# Running SPARQL-Fulltext Queries Inside a Relational DBMS

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

Max Planck Institute for Informatics, Saarbrücken, Germany

**Abstract.** We describe the indexing, ranking, and query processing techniques we implemented in order to process a new kind of SPARQL-fulltext queries that were provided in the context of the INEX 2012 Jeopardy task.

## 1 Introduction

The INEX 2012 Linked Data track provides a new data collection that aims to combine the benefits of both text-oriented and structured retrievals settings in one unified data collection. For the rapid development of a new query engine that could handle this particular combination of XML markup and RDF-style resource/property-pairs, we decided to opt for a relational database system 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. To process the 90 queries of the benchmark, we keep two main index structures, one of which is based on a recent dump of DBpedia core triples, and another one, which is based on keywords extracted from the INEX Wikipedia-LOD collection. Additionally, our engine comes with a rewriting layer that translates the SPARQL-based query patterns into SQL queries, thus formulating joins over both the DBpedia triples and the keywords extracted from the XML articles.

## 2 Document Collection and Queries

### 2.1 Document Collection

Each XML document in the Wikipedia-LOD collection combines structured and unstructured information about a Wikipedia entity (or so-called "resource") in one common XML format. The structured part ("properties") of the document represents factual knowledge, which was obtained from querying DBpedia and YAGO for facts containing this entity as either their subject or object together with the property itself, while the semistructured part (WikiText) is XML-ified content that was obtained from the Wikipedia article describing the entity. Specifically, the main components of each XML document in the collection can be divided into the following blocks:

1. **Meta Data:** Describes the meta information of the document like title, author, etc. and provides a unique Wikipedia entity ID.

***Query1: What is a famous Indian Cuisine dish that mainly contains rice, dhal, vegetables, roti and papad***

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

***Query2: Which mountain range is bordered by another mountain range and is a popular sightseeing and sports destination?***

| | |
|---|---|
| `SELECT ?p WHERE { ?m <http://dbpedia.org/ontology/border> ?p . ?p <http://www.w3.org/1999/02/22-rdf-syntax-ns#type> <http://dbpedia.org/ontology/MountainRange> . FILTER FTContains(?p, "popular sightseeing and sports destination") . }` | `http://dbpedia.org/resource/Alps` |

**Table 1.** Example benchmark queries in SPARQL-fulltext format

2. **WikiText:** This portion of the document includes text from the Wikipedia article in well-formed XML syntax. The Wikipedia were translated from the common MediaWiki [1] format. Additionally XML tags for all infobox attributes (and corresponding values) were included to replace all Wiki markup by proper XML tags. The inter-Wiki links which point to the other Wikipedia entities are extended to include the links to both the YAGO and DBpedia resources of the Wikipedia target pages. Each link has three sub-links: `wiki-link`, `yago-link`, and `dbpedia-link`. The `wiki-link` attribute is a regular hyperlink to the target Wikipedia article, while the `yago-link` and `dbpedia- link` attributes contain pointers to the respective sources in the RDF collections of DBpedia and YAGO2.
3. **Properties:** Finally, each document at the end includes a list of all the properties from the DBpedia and YAGO facts about the entity.

### 2.2 Benchmark Queries

The translated queries are syntactically conforming to the SPARQL standard, with an additional `FTContains` operator that provides the flexibility to add keyword constraints to the query. The new `FTContains` operator takes two parameters, namely the variable name in the SPARQL component and a set of associated keywords. Table 1 shows two such natural-language queries and their SPARQL translations containing `FTContains` operators for the keyword components. For instance, in the SPARQL translation of Query 1, we have the original SPARQL component as `?s ?p <http://dbpedia.org/resource/Category:Indian_cuisine>` and the keyword component as {`rice dhal vegetables roti papad`}. In addition, the subject "?s" is bound by the keyword component as specified in the `FTContains` function.

