SPAR-Key: Processing SPARQL-Fulltext Queries to Solve Jeopardy! Clues

Arunav Mishra[1], Sairam Gurajada[1], and Martin Theobald[2]

[1] Max Planck Institute for Informatics, Campus E1.4, Saarbrücken, Germany
[2] University of Antwerp, Middelheimlaan 1, 2020 Antwerp, Belgium

Abstract. We describe our SPAR-Key query engine that implements indexing, ranking, and query processing techniques to run a new kind of SPARQL-fulltext queries that were provided in the context of the INEX 2013 Jeopardy Task.

1 Introduction

The LOD track of INEX 2012 introduced the new Wikipedia-LODv1.1 collection that combined highly structured semantic data and unstructured (or semi-structured) textual data with the goal of improving IR-tasks (adhoch search task and faceted search task) and Question-Answering tasks (Jeopardy task). The entity-centric collection comprised XML-ified documents, coined Wiki-XML documents, as the basic unit of data, where each documents corresponded to a Wikipedia entity, and combined semantic (RDF) data from DBpedia and YAGO2 Knowledge Bases, and textual data from the Wikipedia article that describe the entity. The Jeopardy task, defined on the Wikipedia-LODv1.1 collection as a part of the LOD track, evaluated retrieval techniques over a unique query benchmark of 90 queries, given in the new SPARQL-FT format. The SPARQL-FT queries of the benchmark represented a translation of Jeopardy-style Natural Language (NL) questions into a combination of W3C standard SPARQL and traditional keyword queries [7]. In INEX 2013, the same efforts have been continued for the LOD track, however, with a new and more complete version of the collection, Wikipedia-LODv2.0, and a larger query benchmark for the Jeopardy task with 105 queries.

In this paper, we introduce a query engine, called SPAR-Key, which we developed in the context of our participation in the Jeopardy Task of the INEX 2013 Linked Data Track. This work has been a continuation of our attempts in the INEX 2012 Jeopardy task [1]. At a high level, the SPAR-Key engine translates a SPARQL-FT query into a conjunctive SQL query and processes it over a relational database schema. We delineate three SPARQL-FT-to-SQL translators as a part of the SPAR-Key engine, namely SPAR-Key Supremacy, SPAR-Key Ultimatum-No Phrases, and SPAR-Key Ultimatum-Phrases, and submit runs based on each variant. In addition, we motivate and analyze the translation strategies, and investigate the improvements in the quality of results against the official evaluations released by the INEX community.

2 Data Storage in Relational-DBMS

Data management and storage becomes a critical in designing a query processor. Thinking of scalable data storage, the most common option is a Relational Data Management

Column	Type
`N3ID`	NUMBER
`Subject`	VARCHAR2(1024)
`Predicate`	VARCHAR2(1024)
`Object`	VARCHAR2(1024)

Table 1. Table schema the `DBpediaCore` table

Attribute	Value
Number Of Rows	200187000
Blocks	4288182
Distinct_ Rows_ Subject	18272256
Distinct_Rows_Object	26873856
Distinct_Rows_Predicate	18310

Table 2. Table Summary of the `DBpediaCore` table

System (relational-DBMS). Though we also explored other approaches like file systems, graph-databases, etc., a relational-DBMS based approach seemed to be a viable option due to the ease of implementation and re-use of the optimization techniques generally adopted by the database community. Designing a respective database schema as our storage model suffices our need, as we aim to investigate the effectiveness and not maximize the efficiency of the query processing.

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, *Entity*, *Property* and *Value* (in entity-relationship terminology). In the last decade, there have been many perspectives put forward by different research communities to manage RDF data. We identify the three most important in order to find the best suited data storage model for our collection, namely *a relational perspective, an entity perspective and a graph-based perspective* [4], and finally, we adopt the relational perspective for our system.

Storing RDF Data as a Single Relational Table. The vertical representation under the relational perspective enables us to view RDF data as a large collection of triples containing SPO components. By assuming such a *serialization* of RDF data into a flat relational table, simply means that a given structured query in SPARQL has to be translated into a SQL query and issued to the relational-DBMS.

As our storage back-end, we use the Oracle 11g relational-DBMS to store the RDF data that we imported from the data dumps of DBpedia and YAGO2. Table 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 on, we refer to the table with this name.

