.3618</td>
<td>0.1667</td>
<td>0.0958</td>
<td>0.0667</td>
<td>0.4086</td>
<td>0.4079</td>
<td>0.4034</td>
</tr>
</tbody>
</table>

Recall | Supremacy | Supremacy + URI Search

<table>
<thead>
<tr>
<th>Recall Level</th>
<th>Supremacy</th>
<th>Supremacy + URI Search</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.521</td>
<td>0.4995</td>
</tr>
<tr>
<td>10%</td>
<td>0.4764</td>
<td>0.4567</td>
</tr>
<tr>
<td>20%</td>
<td>0.4609</td>
<td>0.4145</td>
</tr>
<tr>
<td>30%</td>
<td>0.4408</td>
<td>0.4022</td>
</tr>
<tr>
<td>40%</td>
<td>0.4304</td>
<td>0.3978</td>
</tr>
<tr>
<td>50%</td>
<td>0.3605</td>
<td>0.3459</td>
</tr>
<tr>
<td>60%</td>
<td>0.3492</td>
<td>0.3376</td>
</tr>
<tr>
<td>70%</td>
<td>0.3466</td>
<td>0.3343</td>
</tr>
<tr>
<td>80%</td>
<td>0.3162</td>
<td>0.3035</td>
</tr>
<tr>
<td>90%</td>
<td>0.3162</td>
<td>0.3034</td>
</tr>
<tr>
<td>100%</td>
<td>0.3162</td>
<td>0.3034</td>
</tr>
</tbody>
</table>

Table 7.3: Effect of merging Supremacy with fulltext search on entity URIs for 26 queries

that we process SPARQL-FT queries that have raw DBpedia properties in the Supremacy engine and process SPARQL-FT queries that have classified DBpedia properties in the Ultimatum engine (Chapter 5, Section 5.5.1). In this sub-section we show experimental results that compare both the engines on 11 selected queries (Appendix A, Section A.3) from the benchmark that represents the second kind queries, i.e., queries that have classified DBpedia properties in the SPARQL part.

Table 7.4 shows the evaluation results of SPAR-Key Supremacy and Ultimatum on 11 queries selected from the benchmark. We observe a significant difference in the inverse
Table 7.4: Comparisons of Supremacy and Ultimatum for 11 queries
Ultimatum. In this extended evaluation, we select 11 SPARQL-FT queries (Appendix A, Section A.3) from the benchmark that have classified DBpedia properties in the formulation of the SPARQL part. To perform the extended evaluation, we generate runs by activating individual components of the SPAR-Key Ultimatum translator at a time and then generate evaluation result using the new gold result set.

<table>
<thead>
<tr>
<th>Runs</th>
<th>1/Rank</th>
<th>MAP</th>
<th>P@5</th>
<th>P@10</th>
<th>P@15</th>
<th>ndcg@5</th>
<th>ndcg@10</th>
<th>ndcg@15</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exploiting Dbpedia class information</td>
<td>0.6017</td>
<td>0.3377</td>
<td>0.26</td>
<td>0.16</td>
<td>0.12</td>
<td>0.4251</td>
<td>0.3965</td>
<td>0.3922</td>
</tr>
<tr>
<td>Boosting direct matches on entity URIs</td>
<td>0.6365</td>
<td>0.3836</td>
<td>0.26</td>
<td>0.15</td>
<td>0.1067</td>
<td>0.4753</td>
<td>0.4435</td>
<td>0.4327</td>
</tr>
<tr>
<td>Merging both the approaches</td>
<td>0.7035</td>
<td>0.4219</td>
<td>0.26</td>
<td>0.15</td>
<td>0.1067</td>
<td>0.5081</td>
<td>0.4764</td>
<td>0.4655</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Recall</th>
<th>Exploiting Dbpedia class information</th>
<th>Boosting direct matches on entity URIs</th>
<th>Merging both the approaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>1%</td>
<td>0.6017</td>
<td>0.6532</td>
<td>0.7035</td>
</tr>
<tr>
<td>10%</td>
<td>0.4989</td>
<td>0.5504</td>
<td>0.6007</td>
</tr>
<tr>
<td>20%</td>
<td>0.4633</td>
<td>0.4491</td>
<td>0.5032</td>
</tr>
<tr>
<td>30%</td>
<td>0.4171</td>
<td>0.4196</td>
<td>0.4737</td>
</tr>
<tr>
<td>40%</td>
<td>0.3943</td>
<td>0.4104</td>
<td>0.4645</td>
</tr>
<tr>
<td>50%</td>
<td>0.3359</td>
<td>0.3858</td>
<td>0.4378</td>
</tr>
<tr>
<td>60%</td>
<td>0.3129</td>
<td>0.3657</td>
<td>0.4099</td>
</tr>
<tr>
<td>70%</td>
<td>0.3069</td>
<td>0.358</td>
<td>0.4055</td>
</tr>
<tr>
<td>80%</td>
<td>0.1826</td>
<td>0.2839</td>
<td>0.2847</td>
</tr>
<tr>
<td>90%</td>
<td>0.1825</td>
<td>0.2838</td>
<td>0.2835</td>
</tr>
<tr>
<td>100%</td>
<td>0.1825</td>
<td>0.2838</td>
<td>0.2835</td>
</tr>
</tbody>
</table>

Table 7.5: Extended evaluation of the Ultimatum components on 11 queries
Table 7.5 shows the evaluation results of the SPAR-Key Ultimatum engine with only one component activated. As already mentioned, the use of DBpedia class information reduces the entity search space and also helps in cleaning the false positives from the results. On the other hand, the inclusion of entities by separate fulltext searches on the entity URIs helps to include the false negatives. However, from the results we observe that the true advantage lies in merging the strategies rather than using them separately.