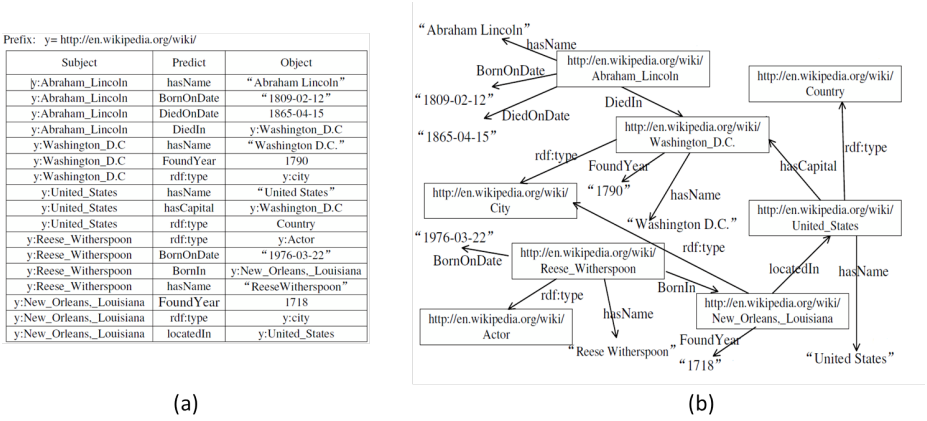


**Fig. 1.** Example showing an RDF graph and the corresponding triplet representation in a relational database *(picture taken from [14])*

## 3 Document Parsing and Index Creation

We employed 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 others, usually also an infobox with additional semistructured 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 [11], which includes Porter stemming, stopword removal and BM25 ranking, but stores the resulting inverted lists for keywords into a relational database instead of TopX's proprietary index structures.

## 4 Data Model for the Document Collection

In this section we describe the storage backend for both the structured and unstructured data parts of our document collection. Our query engine employs two relational tables to import the core tables of DBpedia and the keywords extracted from the Wikipedia LOD collection [2] in one unified database schema. Thus, the SPARQL queries with fulltext filter conditions of the above form can directly be rewritten to SQL queries with various join conditions over these two tables.

### 4.1 Storing RDF Data in a Single Relational Table

An RDF data collection can be defined as a collection of triples (a subject or resource, a predicate or property, and an object or value). In this paper, we refer to the RDF data

| Column | Type |
|---|---|
| N3ID | NUMBER |
| Subject | VARCHAR2(1024) |
| Predicate | VARCHAR2(1024) |
| Object | VARCHAR2(1024) |

**Table 2.** Schema of the DBpediaCore table

| Index Name | Attributes |
|---|---|
| DBpediaIDX_Obj | (Object,Subject,Predicate) |
| DBpediaIDX_Sub | (Subject,Predicate,Object) |
| DBpediaIDX_Prd | (Predicate,Object,Subject) |

**Table 3.** Indices built over the DBpediaCore table

as a collection of SPO triples, each containing exactly one such subject (S), predicate (P), and object (O).

As our storage back-end, we use a relational database (in our case Oracle 11g), to store the RDF data we imported from the core dump of DBpedia v3.7 (see http://downloads.dbpedia.org/3.7/en/).A SPARQL query then conceptually is transformed into a sub-graph matching problem over this large RDF graph. The RDF data model (a collection of triples) can be viewed as a directed, labeled multi-graph (RDF graph) where every vertex corresponds to a subject or object, and each directed edge from a subject to an object corresponds to a predicate. There may be multiple edges directed from a subject towards an object. Thus an RDF graph becomes a multi-graph.

In our query engine, RDF data is treated as a large list of triples and stored in a relational database. Figure 1(b) shows a fragment of an RDF graph constructed over Wikipedia, and Figure 1(a) shows a corresponding relational table representation in the database. Keeping this in mind, and maintaining the simplicity of the model, we use one relational table to store all SPO triples. In our system, we refer to this table as the DBpediaCore table, and its schema is described in Table 2. This route is pursued similarly by many so-called triplet-stores like Jena [12], Sesame [4] and Oracle [6] and RDF-3X [8]. Though there are other advanced and more complex approaches, like vertical partitioning or the exploitation of property tables, our goal was satisfied by this relatively simple idea.