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 (v3.8) and YAGO2, which are available from the following URLs:

- DBpedia v3.8:
  http://downloads.dbpedia.org/3.8/en/
- YAGO2 core and full dumps:
  http://www.mpi-inf.mpg.de/yago-naga/yago/

Column	Type
Entity_ID	VARCHAR2(1024)
Term	VARCHAR2(1024)
Score	NUMBER

Table 3. Table schema of the Keywords table

Number Of Rows	1688869800
Blocks	15228747
Distinct_ Rows_Entity_ID	6180437
Distinct_Rows_Term	1576558

Table 4. Table Summary of the Keywords table

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 an RDF store. However, we use the JAVA interface provided by the Jena TDB to traverse over all the triples and further bulk load them 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 2 shows some statistics of the DBpediaCore table.

2.2 Storage Model for Unstructured Data

Traditionally, inverted indices are used as core data structures in keyword-based retrieval systems. Abstractly, these inverted indices facilitate efficient fulltext searches and retrieval of most relevant or best matched documents to a given keyword query. Essentially, inverted indices map every term (as keys) in the corpus to a set of documents with a similarity score that is generated by a scoring function. We use a similar approach however by first generating and later translating such an inverted index into a relational table.

Storing Textual Data in a Single Relational Table. 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. Realizing this, we create a relational table called **Keywords** table to store all the term-entity pairs extracted from the Wikipedia fulltext collection. From here on we refer to the table with this name. The schema of this table is shown in Table 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. Every tuple of the Keywords table represents a term mapped to DBpedia entity and a similarity score to the entity's Wikipedia page.

Creation of the Keywords Table. We employee a regular SAX parser to parse the XML articles whose general XML structure is still based on that of the original articles. That is, these articles contain a meta-data header with information about the ID, authors, creation date and others, usually also an infobox with additional semi-structured

Index Name	**Attributes**
`DBpediaIDX_Obj`	`(Object,Subject,Predicate)`
`DBpediaIDX_Sub`	`(Subject,Predicate,Object)`
`DBpediaIDX_Prd`	`(Predicate,Object,Subject)`

Table 5. Schema of the `DBpediaCore` table

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 [5], 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 4 shows some statistics of the `Keywords` table.

2.3 Indexes on the Relational Tables

We note that solving a complex SPARQL query, with one or 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 optimization by facilitating index creations over a relational table.

Many approaches motivate to build multiple indexes for different permutations and combinations of the triple components. For example RDF-3x [6] creates 15 indexes over a triple store. However, we observe that we already achieve decent efficiency with three non-unique, visible and composite indexes on `DBpediaCore` table as shown in Table 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 purposes of ranking. Table 6 describes these two indexes built over the `Keywords` table.

2.4 Keyword Ranking: Okapi BM25

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. There many well studied state-of-art scoring functions that work well for a definite setting and it would be difficult to claim a generic ranking function that is optimal. For our data, we select a variant of the Okapi BM25 [2] scoring function with parameters k=1.2 and b=2.0, which works well for the

Index Name	**Attributes**
`Keywords_Entity_IDX`	`(Entity_ID, Term, Score)`
`Keywords_Term_IDX`	`(Term, Entity_ID, Score)`

Table 6. Indexes built over the `Keywords` table

fulltext searches involved in our query processing. The exact BM25 variant we used for ranking an entity e by a string of keywords S in an `FTContains` operator is given by the following formula:

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

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

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) $len(e)$ 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 [2] for tuning BM25 on earlier INEX settings).

2.5 Entity Ranking

We realize that ranking entities in context of a given structured query becomes a difficult challenge. As a simple approach, we consider it to be reasonable to carry over the aggregated scores of the entities obtained from fulltext searches that are performed for associated fulltext constraints on the entity. However, there may arise a special case where in a SPARQL-FT query, entities do not have fulltext constraints. In such a case all the entities that satisfy the semantic structure defined by the query triple (or triples) pattern, become candidates to either become the final answer, or for further processing. In such cases, we give a default score of 1 to these entities. A more elaborate discussion is presented in [1].

3 SPAR-Key Supremacy

In this section, we introduce the first variant of our query engine, called **SPAR-Key Supremacy**. In addition to the basic ranking methodology discussed in the last section, this system implements a SPARQL-FT-to-SQL translator that: 1) uses SQL Joins over simple "AND conditions" in conjunctive query processing, 2) materializes temporary tables and sub queries to represent intermediate results to improve efficiency, 3) uses a simple selectivity estimation technique to decide join order of temporary tables, and 4) includes additional query optimizations to force the optimizer to follow the decided join order.

3.1 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. 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 optimizer 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.

3.2 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 an Inverse Document Frequency (IDF) index containing the most frequent terms that occur in the collection. This index follows a very simple layout of a key-value pair, where a key is a term and the value is it's IDF. A frequent term will have lower IDF and hence 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.