We observe that we get an inverse rank score of 0.617 by only exploiting the DBpedia classes and we get an inverse rank of 0.6365 by only exploiting the additional entity URI search. By merging the two strategies, i.e., by performing entity URI search and then by increasing the selectivity by class information, we get a significant rise of 10% in inverse rank score. Moreover, we also see a rise in the MAP value. This clearly indicates that the engine is able to return highly relevant results when both the components are activated.
Chapter 8

Conclusion and Future Work

Question-Answering (QA) over Natural Language (NL) remains an open research challenge. With the evolution of research, retrieval systems have presented many perspectives and approaches to stand up to this challenge. However, so far these systems ultimately use either traditional keyword queries or more modern SPARQL structured queries to represent a given NL question. In this thesis, we presented that the true advantage lies in merging both the approaches of query translation. Our main goal was to close the gap between IR-style keyword search and semantic-web-style reasoning techniques.

To investigate the advantage of merging structured and unstructured data, we introduced a unique data collection, the Wikipedia-LOD collection. This entity-centric collection contains XML-ified documents with structured RDF data and unstructured (or textual) Wikipedia article contents corresponding to a DBpedia entity. Moreover, the design of the data collection was such that it could accommodate for multiple sources of linked data and could be easily scaled by adding more sources. Due to these reasons, this collection reserved the properties of a linked open data collection and we highlighted this property by adding LOD to its name.

Having built the document collection, we designed a new query format by extending the W3C standard SPARQL with fulltext conditions and called it SPARQL-FT queries. These queries essentially represented a collation of SPARQL and keyword queries that could be processed over the collection. In order to investigate the advantage of QA tasks with such queries, we designed a benchmark of 90 SPARQL-FT queries manually translated from Jeopardy style NL questions. The benchmark consisted of two kinds of queries, first that looked for a single target entity, and second that looked for a ranked list of target entities. We motivated that such queries could easily express a NL question and return better results.
To process the SPARQL-FT queries over the document collection, we proposed the SPARKey engine. We parsed the documents and stored it in a relational-DBMS, specifically Oracle 11g, following the relational perspective to store structured and unstructured data. To show that the SPARQL-FT queries could be translated into conjunctive SQL queries and be processed over the relational database schema. For the translation, we summarized two translation algorithms namely, SPAR-Key Supremacy and SPAR-Key Ultimatum.

We organised the Linked Data Track, which was a new track in INEX 2012. The track is one of the earliest guiding themes of INEX, namely to investigate whether structure may help to improve the results of Ad-hoc keyword search. As a core of this effort, we introduced the document collection introduced in the thesis, or Wikipedia-LOD v1.1, of XML-ifed Wikipedia articles, which were additionally annotated with RDF-style resource-property pairs from both DBpedia and YAGO2. This document collection serves as the basis for three tasks: i) the Ad-hoc Retrieval Task, ii) the Faceted Search Task, and iii) a new Jeopardy Task, which were all held as part of last year’s Linked Data Track. We the query benchmark designed in this thesis was made available for the Jeopardy Task by allowing the participant to evaluate their engines on a common platform. We also took part in the Jeopardy Task by submitting runs from the initial prototype of the SPAR-Key Supremacy engine.

By experimental evaluations, we showed that indeed SPARQL-FT queries retrieve better results in QA settings. However, in a document retrieval style evaluation, our engine did not perform well since we focused on retrieving correct entities rather than relevant document. This was clearly shown by the official INEX evaluation for the Jeopardy Task. For evaluating in a QA setting, we create a gold result set by mapping the answers to the Jeopardy questions to DBpedia entities. The experiments showed a significant increase in effectiveness indicating clear advantage of exploiting structured and unstructured information for entity-retrieval or QA tasks.

As a part of the future work, we would like to explore more on the organisation of the Wikipedia-LOD collection introduced by us. We aim, to make it an apt collection for Ad-hoc retrieval tasks as well as entity-retrieval tasks. We plan to release an even more comprehensive collection, Wikipedia-LOD v 1.2, for INEX 2013 and hope to receive more participation from groups for the Jeopardy task. We also plan to design a more concrete framework for translating a NL question into a SPARQL-FT query.

As an enhancement for the SPAR-Key engine, we would like to explore more with the advanced keyword-based retrieval techniques, like keyword expansions, context identification from fulltext, etc. We also plan to develop an automated translation framework for translating a Jeopardy-style NL question into a SPARQL-FT query. Such
a complete system could find a number of real-world applications like, recommender system, question answering systems like Siri (introduced by iPhone), expert systems, and so on.
List of Figures

1.1 Top 5 results by Google for query “This mountain range is bordered by another mountain range and is a popular sightseeing and sports destination” (as on 24th Jan, 2013) .................................................. 4
1.2 Unified data collection ............................................................................. 5

2.1 Sample data and possible properties table [Gar08] ................................. 11
2.2 Data store schema of Sesame [Bro02] .......................................................... 13
2.3 Architecture of Jena2 [Kev03] .................................................................. 14
2.4 Sample data and possible properties table with an example query [Aba09] 15
2.5 FREyA work-flow chart [Dam12] ................................................................. 18
2.6 Architectural components of PowerAqua [LFMS11] ................................. 19
2.7 Query framework [Elb09] ......................................................................... 21

3.1 XML-ified Wikipedia articles with DBPedia and YAGO2 properties ............ 25

4.1 Sample RDF data of some music albums .................................................... 30
4.2 Jeopardy! game show screen (from Google Images) ................................. 34
4.3 Type 1 : Query that aims for one target entity ........................................... 37
4.4 Type 2 : Query that aims for one a ranked list of target entities ............... 37