### 4.2 Indices on the RDF Table

Representing RDF data as a relational table opens up for us all kinds of optimization techniques used by the database community. One such technique is to build indices over the DBpediaCore table to improve the data retrieval operations on it. Since we translate a SPARQL query with fulltext conditions into conjunctive SQL queries (described in the following section), we employ a large amount of self joins over this table. These self joins could occur on any of the three columns described in Table 2. Thus we build three multi-attribute indices, using each of the SPO columns of the table as key and the remaining two attributes as depending data values, to accelerate the lookup times over this table for the case when each of the attributes is provided as a constant by the SPARQL query. Table 3 describes these three indices built over the DBpediaCore table.

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

**Table 4.** Table schema of the Keywords table

| Index Name | Attributes |
|---|---|
| Keywords_Entity_IDX | (Entity_ID, Term, Score) |
| Keywords_Term_IDX | (Term, Entity_ID, Score) |

**Table 5.** Indices built over the Keywords table

### 4.3 Storing Keywords in a Single Relational Table

As per traditional IR, a fulltext search allows for identifying documents that satisfy a keyword query, and optionally sorting the matching documents by their relevance to the query. 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 data collection. In our engine, we store the unstructured text contents of all the documents in the collection as another table in the relational database. This table essentially contains three columns, relating a keyword (or term) with the documents in which it occurs and the similarity scores calculated for this particular term and document based on the underlying scoring model.

We define a term as a sequence of characters that occur grouped together in some document and thus yield a useful semantic unit for processing. To obtain this set of terms from the fulltext content of a Wikipedia article, we use a variant of the TopX indexer [11], which applies a standard white-space tokenizer, Porter stemming, and stopword removal to the unstructured text inputs. For this purpose, we treat all CDATA sections and attribute values of the Wikipedia-LOD articles as one flat collection of text; that is we treat the XML collection as an unstructured set of Wikipedia text documents. We use a default BM25 scoring model (using $k = 2.0$ and $b = 0.75$) to calculate the relevance score of a term in a document. In our approach, we create a Keywords table to store all the terms occurring in the unstructured Wikipedia fulltext collection. The schema of this table is shown in Table 4. The Entity_ID column essentially stores the *Uniform Resource Identifier* (URI) of the DBpedia entities. It may be noted that every document in our collection also corresponds to a DBpedia entity. So we prefer to use the RDF prefix defined by DBpedia to represent an entity, which is ⟨http://dbpedia.org/resource/entity⟩. To obtain a mapping from a DBpedia entity to the Entity_ID's from a Wikipedia page, we maintain a hashmap that maps every Wikipedia page to its corresponding DBpedia entity URI. Thus every statement of the Keywords table represents a term that is mapped to an entity in DBpedia together with its content similarity score to the entity's Wikipedia page.

### 4.4 Indices on the Keywords Table

We can easily imagine that the Keywords table would be very large (containing more than a billion entires), considering the number of terms extracted from almost the entire Wikipedia encyclopedia. The Keywords table basically maps every term occurring in Wikipedia to a DBpedia entity. Taking the size of this table into consideration, data

retrieval operations on the table becomes costly and inefficient. Also processing conjunctive queries over multiple self joins becomes infeasible unless correct indices are built over the table. So we build two more multi-attribute indices over this table as shown in Table 5.

# 5 Scoring

## 5.1 Keywords Scoring and retrieval

We use the TopX 2.0 indexer to create per-term inverted lists from the plain (unstructured) text content of the Wikipedia documents, which each corresponds to an entity in DBpedia. 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 [7] for tuning BM25 on earlier INEX settings).