In our case, since we use the Oracle relational-DBMS as our back-end, it provides a functionality by which the joining of intermediate results can be enforced on the Oracle optimizer. This is achieved by adding *optimizer hints* to the queries. Of the made available hints by Oracle to guide the query optimizer, we identify that the `Ordered` hint could force the joining of tables in the determined order while preserving the logics of the join condition in the original query. Thus our query translator automatically adds this hint in the translated SQL queries.

3.3 SPAR-Key Supremacy: The Rewriting Algorithm

We can now develop an overall rewriting algorithm by putting together all the afore mentioned steps.

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. Create temporary $Keys_i$ tables for each fulltext condition: these contain the results of the OUTER joins over the `Keywords` table constrained by the terms.
4. Create temporary Tab_i tables for each triplet pattern. These contain the results of the INNER joins over the `DBpediaCore` table which are additionally joined with $Keys_i$ temporary tables for each `FTContains` fulltext condition in the query.
5. Assign a default score of 1 to all triples without any fulltext condition.

6. Formulate the main select query that combines the Tab_i temporary tables via an INNER joins; the join logic is based on the joins given in the original SPARQL query.
7. Finally, drop the temporary tables $Keys_i$ and Tab_i.

4 SPAR-Key Ultimatum

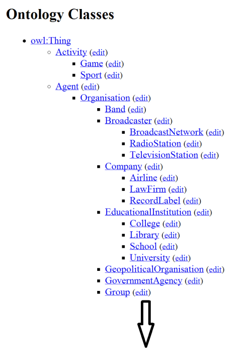
In this section we introduce the second and third variant of the query engine, called the **SPAR-Key Ultimatum No Phrases** and the **SPAR-Key Ultimatum Phrases**. In fact, these are implementation-wise the same with only one additional component called *Phrase Search* (discussed in later in Section 4.4) activated in the latter. Thus we describe them together in this section.

In addition to the basic ranking and efficiency improvement methodologies discussed so far, this system implements a SPARQL-FT-to-SQL translator that: 1) uses *Class Selection* to prune out false positives, 2) exploits the structure of a give query to identify additional constraints, 3) incorporates *URI Search* as a basic entity disambiguation tool, and 4) extracts noun phrases and performs proximity search to improve ranking.

4.1 Class Selection

We note 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 3.3). 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 1 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>`.

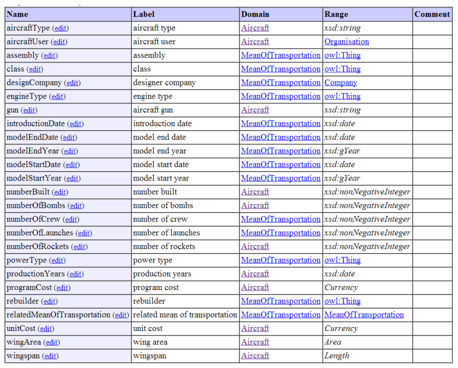
The property definition in a class specifies a signature. To understand this, let us consider an example shown in Figure 2. This example shows the properties defined in the `<Aircraft>` class with their signature. In the above example, one can see a property `<aircraftUser>` defined with a *Domain*, `<Aircraft>` and *Range* `<Organization>` 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 `<Organization>` 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.



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)

Fig. 1. Snapshot of DBpedia class hierarchy [*http://mappings.dbpedia.org*]



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

Fig. 2. Snapshot of DBpedia class hierarchy [*http://mappings.dbpedia.org*]

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 7 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.

4.2 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 [6].

- **Chain pattern** is where the *Object* of the first triple pattern is the *Subject* of the next triple pattern, again with given *Predicates*. Figure 3 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 4 shows a generic illustration of a star pattern.

The query pattern can be used to derive classes of entities even though in a triple pattern a *Predicate* is not specified by a literal. To see this let us consider an example query as shown in Figure 4. By analyzing the *Predicate* of the first triple pattern `(<museum> <located> ?a.)`, i.e., `<located>`, we can mark the *Object* `(?a)`

Index Name	**Attributes**
`PredicateDomainIDX`	`(Predicate, Domain)`
`PredicateRangeIDX`	`(Predicate, Range)`

Table 7. Schema of the `PredicateDomainIDX` and `PredicateRangeIDX` indexes

Index Name	**Attributes**
URI_IDX	(Entity_ID, Term)

Table 8. Schema of the URI index

of to be class <City> which is specified by the *Range* of the *Predicate* signature ; and then by analyzing the second *Predicate* <partOf>, we can mark the *Subject* (?c) of the second triple pattern (?b <partOf> ?a.) to be of class <Country>. Similar analogies can be drawn for chain query patterns also.