5.1 Graph representation of the sample RDF data. ......................................... 44
5.2 Relational table representation of RDF data ............................................ 45
5.3 Example showing the translation of keyword scores to the entity-relationship graph .............................................................. 51
5.4 A basic conjunctive query from a SPARQL-FT query .............................. 53
5.5 Optimizer Statistics stored in the data dictionary used by the Optimizer to determine execution plans [www.oracle.com] ........................................ 55
5.6 Queries to create the Keys temporary tables ............................................ 59
5.7 Queries to create the TAB temporary tables ............................................ 60
5.8 The final SELECT query to obtain the answer ......................................... 60
5.9 Block diagram illustrating the SPAR-Key Supremacy Engine .................. 61
5.10 Snapshot of DBpedia class hierarchy [http://mappings.dbpedia.org] .......... 64
5.11 Snapshot of DBpedia class hierarchy [http://mappings.dbpedia.org] .......... 65
5.12 Marking the classes from the Predicates ................................................. 65
5.13 Chain query structure ............................................................................. 67
5.14 Star query structure ................................................................................ 67
5.15 Creation of the KEYS_SEARCHi tables .................................................. 70
5.16 Creation of the KEYS_FINAL0i tables .................................................. 71
5.17 Creation of the TABi tables .................................................................. 72
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.18</td>
<td>Final Select query by joining the Tab tables.</td>
<td>72</td>
</tr>
<tr>
<td>5.19</td>
<td>Block diagram illustrating the SPAR-Key Ultimatum Engine</td>
<td>72</td>
</tr>
<tr>
<td>5.20</td>
<td>Home page of the interface.</td>
<td>75</td>
</tr>
<tr>
<td>5.21</td>
<td>Results page of the interface.</td>
<td>76</td>
</tr>
<tr>
<td>5.22</td>
<td>Statistics page of the interface, showing the intermediate fulltext results.</td>
<td>77</td>
</tr>
<tr>
<td>5.23</td>
<td>Statistics page of the interface, showing the intermediate SQL Queries.</td>
<td>78</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Wikipedia-LOD (v1.1) collection statistics ........................................ 27
5.1 Table schema the DBpediaCore table .............................................. 44
5.2 Table Summary of the DBpediaCore table ........................................ 46
5.3 Table schema of the Keywords table ............................................... 47
5.4 Table Summary of the Keywords table ............................................. 48
5.5 Schema of the DBpediaCore table .................................................. 49
5.6 Indexes built over the Keywords table ............................................. 49
5.7 Example to demonstrate a special case in translating scores from keywords to SPO triples. ...................................................... 52
5.8 A given SPARQL-fulltext query of the INEX 2012 Jeopardy task .......... 53
5.9 Schema of the PredicateDomainIDX and PredicateRangeIDX indexes ..... 66
5.10 Schema of the Class Selectivity index ........................................... 68
5.11 Schema of the URI index ............................................................. 68
5.12 A given SPARQL-fulltext query of the INEX 2012 Jeopardy task .......... 69
6.1 Organisers of INEX’12 LOD Ad-hoc Search Task .............................. 81
6.2 Organisers of INEX’12 LOD Faceted Search Task .............................. 81
6.3 Organisers of INEX’12 LOD Jeopardy Task ....................................... 83
7.1 Official INEX’12 Jeopardy Task evaluations for 50 Jeopardy topics ...... 89
7.2 Jeopardy Task re-evaluations with gold result set for 26 queries .......... 92
7.3 Effect of merging Supremacy with fulltext search on entity URIs for 26 queries ................................................................. 94
7.4 Comparisons of Supremacy and Ultimatum for 11 queries ................ 95
7.5 Extended evaluation of the Ultimatum components on 11 queries ......... 96
A.1 50 Topics selected for official INEX Jeopardy task evaluations .......... 107
A.2 26 topics selected for the re-evaluations of the INEX Jeopardy task with the gold result set ..................................................... 107
A.3 11 topics selected to evaluate Spar-Key Ultimatum engine ................. 108
Appendix A

Topics for the Evaluations

A.1 50 Topics Selected for Official INEX Evaluations

Table shows the 50 topics selected for evaluating the runs submitted by the participants. For these topics a Ad-hoc search style assessment was conducted by the

<table>
<thead>
<tr>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012301</td>
<td>11</td>
<td>2012319</td>
<td>21</td>
<td>2012337</td>
<td>31</td>
<td>2012355</td>
</tr>
<tr>
<td>2</td>
<td>2012303</td>
<td>12</td>
<td>2012321</td>
<td>22</td>
<td>2012339</td>
<td>32</td>
<td>2012357</td>
</tr>
<tr>
<td>3</td>
<td>2012305</td>
<td>13</td>
<td>2012323</td>
<td>23</td>
<td>2012341</td>
<td>33</td>
<td>2012359</td>
</tr>
<tr>
<td>4</td>
<td>2012307</td>
<td>14</td>
<td>2012325</td>
<td>24</td>
<td>2012343</td>
<td>34</td>
<td>2012361</td>
</tr>
<tr>
<td>5</td>
<td>2012309</td>
<td>15</td>
<td>2012327</td>
<td>25</td>
<td>2012345</td>
<td>35</td>
<td>2012363</td>
</tr>
<tr>
<td>6</td>
<td>2012311</td>
<td>16</td>
<td>2012329</td>
<td>26</td>
<td>2012347</td>
<td>36</td>
<td>2012365</td>
</tr>
<tr>
<td>7</td>
<td>2012313</td>
<td>17</td>
<td>2012331</td>
<td>27</td>
<td>2012349</td>
<td>37</td>
<td>2012367</td>
</tr>
<tr>
<td>8</td>
<td>2012315</td>
<td>18</td>
<td>2012333</td>
<td>28</td>
<td>2012351</td>
<td>38</td>
<td>2012369</td>
</tr>
<tr>
<td>9</td>
<td>2012317</td>
<td>19</td>
<td>2012335</td>
<td>29</td>
<td>2012353</td>
<td>39</td>
<td>2012371</td>
</tr>
<tr>
<td>10</td>
<td>2012318</td>
<td>20</td>
<td>2012336</td>
<td>30</td>
<td>2012354</td>
<td>40</td>
<td>2012372</td>
</tr>
</tbody>
</table>