## 5.2 Translating the Keyword-Scores to SPO Triples

Recently there has been a lot of work on identifying appropriate entity scoring and ranking models. Many techniques use language models [13, 9, 10] while other approaches try to adopt more complex measures form IR. Ranking for structured queries have been intensively investigated for XML [3], and, to a small extent, also for restricted forms of SQL queries [5]. While some of these approaches carry over to large RDF collections and expressive SPARQL queries as discussed earlier, the LOD track makes a significant simplifying assumption: since every DBpedia or YAGO 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 to the RDF entities from a keyword-based search on the fulltext part of the Wikipedia LOD collection. As discussed earlier, every entity in our collection is essentially a document containing all the DBpedia and YAGO properties about this entity

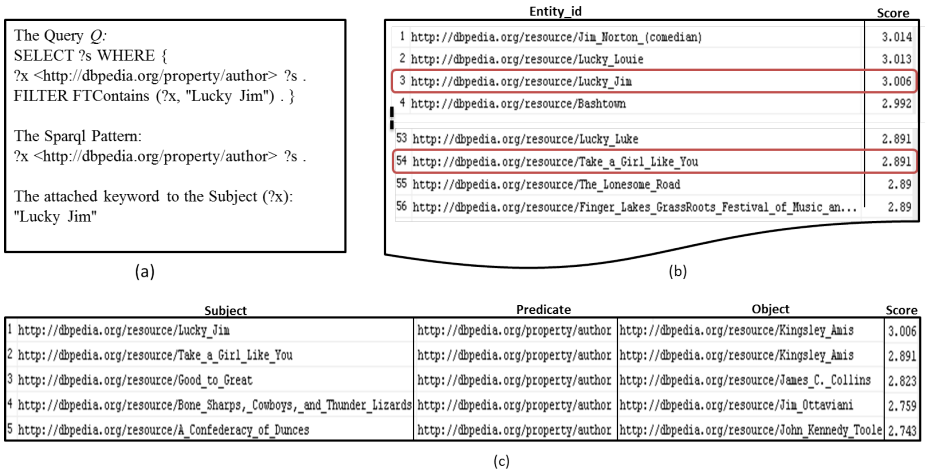


**Fig. 2.** Example showing the translation of keyword scores to the entity-relationship graph

together with the fulltext content from the corresponding Wikipedia page. Thus we perform a keyword search on this fulltext content by using the fulltext condition specified by the `FTContains` operator (as discussed earlier). From this search, we get a ranked entity list containing the candidates for the answer to the given query. This is can be best explained by an example as shown in Figure 2 .

Figure 2(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 2(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 2(a) and the final top-$k$ entities are retrieved from the entities which qualify for the graph pattern. Figure 2(c) shows the retrieved results after searched for the pattern in the graph.

## 6 Rewriting SPARQL-Fulltext Queries to SQL

### 6.1 Basic Idea of a Conjunctive Query

Conjunctive queries have high practical relevance because they cover a large part of queries issued on relational databases and RDF stores. That is, many SQL and SPARQL queries can be written as conjunctive queries. In our scenario, we are required to generate an appropriate translation of a given SPARQL query with multiple fulltext conditions into an SQL query. Our system-generated queries are essentially conjunctive queries over multiple instances of the `DBpediaCore` and `Keywords` tables (as described earlier).

To capture the idea behind translating a given SPARQL query with fulltext conditions into a conjunctive SQL query, let us take an example query $Q$ taken from the Jeopardy benchmark as shown in Table 6. $Q$ contains three triple patterns $T1$, $T2$ and $T3$, and two fulltext conditions $K1$ and $K2$. Each $K_i$ contains a variable occurring in a triple pattern and is bound to a string by an `FTContains` operator. To begin translation, firstly, 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 `Terms` 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 $?sub$ $?o$ $?s$, instance $t_2$ represents the triple $?x$ $?r$ $?s$, and instance $t_3$ represents the triple $?sub$ `rdf:type` ⟨`http://dbpedia.org/ontology/Place`⟩. These instances $t_i$ are further restricted on their `Subject` or `Object` by the `Entity_ID` of the $k_j$ instance of the `Keywords` table as specified by the fulltext condition. Finally, the instances $t_i$ are restricted by each other on `Subject`, `Predicate` or `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 7.