4.3 Search on the URI Index

Document titles are considered as an important feature by IR systems among other features like content, context, etc. These document titles tend to summarize the major context of the articles. Following this notion, 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 idea, 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 8 shows the schema of the URI index used by SPAR-Key Ultimatum.

4.4 Phrase Search

We observe that the in most of the Jeopardy-style NL questions, the occurrence of the clues are in form of phrases, for example *"King of Pop"*, *"Don't be Evil"*, etc. In addition, we also observe that other clues that mostly describe an entity (in the associated fulltext conditions), contain nouns that most likely to occur in close proximity in the textual data corresponding to the entities, for example, *"Christian church founder"*, *"video sharing"*, etc. Thus we identify these noun phrases as additional features to improve the ranking of the entities. The main idea is to perform a basic Natural Language Processing on the Jeopardy-style questions to extract all the noun phrases and

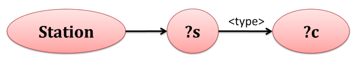


Fig. 3. Chain query structure

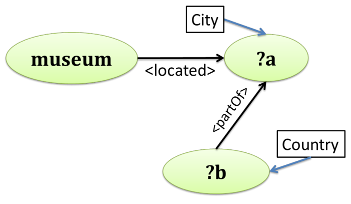


Fig. 4. Star query structure

then boost the scores of the entities that contain these noun phrases. To perform the noun phrase extraction, the SPAR-Key Ultimatum uses the Stanford NLP Core [3] as a black box and then automatically formulates a proximity keyword query by setting the slope between the keywords as the average length of a sentence in Wikipedia. It then performs an OUTER join of the results from the proximity search to the the results of the fulltext search (by assuming independence between keywords), boosting the scores of the entities that are returned by both.

4.5 SPAR-Key Ultimatum: The Rewriting Algorithm

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. Analyze 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:
   $classSet = \{\{variable_1, class_1\}, \{variable_2, class_2\}, \{variable_3, class_3\}, ...\}$.
4. Exploit the query structure to find most selective class for an entity place-holder 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 addition'ally 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 Evaluations

In this section 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 with query processing techniques proposed in this paper.

5.1 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. 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.

5.2 Measures

We use standard TREC metrics to measure the performance of all the runs. To compare our SPAR-Key engine variants we use the **Mean-Average-Interpolated-Precision (MAiP), Precision at K (P@K)**, specifically P@5, P@10, P@15 and show plotting of the **Average-interpolated-Precision values (AiP) at 11 standard points**. We also perform a QA style evaluation with the **Mean-Reciprocal-Rank (MRR) and Normalized-Discounted-Cumulative-Gain (NDCG)** specifically, NDCG@5, NDCG@10 and NDCG @15. For further details on the metrics used to evaluate the run please refer to [8].

5.3 Experimental Runs

In this section we analyze our query engine based on the official evaluation results presented by the INEX. Table 9 shows the official INEX results over the Jeopardy topics. To generate the results, top-20 results from each run were pooled and assessed by crowd sourcing through the Crowdflower platform [3]. For further details on the evaluation procedure please refer to the Overview paper of the INEX 2013 LOD track. We note that we were the only group that participated in the Jeopardy task this year and hence we cannot compare our results to any other competitor. However, we compare our own variants of the query processor based on the three runs (one from each variant) submitted to the INEX.

We identify four classes of queries in the benchmark that: 1) target single entity as the only correct answer, 2) target combination of entities as the only correct answer, 3) target list of entities as correct answer and 4) target list of combination of entities as correct answer. In this evaluation we present the combined results of our engine over all the queries.

From the evaluation results we observe that by activating the phrase search component (described in Section 4.4) we obtain the best results in terms of MRR and NDCG values. This clearly supports our choice of recognizing the noun phrases as valuable features for the query processing. In addition, we find that the simpler Supremacy variant gives better performance than the Ultimatum-No phrase (though not significantly better) as the URI Search (described in Section 4.3) which is activated in the Ultimatum-No phrase becomes an overkill for most of the queries.