Table A.1: 50 Topics selected for official INEX Jeopardy task evaluations

A.2 26 Topics Selected for INEX Re-Evaluations

Table shows the 26 topics selected for re-evaluating the runs submitted by the participants for INEX’12 Jeopardy task with the gold results set. We also

<table>
<thead>
<tr>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012317</td>
<td>7</td>
<td>2012336</td>
<td>13</td>
<td>2012349</td>
<td>19</td>
<td>2012361</td>
</tr>
<tr>
<td>2</td>
<td>2012318</td>
<td>8</td>
<td>2012337</td>
<td>14</td>
<td>2012351</td>
<td>20</td>
<td>2012363</td>
</tr>
<tr>
<td>3</td>
<td>2012319</td>
<td>9</td>
<td>2012339</td>
<td>15</td>
<td>2012353</td>
<td>21</td>
<td>2012367</td>
</tr>
<tr>
<td>4</td>
<td>2012321</td>
<td>10</td>
<td>2012341</td>
<td>16</td>
<td>2012354</td>
<td>22</td>
<td>2012369</td>
</tr>
<tr>
<td>5</td>
<td>2012329</td>
<td>11</td>
<td>2012343</td>
<td>17</td>
<td>2012355</td>
<td>23</td>
<td>2012375</td>
</tr>
<tr>
<td>6</td>
<td>2012331</td>
<td>12</td>
<td>2012347</td>
<td>18</td>
<td>2012359</td>
<td>24</td>
<td>2012377</td>
</tr>
</tbody>
</table>

Table A.2: 26 topics selected for the re-evaluations of the INEX Jeopardy task with the gold result set
A.3  11 Topics Selected for INEX Re-Evaluations

Table shows the 11 topics selected for comparing SPAR-Key Supremacy and Ultimatum engines. We also use these queries to compare the individual components of the SPAR-Key Ultimatum engine. For the evaluations of these queries we use the gold result set designed manually.

<table>
<thead>
<tr>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
<th>S.no</th>
<th>Topic ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2012317</td>
<td>4</td>
<td>2012329</td>
<td>7</td>
<td>2012353</td>
<td>10</td>
<td>2012369</td>
</tr>
<tr>
<td>2</td>
<td>2012318</td>
<td>5</td>
<td>2012337</td>
<td>8</td>
<td>2012354</td>
<td>11</td>
<td>2012390</td>
</tr>
<tr>
<td>3</td>
<td>2012325</td>
<td>6</td>
<td>2012343</td>
<td>9</td>
<td>2012355</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Table A.3:* 11 topics selected to evaluate Spar-Key Ultimatum engine
Appendix B

SPARQL-FT Queries constituting the Query Benchmark

<jeopardy_clue>
Niagara Falls has its source of origin from this lake.
</jeopardy_clue>
(keyword_title)Niagara falls origin lake
<sparql_ft>
<escape>
SELECT ?q WHERE {
FILTER FTContains (?o, "river water course Niagara") .
FILTER FTContains (?q, "lake origin") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012302" category="People and places">
<jeopardy_clue>When Canarians fly home, they fly here.</jeopardy_clue>
(keyword_title)home Canarians
<sparql_ft>
<escape>
SELECT ?sub WHERE {
?x ?r ?s .
?sub ?o ?s .
?sub rdf:type <http://dbpedia.org/ontology/Place> .
FILTER FTContains (?x, "Canarian") .
FILTER FTContains (?sub, "place islands country") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012303" category="Valleys">
<jeopardy_clue>A fungal infection common in this state’s San Joaquin valley is now called Valley fever.</jeopardy_clue>
(keyword_title)Valley fever fungal infection San Joaquin
<sparql_ft>
<escape>
SELECT DISTINCT ?s3 WHERE {
?s2 ?r3 ?s3 . FILTER FTContains (?s3, "region place valley") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012304" category="Flowers">

109
<jeopardy_clue>
This river’s 350-foot drop at the Zambia–Zimbabwe border creates Victoria Falls.
</jeopardy_clue>

<keyword_title>
rivers’s 350-foot drop Zambia–Zimbabwe Victoria Falls
</keyword_title>

<sparql_ft>
<escape>
SELECT DISTINCT ?o WHERE {
FILTER FTContains (?x, "Victoria Falls") .
FILTER FTContains (?o, "river water course Victoria 350-foot drop Zimbabwe") .
}
</escape>
</sparql_ft>

<topic id="2012305" category="Flowers">
<jeopardy_clue>
North Dakota’s highest point is White Butte; its lowest is on this river of another colour.
</jeopardy_clue>

<keyword_title>North Dakota’s lowest river of another colour</keyword_title>

<sparql_ft>
<escape>
SELECT DISTINCT ?o1 WHERE {
FILTER FTContains (?o1, "river colour lowest point north Dakota") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012306" category="Heads of state">
<jeopardy_clue>
This president was born in 1921 and died in Indonesia.
</jeopardy_clue>

<keyword_title>president born in 1921 died in Indonesia</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
?s rdf:type <http://dbpedia.org/ontology/President> .
FILTER FTContains (?x, "Indonesia") .
FILTER FTContains (?s, "death place Indonesia born in 1921") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012307" category="U.S. Presidents">
<jeopardy_clue>
On July, 1850 this president died in office; Millard Fillmore was sworn in the following day.
</jeopardy_clue>

<keyword_title>July, 1850 president died Millard Fillmore sworn following day</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
FILTER FTContains (?s, "July, 1850 president died in office") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012308" category="Sports">
<jeopardy_clue>
Rosie Ruiz reportedly rode the subway to reach the finish line of this 1979 NYC race.

SELECT ?s WHERE {
  ?o rdfs:label ?l .
  ?s ?r ?l .
  FILTER FTContains (?x, "Rosie Ruiz") .
  FILTER FTContains (?s, "race Rosie Ruiz scandal") .
}

Most of the residents of this small island city–state off the Malay Peninsula are Chinese.