## 7 Materialization SQL Joins, Temporary Tables, and Join Orders

In the previous section, we presented a translation method for any given (conjunctive) SPARQL query with fulltext conditions into a conjunctive SQL query. But, there are a few problems with such a direct translation. Firstly, this form of translation does not handle empty results returned from any intermediate join in the desired way. For example, lets say we encounter with a very unlikely keyword term which is missing in the `Keywords` table. Then an empty result will be returned for that instance, and, since we issue a conjunctive query with all AND conditions, the overall result of the query will also be empty. Secondly, this type of query with many (self-)joins is inefficient and difficult to optimize. During query processing, we rely on the Oracle optimizer for selecting an appropriate query plan. It becomes difficult for the optimizer to choose the best plan due to the redundancy of data in the `DBpediaCore` table, i.e., due to multiple occurrences of an entity as a `Subject` or `Object` and an unknown amount of occurrences of a term in the `Keywords` table. Due to apparently insufficient statistics on these large tables (although we had analyzed the tables and enabled this feature in

| **A manually translated query in SPARQL syntax:** |
|---|
| 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 6.** A given SPARQL-fulltext query of the INEX 2012 Jeopardy task

| **The automatically converted conjunctive SQL query:** |
|---|
| SELECT DISTINCT t1.SUBJECT AS sub FROM dbpediacore t2, dbpediacore t1, dbpediacore t3, keywords k40, keywords k41, keywords k42, keywords k3 |
| WHERE |
| t2.OBJECT = t1.OBJECT |
| AND t1.SUBJECT = t3.SUBJECT |
| AND t3.PREDICATE = 'http://www.w3.org/1999/02/22-rdf-syntax-ns#type' |
| AND t3.OBJECT = 'http://dbpedia.org/ontology/Place' |
| AND k0.TERM = 'place' AND k1.TERM = 'island' |
| AND k2.TERM = 'countri' |
| AND k3.TERM = 'canarian' |
| AND t1.SUBJECT = k0.ENTITY_ID |
| AND t1.SUBJECT = k1.ENTITY_ID |
| AND t1.SUBJECT = k2.ENTITY_ID |
| AND t2.SUBJECT = k3.ENTITY_ID |

**Table 7.** SQL translation of the SPARQL query of Table 6

the optimizer), we found the Oracle optimizer to often choose a bad query plan, which initially resulted in individual SQL queries that did not even finish after several days.

### 7.1 SQL Joins

Most join queries contain at least one join condition, either in the FROM (for outer joins) or in the WHERE clause. The join condition compares two columns, each from a different table. To execute a join, the database system combines pairs of rows, each containing one row from each table, for which the join condition evaluates to true. The columns in the join conditions need not also appear in the select list.

To execute a join of three or more tables, Oracle first joins two of the tables based on the join conditions comparing their columns and then joins the result to another table based on the join conditions containing columns of the previously joined tables and the new table. Oracle continues this process until all tables are joined into the result. The optimizer determines the order in which Oracle joins tables based on the join conditions, indexes on the tables, and any available statistics for the tables.