[3] https://crowdflower.com/

	MPIUltimatum_Phrase	MPIUltimatum_NoPhrase	MPISupremacy
MAiP	0.7491	0.701	0.719
MRR	0.7671	0.7358	0.7539
NDCG@5	0.7723	0.7307	0.7393
NDCG@10	0.7864	0.7347	0.7598
NDCG@15	0.7968	0.7484	0.7728
AiP@1%	0.7804	0.7411	0.7669
AiP@10%	0.7804	0.7411	0.7669
AiP@20%	0.7804	0.731	0.7653
AiP@30%	0.7804	0.731	0.763
AiP@40%	0.7772	0.7255	0.7468
AiP@50%	0.7737	0.7232	0.7417
AiP@60%	0.7337	0.6803	0.6991
AiP@70%	0.7245	0.6771	0.6952
AiP@80%	0.7223	0.6747	0.685
AiP@90%	0.7208	0.673	0.6817
AiP@100%	0.7208	0.6662	0.6694

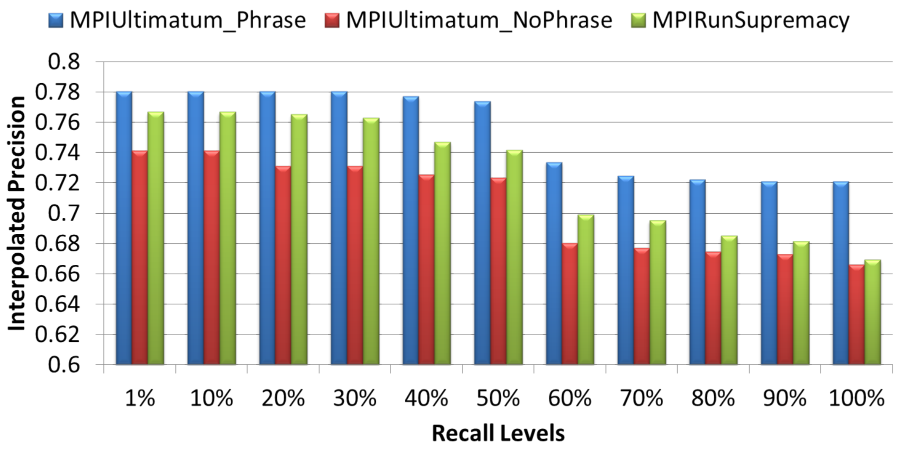


Table 9. Jeopardy Task retrieval results.

6 Conclusion

We presented an approach for storing structured RDF data and unstructured data in relational database. We also presented the necessary indices required to efficiently process queries over the relational schema. Our approach converts a SPARQL query with fulltext conditions into unions of conjunctive SQL queries by materializing temporary tables. These temporary tables store intermediate results from inner or outer joins over our relations, based on given conditions in the query. We also presented a simple yet effective way to rank entities by translating scores from keywords. In addition, we showed three variants of the query processor, each following different query processing strategies by recognizing a different set of features and score boosting to obtain the entity ranking. Finally we compared the variants based on the official evaluation released by the INEX and underline some of the key advantages and disadvantages of each query processing strategies. As a future work we would like to focus on the efficiency of the query processing and shifting the storage model from relational-DBMS to file system. Also we are keen in designing an automatic translator for translating a Jeopardy-style NL question into a SPARQL-FT query which can be then processed by our current system.

References

1. Arunav Mishra and Sairam Gurajada and Martin Theobald. Running SPARQL-Fulltext Queries Inside a Relational DBMS. In *CLEF (Online Working Notes/Labs/Workshop)*, 2012.
2. Clarke, Charles L. A. Controlling overlap in content-oriented XML retrieval. In *SIGIR*, 2005.
3. M.-C. De Marneffe and C. D. Manning. Stanford typed dependencies manual. *URL http://nlp. stanford. edu/software/dependencies_manual. pdf*, 2008.
4. Luo, Yongming and Picalausa, François and Fletcher, George H. L. and Hidders, Jan and Vansummeren, Stijn. Storing and Indexing Massive RDF Datasets Semantic Search over the Web. In *Semantic Search over the Web*, Data-Centric Systems and Applications. 2012.
5. Martin Theobald and Ablimit Aji and Ralf Schenkel. TopX 2.0 at the INEX 2009 Ad-Hoc and Efficiency Tracks. In *INEX*, 2009.
6. Neumann, Thomas and Weikum, Gerhard. RDF-3X: a RISC-style engine for RDF. *Proc. VLDB Endow.*, 1, 2008.
7. Qiuyue Wang and Jaap Kamps and Georgina Ramirez Camps and Maarten Marx and Anne Schuth and Martin Theobald and Sairam Gurajada and Arunav Mishra. Overview of the INEX 2012 Linked Data Track. In *CLEF (Online Working Notes/Labs/Workshop)*, 2012.
8. Qiuyue Wang and Jaap Kamps and Georgina Ramirez Camps and Maarten Marx and Anne Schuth and Martin Theobald and Sairam Gurajada and Arunav Mishra. Overview of the INEX 2013 Linked Data Track. In *CLEF (Online Working Notes/Labs/Workshop)*, 2013.