SELECT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/Malay-speakingCountriesAndTerritories> .
  FILTER FTContains (?s, "Most residents small island city–state Malay Peninsula Chinese") .
}

This country’s capital is Wellington.

SELECT ?s WHERE {
}

"(Just Like) Starting Over" is one of the songs from this 1980 John Lennon & Yoko Ono album.

SELECT ?s WHERE {
  FILTER FTContains (?o, "(Just Like) Starting Over") .
  FILTER FTContains (?x, "John Lennon Yoko Ono") .
}

He directed "8 1/2", a movie about a movie director.

SELECT ?o WHERE {
  FILTER FTContains (?s, "Eight and one half 8 1/2 movie director") .
}
<topic id="2012313" category="Movies about the movies">
<jeopardy_clue>
John Turturro played this title screenwriter in a 1991 Coen Brothers film.
</jeopardy_clue>
<keyword_title>John Turturro 1991 Coen Brothers film</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
FILTER FTContains(?s, "John Turturro played 1991 Coen Brothers film") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012314" category="20th century literature">
<jeopardy_clue>
Middle name of "Naked and the Dead" author Mailer or first name of "Lucky Jim" author Amis.
</jeopardy_clue>
<keyword_title>Middle name of "Naked and the Dead" author Mailer or first name of "Lucky Jim" author Amis</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
FILTER FTContains(?x, "Lucky Jim") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012315" category="Around the world">
<jeopardy_clue>
Cities in this country include Baguio, Quezon City & Manila got their official independence in 1945.
</jeopardy_clue>
<keyword_title>Baguio Quezon City Manila official independence 1945</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
FILTER FTContains(?x, "Baguio") .
FILTER FTContains(?y, "Quezon City") .
FILTER FTContains(?z, "Manila") .
FILTER FTContains(?s, "official independence in 1945") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012316" category="Around the world">
<jeopardy_clue>
7 countries that make up Central America.
</jeopardy_clue>
<keyword_title>countries Central America</keyword_title>
<sparql_ft>
<escape>
SELECT ?c WHERE {
FILTER FTContains(?x, "Central America") .
FILTER FTContains(?y, "Central America") .
FILTER FTContains(?z, "Central America") .
}
</escape>
</sparql_ft>
</topic>
<topic id="2012317" category="Novel vocabulary">
<jeopardy_clue>
He may have coined the term daggeroso, meaning "inclined to use a dagger"; it's in his novel "Sons and Lovers".
</jeopardy_clue>
</topic>

<topic id="2012318" category="The bela lugosi file">
<jeopardy_clue>
This man directed Bela in 3 films, "Glen or Glenda?", "Bride of the Monster" & "Plan 9 from Outer Space".
</jeopardy_clue>
</topic>

<topic id="2012319" category="Oxymorons">
<jeopardy_clue>
Shh! A 1994 short story collection by Alice Munro is titled "Open" these.
</jeopardy_clue>
</topic>

<topic id="2012320" category="Authors">
<jeopardy_clue>
His "Pickwick Papers" was originally published serially under the pseudonym Boz.
</jeopardy_clue>
</topic>
<topic id="2012321" category="Si, the world">
<jeopardy_clue>
This Asian port state–city wasn’t won in a raffle, but it was founded by one Sir Stamford Raffles.
</jeopardy_clue>
<keyword_title>Asian port state–city Sir Stamford Raffles</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
  FILTER FTContains (?s, "Asian port state–city founded Sir Stamford Raffles") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012322" category="In Paris">
<jeopardy_clue>
A Parisian must–see on anyone’s list, this museum is the home of the Mona Lisa.
</jeopardy_clue>
<keyword_title>Parisian museum Mona Lisa</keyword_title>
<sparql_ft>
<escape>
SELECT ?o WHERE {
  FILTER FTContains (?x, "Mona Lisa") .
  FILTER FTContains (?o, "museum home Mona Lisa") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012323" category="Cosmopolitan">
<jeopardy_clue>
Large glaciers covering this island nation include Langjökull, Hofsjökull & Vatnajökull.
</jeopardy_clue>
<keyword_title>Large glaciers island nation Langjökull Hofsjökull Vatnajökull</keyword_title>
<sparql_ft>
<escape>
SELECT ?x WHERE {
  FILTER FTContains (?x, "Vatnajökull") .
  FILTER FTContains (?y, "Langjökull") .
  FILTER FTContains (?z, "Hofsjökull") .
  FILTER FTContains (?x, "Large glaciers covering island nation") .
}
</escape>
</sparql_ft>
</topic>

<topic id="2012324" category="State capitals">
<jeopardy_clue>
This capital’s executive mansion was once the home of James G. Blaine.
</jeopardy_clue>
<keyword_title>capital’s executive mansion James G. Blaine</keyword_title>
<sparql_ft>
<escape>
SELECT ?o1 WHERE {
  FILTER FTContains (?x, "James G. Blaine house") .
}
</escape>
</sparql_ft>
Appendix B SPARQL-FT Queries constituting the Query Benchmark

115

This successor of James G. Blaine studied law.

```
SELECT ?o WHERE {
  FILTER FTContains (?o, "studies law") .
  FILTER FTContains (?x, "James G. Blaine house") .
}
```

The island of Mindanao is a Muslim centre in this mainly Roman Catholic country.

```
SELECT ?o WHERE {
  FILTER FTContains (?z, "Mindanao") .
}
```

This "Beloved" author was the first African–American to win a Nobel Prize for Literature.

```
SELECT ?x WHERE {
  FILTER FTContains (?x, "author first African–American win Nobel Prize Literature author beloved") .
}
```

Locally, this world capital is spelled W–I–E–N.

```
SELECT DISTINCT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/CapitalsInEurope> .
  FILTER FTContains (?s, "Wien") .
}
```

Like Sweden, Iceland uses currency called this.

```
SELECT ?o WHERE {
  FILTER FTContains (?o, "Sweden Iceland currency") .
}
```
This is an ambassador to the US and an active member of the Knesset.