**FULL OUTER JOIN** A FULL OUTER JOIN does not require each record in the two joined tables to have a matching record. The joined table retains each record even if no other matching record exists. Where records in the FULL OUTER JOIN'ed tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those records that do match, a single row will be produced in the result set (containing fields populated from both tables). This join can be used to solve the first problem mentioned above. All instances of the `Keywords` table representing a fulltext condition $K_i$ can undergo a FULL OUTER JOIN on the `Entity_ID` attribute. Thus we will have results from the keywords table even if one of the instance matches any tuples or has a NULL as a value.

```
CREATE TABLE KEYS0 AS  SELECT/*+ORDERED*/ * FROM
( SELECT/*+ORDERED*/ DISTINCT ENTITY_ID , MAX(SCOREss) AS SCORE FROM
(  SELECT DISTINCT K1.ENTITY_ID  AS ENTITY_ID , (NVL(k1.SCORES,0)+1  + NVL(k2.SCORES,0)+1 +
NVL(k3.SCORES,0)+1 ) AS SCORESS FROM
(SELECT DISTINCT ENTITY_ID  , SCORE  AS SCORES  FROM KEYWORDS WHERE TERM='island') K1
FULL OUTER JOIN
(SELECT DISTINCT ENTITY_ID  , SCORE  AS SCORES  FROM KEYWORDS WHERE TERM='countri') K2
ON  k1.ENTITY_ID = k2.ENTITY_ID
FULL OUTER JOIN
(SELECT DISTINCT ENTITY_ID  , SCORE  AS SCORES  FROM KEYWORDS WHERE TERM='place') K3
ON  k1.ENTITY_ID = k3.ENTITY_ID
ORDER BY SCORESS DESC)
GROUP BY ENTITY_ID ORDER BY SCORE DESC )
```

```
CREATE TABLE KEYS1 AS  SELECT/*+ORDERED*/ * FROM
( SELECT/*+ORDERED*/ DISTINCT ENTITY_ID , MAX(SCOREss) AS SCORE FROM
(  SELECT DISTINCT K1.ENTITY_ID  AS ENTITY_ID , (NVL(k1.SCORES,0)+1 ) AS SCORESS FROM
(SELECT DISTINCT ENTITY_ID  , SCORE  AS SCORES  FROM KEYWORDS WHERE TERM='canarian')
K1
ORDER BY SCORESS DESC)GROUP BY ENTITY_ID ORDER BY SCORE DESC )
```

**Fig. 3.** Queries to create the `Keys` temporary tables

**INNER JOIN** An INNER JOIN is the most common join operation used in applications and can be regarded as the default join type. INNER JOIN creates a new result table by combining column values of two tables (A and B) based upon the join predicate. The query compares each row of A with each row of B to find all pairs of rows which satisfy the join predicate. When the join predicate is satisfied, column values for each matched pair of rows of A and B are combined into a result row. The result of the join can be defined as the outcome of first taking the Cartesian product (or cross join) of all records in the tables (combining every record in table A with every record in table B). It returns all records which satisfy the join predicate. This join returns the same result as the "equality" join as described in the previous section but gives more flexibility to the optimizer to select different types of joins like hash joins, merge joins, or nested loop joins. We can replace all the equality join with Oracle's INNER JOIN. In some queries this trick shows considerable improvement in query processing time by Oracle's query engine.

### 7.2 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 turns out that—at least in our setting—it often chooses an inefficient plan. For example, in a big conjunctive query, the optimizer often chose to join instances `DBpediaCore` tables before restricting the relevant entities with the given conditions. These types of query plans proved to be highly expensive. Thus, to prevent the optimizer from taking such inappropriate decisions, we materialize temporary tables by separately joining the `Keywords` table instances and

```
CREATE TABLE tab3 as select subject , predicate , object  , ( NVL(KEYS0.score,0) ) AS 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.score,0) ) AS 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.score,0) ) AS SCORE  from
(SELECT * FROM  dbpediacore3  t2 )temp
INNER JOIN
KEYS1
ON temp.SUBJECT=keys1.entity_id
```

**Fig. 4.** Queries to create the TAB temporary tables

the `DBpediaCore` table instances. This acts as a strong support for the optimizer to select better query plans for smaller intermediate queries and store their results into temporary tables which are later joined together to retrieve the final result.

### 7.3 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 an IDF index containing the most frequent terms that occur in the collection. This index has a very simple and intuitive schema `Features(Term, IDF)`. The first column represents a term and the second column represents its Inverse Document Frequency (IDF). It can be intuitively be seen that a frequent term will have lower IDF and a select query will result in bigger intermediate joins. At the same time, if a term is absent in the feature index, then it can be assumed to be infrequent and thus have a high IDF value. Every instance of the `Keywords` table

```
SELECT /*+Ordered*/  sub , MAX(SCORE) AS SCORE_MAX FROM
 (SELECT /*+ORDERED*/ DISTINCT tab3.SUBJECT AS sub , ( NVL(tab3.score ,0 )+1
+  NVL(tab1.score ,0 )+1 +  NVL(tab2.score ,0 )+1 +  0 ) AS score FROM
tab3 , tab1 , tab2
WHERE tab1.OBJECT = tab2.OBJECT
AND tab2.SUBJECT = tab3.SUBJECT
) GROUP BY sub ORDER BY SCORE_MAX DESC
```