Seoul, Korea straddles this river which shares its name with the main ethnic group of China.

In the 1960s Michael Abdul Malik of the U.K. & Malcolm Little of the U.S. both used this as their last name.

This prime minister of Canada was nicknamed "Silver-Tongued Laurier", the longest unbroken term of office among Prime Ministers.
<topic id="2012334" category="Little-read books">

<jeopardy_clue>
This author of "An American Tragedy" also wrote a little-read treatise called "Tragic America".
</jeopardy_clue>

<keyword_title>author American Tragedy little-read treatise Tragic America</keyword_title>

<sparql_ft><escape>
SELECT ?o WHERE {
?x <http://dbpedia.org/ontology/author> ?o . FILTER FTContains (?o, "An American Tragedy and little-read treatise called Tragic America") . }
</escape></sparql_ft>

</topic>

<topic id="2012335" category="Famous names">

<jeopardy_clue>
He was the U.S. president to authorise nuclear weapons against Japan.
</jeopardy_clue>

<keyword_title>U.S. president authorise nuclear weapons against Japan</keyword_title>

<sparql_ft><escape>
SELECT ?s WHERE {
?s rdf:type <http://dbpedia.org/class/yago/PresidentsOfTheUnitedStates> . FILTER FTContains (?s, "authorize nuclear weapons against Japan") . }
</escape></sparql_ft>

</topic>

<topic id="2012336" category="Islands">

<jeopardy_clue>
In 1906 the territory of Papua was established on this island as a protectorate under Australian control.
</jeopardy_clue>

<keyword_title>1906 territory Papua island Australian</keyword_title>

<sparql_ft><escape>
SELECT ?o WHERE {
</escape></sparql_ft>

</topic>

<topic id="2012337" category="Twister">

<jeopardy_clue>
This Texas city, the home of Baylor University, still remembers the tornado of 1953.
</jeopardy_clue>

<keyword_title>Texas city Baylor University tornado 1953</keyword_title>

<sparql_ft><escape>
SELECT ?o WHERE {
</escape></sparql_ft>

</topic>

<topic id="2012338" category="Historic chicago">

<jeopardy_clue>
A graduate of Chicago's De Paul University, he was mayor from 1955 to 1976.
</jeopardy_clue>

<keyword_title>A graduate of Chicago's De Paul University, he was mayor from 1955 to 1976</keyword_title>

<sparql_ft><escape>
SELECT ?o WHERE {
?x <http://dbpedia.org/ontology/graduateOf> ?o . FILTER FTContains (?o, "A graduate of Chicago's De Paul University, he was mayor from 1955 to 1976") . }
</escape></sparql_ft>

</topic>
APPENDIX B SPARQL-FT QUERIES CONSTITUTING THE QUERY BENCHMARK

〈jeopardy_clue〉
graduate Chicago’s De Paul University mayor 1955-1976
</jeopardy_clue>

〈keyword_title〉
graduate Chicago’s De Paul University mayor 1955-1976
</keyword_title>

〈sparql_ft〉
〈escape〉
SELECT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/ontology/Politician> .
  FILTER FTContains (?s, "mayor from 1955 to 1976") .
} .
</escape>
</sparql_ft>

〈topic id="2012339" category="Abbreviations">
〈jeopardy_clue〉
What do Nelson Mandela and John Dube have in common to each other.
</jeopardy_clue>
〈keyword_title〉Nelson Mandela John Dube</keyword_title>

〈sparql_ft〉
〈escape〉
SELECT ?o WHERE {
  FILTER FTContains (?x, "John Dube") .
} .
</escape>
</sparql_ft>

〈topic id="2012340" category="Russian cities">
〈jeopardy_clue〉
Grozny is the capital city of this region that has waged a struggle for independence since the mid-1990s.
</jeopardy_clue>
〈keyword_title〉Grozny capital struggle independence 1990s</keyword_title>

〈sparql_ft〉
〈escape〉
SELECT ?o WHERE {
  FILTER FTContains (?x, "Grozny") .
} .
</escape>
</sparql_ft>

〈topic id="2012341" category="U.S. Presidents">
〈jeopardy_clue〉
In 1997 a Houston airport was renamed this in honour of recent president.
</jeopardy_clue>
〈keyword_title〉1997 Houston airport president</keyword_title>

〈sparql_ft〉
〈escape〉
SELECT ?s WHERE {
} .
</escape>
</sparql_ft>

〈topic id="2012342" category="Foreign colleges & universities">
〈jeopardy_clue〉
The university of this city grew out of the cathedral schools of Notre Dame.
</jeopardy_clue>
〈keyword_title〉university cathedral schools Notre Dame</keyword_title>

〈sparql_ft〉
〈escape〉
SELECT ?s WHERE {
  FILTER FTContains (?s, "university cathedral schools Notre Dame") .
} .
</escape>
</sparql_ft>
Appendix B SPARQL-FT Queries constituting the Query Benchmark

119

119

119

119

119

The first nonfiction work selected, "The Heart of a Woman" is volume 4 of this poet's autobiography.

The Heart of a Woman poet's autobiography

SELECT ?s WHERE {
  FILTER FTContains (?x, "The Heart of a Woman").
}

A Sunday Afternoon on the Island of La Grande Jatte Art Institute Chicago

SELECT ?s WHERE {
  FILTER FTContains (?x, "A Sunday Afternoon on the Island of La Grande Jatte the Art Institute of Chicago").
}

Kennedy assassination governor of Texas seriously injured

SELECT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/GovernorsOfTexas>. FILTER FTContains (?s, "Kennedy assassination seriously injured").
}

Undershaw, the home where he wrote "The Hound of the Baskervilles", is now a hotel in Surrey, England.

Undershaw home The Hound of the Baskervilles hotel Surrey England

SELECT ?s WHERE {
  FILTER FTContains (?x, "The Hound of the Baskervilles").
}
It’s the seat of the Florida country formerly known as Dade.

SELECT ?o WHERE {
  ?s http://dbpedia.org/ontology/countySeat ?o . FILTER FTContains (?s, “Florida county seat Dade”).}

During the Middle Ages, this capital was mostly confined to Stadsholmen & Riddarholmen islands.

SELECT ?s WHERE {
  ?x rdf:type ?s . ?y rdf:type ?s . FILTER FTContains (?x, "Stadsholmen") . FILTER FTContains (?y,"Riddarholmen") .}

Alexander Nevsky Cathedral in the centre of this Bulgarian city celebrates its liberation from the Turks.

SELECT ?l WHERE {

This saint taught at the University of Paris while working on “Summa Theologica” in the 13th century.

SELECT ?l WHERE {

What is a famous Indian Cuisine dish that mainly contains rice, dhal, vegetables, roti and papad?
Appendix B SPARQL-FT Queries constituting the Query Benchmark

<jeopardy_clue>
Indian Cuisine dish rice dhal vegetables roti papad
</jeopardy_clue>

<keyword_title>
Indian Cuisine dish rice dhal vegetables roti papad
</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
?s ?p <http://dbpedia.org/resource/Category:Indian_cuisine> . FILTER FTContains (?s, " rice dhal vegetables roti papad") . }
</escape>
</sparql_ft>

<topic id="2012352" category="Cricket">
<jeopardy_clue>
Who is the famous One Day Cricket and an Indian Cricket Test Captain also known as "India's Best Schoolboy Cricketer"?
</jeopardy_clue>

<keyword_title>
One Day Cricket Indian Test Captain India’s Best Schoolboy Cricketer
</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
?s rdf:type <http://dbpedia.org/class/yago/IndianTestCaptains> . FILTER FTContains (?s, "India's Best Schoolboy Cricketer") . }
</escape>
</sparql_ft>

<topic id="2012353" category="Official language">
<jeopardy_clue>
In which country is the German language the predominant spoken language?
</jeopardy_clue>

<keyword_title>
country German language
</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
?s rdf:type <http://dbpedia.org/ontology/Country> . FILTER FTContains (?s, "predominant spoken German language") . }
</escape>
</sparql_ft>

<topic id="2012354" category="Music">
<jeopardy_clue>
Who is the greatest guitarist of all times?
</jeopardy_clue>

<keyword_title>
greatest guitarist
</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
?s <http://dbpedia.org/ontology/instrument> <http://dbpedia.org/resource/Guitar> . FILTER FTContains (?s, "greatest guitarist all times") . }
</escape>
</sparql_ft>

<topic id="2012355" category="Sports">
<jeopardy_clue>
Which England football player has been the highest paid footballer of the world?
</jeopardy_clue>

<keyword_title>
England football player highest paid
</keyword_title>

<sparql_ft>
<escape>
SELECT ?s WHERE {
</escape>
</sparql_ft>
Which monuments built by Mughals in India are also World Heritage Sites and entirely made of marble?

SELECT ?s WHERE {
  FILTER FTContains (?s, "Mughals entirely made Marble") .
}

Who became their prima ballerina of Bolshoi Theatre in 1960?

SELECT ?s WHERE {
  FILTER FTContains (?s, "1960 prima ballerina Bolshoi Theatre") .
}

A celebrity couple where the husband "is described as one of the world’s most attractive men" and wife has been cited as the world’s "most beautiful woman".

SELECT ?s ?o WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/Actor109765278> .
  FILTER FTContains (?s, "world’s most attractive male actor") .
  FILTER FTContains (?o, "most beautiful female actor") .
}

Bob Ricker was the Executive Director of this organisation which was described as "the latest front group for the anti-gun movement".

SELECT ?s ?o WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/Actor109765278> .
  FILTER FTContains (?s, "world’s most attractive male actor") .
  FILTER FTContains (?o, "most beautiful female actor") .
}
Appendix B SPARQL-FT Queries constituting the Query Benchmark

  ?s2 ?p2 ?o2 . FILTER FTContains (?o2, "Bob Ricker") . ?s1 ?p3 ?s1 . FILTER FTContains (?s1, "latest front group anti-gun movement") . }
</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

SELECT ?s WHERE { ?s rdf:type <http://dbpedia.org/class/yago/RockFestivalsInAustralia> . FILTER FTContains (?s, "Beenie Man and Odd Future controversy") . }
</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>

</escape>
</sparql_ft>
</topic>
Appendix B SPARQL-FT Queries constituting the Query Benchmark

SELECT ?s WHERE {
  FILTER FTContains (?s, "American professional tennis double players") .
}
</sparql_ft>
</topic>

<topic id="2012364" category="Business world">
<jjeopardy_clue>Which business family was greatly involved in saving tens of thousands Jewish lives during World War II?
</jeopardy_clue>
</keyword_title>
<sparql_ft>
SELECT ?s WHERE {
  FILTER FTContains (?s, "saving thousands Jewish lives") .
}
</sparql_ft>
</topic>

<topic id="2012365" category="Science">
<jjeopardy_clue>Which mathematician turned computer scientist was designated as one of MIT's six inaugural MacVicar Faculty Fellows?
</jeopardy_clue>
</keyword_title>
<sparql_ft>
SELECT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/AmericanComputerScientists> .
  FILTER FTContains (?s, "MIT's six inaugural MacVicar Faculty Fellows") .
}
</sparql_ft>
</topic>

<topic id="2012366" category="Science">
<jjeopardy_clue>Which German mathematicians were also science fiction writers?
</jeopardy_clue>
</keyword_title>
<sparql_ft>
SELECT ?s WHERE {
  ?s rdf:type <http://dbpedia.org/class/yago/GermanMathematicians> .
  FILTER FTContains (?s, "science fiction writer") .
}
</sparql_ft>
</topic>