**Fig. 5.** The final SELECT query to obtain the answer

```
drop table keys0
drop table keys1
drop table tab3
drop table tab1
drop table tab2
```

**Fig. 6.** Queries to DROP the temporary tables

can now be joined in increasing order of the IDF values of their respective term, thus ensuring the smaller tables to be joined first. This order of joining can be enforced on the Oracle optimizer by adding so-called optimizer hints to the queries.

A hint is a code snippet that is embedded into an SQL statement to suggest to Oracle how the statement should be executed. There are many hints provided to assist the optimizer. In our case, we found 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 the appropriate order in the FROM clause of the translated SQL query.

## 8 The Rewriting Algorithm

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

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 join over the `Keywords` table instances constrained by the terms. This is shown in Figure 3.
4. 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 $Keys_i$ temporary tables for each `FTContains` fulltext condition in the query. This is shown in Figure 4.
5. Assign a default score of 1 to all triples in absence of a fulltext condition: in absence of a fulltext condition on any triplet pattern, a default score of 1 is assigned to all the triples as a constant score for each triplet condition (as discussed earlier).
6. Final query: the main select query combines the $Tab_i$ temporary tables via an inner join; the join logic is based on the joins given in the original SPARQL query. This is shown in Figure 5.
7. Finally, drop the temporary tables $Keys_i$ and $Tab_i$. This is shown in Figure 6.

## 9 INEX Results

Since a detailed evaluation of the run submissions was not available at the time this paper was submitted, we defer a discussion of the results until to the INEX workshop at the CLEF conference in September.

## 10 Conclusions

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 this 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. Finally, we showed the advantages of predeciding the join orders of tables and techniques to enforce them in the Oracle optimizer

## References


1. Media Wiki. http://en.wikipedia.org/wiki/MediaWiki.
2. Overview of the INEX 2012 Linked Data Track.
3. S. Amer-Yahia and M. Lalmas. XML search: languages, INEX and scoring. *SIGMOD Record*, 35, 2006.
4. J. Broekstra, A. Kampman, and F. van Harmelen. Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema. In *LNCS*, volume 2342, Sardinia, Italy, 2002.
5. S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic information retrieval approach for ranking of database query results. *ACM Trans. Database Syst.*, 31, 2006.
6. E. I. Chong, S. Das, G. Eadon, and J. Srinivasan. An efficient SQL-based RDF querying scheme. In *VLDB '05*, 2005.
7. C. L. A. Clarke. Controlling overlap in content-oriented XML retrieval. In *SIGIR*, 2005.
8. T. Neumann and G. Weikum. The RDF-3X engine for scalable management of RDF data. *The VLDB Journal*, 19(1), 2010.
9. D. Petkova and W. B. Croft. Hierarchical Language Models for Expert Finding in Enterprise Corpora. In *ICTAI '06*, 2006.
10. P. Serdyukov and D. Hiemstra. Modeling Documents as Mixtures of Persons for Expert Finding. In *LNCS*, volume 4956, 2008.
11. M. Theobald, A. Aji, and R. Schenkel. TopX 2.0 at the INEX 2009 Ad-Hoc and Efficiency Tracks. In *INEX*, 2009.
12. K. Wilkinson, C. Sayers, H. Kuno, and D. Reynolds. Efficient RDF Storage and Retrieval in Jena2. In *Proc. First International Workshop on Semantic Web and Databases*, 2003.
13. C. Yang, Y. Cao, Z. Nie, J. Zhou, and J.-R. Wen. Closing the Loop in Webpage Understanding. *IEEE Transactions on Knowledge and Data Engineering*, 22, 2010.
14. L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gStore: answering SPARQL queries via subgraph matching. *PVLDB*, 4(8), 2011.