<topic id="2012367" category="Inventions">
<jjeopardy_clue>Who invented the telescope?
</jeopardy_clue>
</keyword_title>
<sparql_ft>
SELECT ?s WHERE {
  FILTER FTContains (?s, "invented telescope") .
}
</sparql_ft>
</topic>
<topic id="2012368" category="Music">
<jeopardy_clue>
Which very famous music composer converted to Islam in 1989?
</jeopardy_clue>
<keyword_title>famous music composer converted Islam 1989</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
?s rdf:type <http://dbpedia.org/class/yago/IndianComposers> .
FILTER FTContains (?s, "converted to Islam in 1989") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012369" category="War">
<jeopardy_clue>Which are the most famous civic–military airports?</jeopardy_clue>
<keyword_title>most famous civic–military airports</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE {
FILTER FTContains (?s, "civic–military airport") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012370" category="Movies">
<jeopardy_clue>What are famous couples of actors acting in crime movies?
</jeopardy_clue>
<keyword_title>famous couples actors crime movies</keyword_title>
<sparql_ft>
<escape>
SELECT Distinct ?s ?o WHERE {
FILTER FTContains (?m1, "crime movie") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012371" category="Places and People">
<jeopardy_clue>What are the most beautiful railway stations in the world, and in which cities are they located?
</jeopardy_clue>
<keyword_title>most beautiful railway stations world cities located</keyword_title>
<sparql_ft>
<escape>
SELECT ?s ?c WHERE {
?s rdf:type <http://dbpedia.org/ontology/Station> .
FILTER FTContains (?s, "most beautiful railway station") .}
</escape>
</sparql_ft>
</topic>

<topic id="2012372" category="War">
<jeopardy_clue>What are famous historical battlefields, and which opponents fought there?
</jeopardy_clue>
<keyword_title>famous historical battlefields opponents fought</keyword_title>
<sparql_ft>
</escape>
SELECT ?s ?c WHERE {
  FILTER FTContains ( ?s, "most famous battle") .
}

SELECT ?s WHERE {
}

SELECT ?s ?s1 WHERE {
  FILTER FTContains ( ?s, "stepped down early") .
}

SELECT ?s WHERE {
  FILTER FTContains ( ?s, "lays eggs") .
}

SELECT  ?s WHERE {
  ?p1 ?s .
  ?p2 ?s .
  FILTER ( ?s != ;
  <http://dbpedia.org/ontology/Person> ).
  FILTER ( ?s != <http://schema.org/Person> ).
  FILTER ( ?s != <http://xmlns.com/foaf/0.1/Person> ).
}
Appendix B SPARQL-FT Queries constituting the Query Benchmark

<topic id="2012377" category="War">

What were the events that allegedly caused World War I?

SELECT ?s WHERE {
  FILTER FTContains (?s, "Causes of World War I") .
}
</topic>

<topic id="2012378" category="Places and People">

Which countries are located in Europe but are not part of the European Union?

SELECT ?s2 WHERE {
  MINUS { ?s2 rdf:type <http://dbpedia.org/class/yago/EuropeanUnionMemberStates> . }
}
</topic>

<topic id="2012379" category="Places and People">

Which are pairs of cities, in which the same language is spoken, lie on the same longitude but are located in different countries?

  ?c1 http://www.w3.org/2003/01/geo/wgs84_pos#long> ?l .
  ?c2 http://www.w3.org/2003/01/geo/wgs84_pos#long> ?l .
  FILTER (?c1 != ?c2) .
  FILTER (?coun1 != ?coun2) .
}
</topic>

<topic id="2012380" category="Music">

Which musicians have played in more than two bands?

SELECT ?s WHERE {
  FILTER FTContains (?s, "Musicians who have played in more than two bands") .
}
</topic>
SELECT ?s WHERE { ?s rdf:type <http://dbpedia.org/ontology/Scientist> . FILTER FTContains ( ?s , "vegetarian") . }
</sparql_ft>
</topic>

<topic id="2012385" category="Politics">
<jeopardy_clue>Which famous politicians are or were vegetarians?</jeopardy_clue>
<keyword_title>famous politicians vegetarians</keyword_title>
<sparql_ft>
<escape>
SELECT ?s WHERE { ?s rdf:type <http://umbel.org/umbel/rc/Politician> . FILTER FTContains ( ?s , "vegetarians") . }
</sparql_ft>
</topic>

<topic id="2012386" category="Movies">
<jeopardy_clue>What are famous crime couples occurring in crime movies?</jeopardy_clue>
<keyword_title>famous crime couples movies</keyword_title>
<sparql_ft>
<escape>
</sparql_ft>
</topic>

<topic id="2012387" category="Rivers and Confluences">
<jeopardy_clue>Which are the famous rivers formed of a confluence of two rivers on which dams have been constructed?</jeopardy_clue>
<keyword_title>famous river confluence dam constructed</keyword_title>
<sparql_ft>
<escape>
</sparql_ft>
</topic>

<topic id="2012388" category="Geography">
<jeopardy_clue>Which mountain range is bordered by another mountain range and is a popular sightseeing and sports destination?</jeopardy_clue>
<keyword_title>popular sightseeing and sports destination mountain range bordered</keyword_title>
<sparql_ft>
<escape>
</sparql_ft>
</topic>
Which gulfs of the Indian Ocean are frequently visited by sharks?
Which baseball player scored the most homeruns while playing in the national league?
Bibliography


[cit] Towards web scale rdf.


[Dav09] Dave Kolas and Ian Emmons and Mike Dean. Efficient linked-list rdf indexing in parliament, 2009.


[Luo12] Luo, Yongming and Picalausa, François and Fletcher, George H. L. and Hidders, Jan and Vansummeren, Stijn. Storing and Indexing Massive RDF


