Data Exploration on Large Amount of Relational Data through Keyword Queries

Domenico Beneventano, Francesco Guerra
University of Modena and Reggio Emilia
Modena, IT
Email: first_name.last_name@unimore.it

Yannis Velegrakis
University of Trento
Trento, IT
Email: velgias@disi.unitn.eu

***Abstract*—The paper describes a new approach for querying relational databases through keyword search by exploting Information Retrieval (IR) techniques. When users do not know the structures and the content, keyword search becomes the only efficient and effective solution for allowing people exploring the data of a relational database.**

The approach is based on a unified view of the database relations (performed through the *full disjunction* operator), where its composing tuples will be considered as documents to be indexed and searched by means of an IR search engine. Moreover, as it happens in relational database, the system can merge the data stored in different documents for providing a complete answer to the user. In particular, two documents can be joined because either their tuples in the original database share some Primary Key or, always in the original database, some tuple is connected by a Primary / Foreign Key Relation. Our preliminary proposal, the description of the tabular data structure for storing and retrieving the possible connections among the documents and a metrics for scoring the results are introduced in the paper.

I. Introduction

Traditional database technology has been designed to serve query answering where users know well the elements of interest and through a formal and unambiguous query specify the conditions these elements should satisfy in order to be in the answer set. In the era of Big Data, data exploration has become as important as data querying since users need not to retrieve a portion of the data but to understand the data set in general. Often they may browse different parts of the data until they find something of interest. Unfortunately, existing query languages cannot be used easily for data exploration due to their strict semantics. Queries on structured data are issued assuming that a correct specification of the user information need exists and that answers are perfect, i.e. they follow the "exact match" search paradigm. On the other hand, end-users exploring a database are more oriented towards a "best match" search paradigm given that their information needs are often vague and subjected to a progressive process of refinement enabled by the search activity itself [**?**].

A technology that has recently received considerable attention and is gaining momentum is keyword querying on structured data sources [**?**]. We claim that keyword queries offer a great opportunity for data exploration since users can form a high level and generic request, but the results returned, since they come from structured data sources, contain not only the values but also the schema, i.e., the semantics, of these values, helping the user understand how the data is structured and its meaning.

The majority of the existing techniques that support keyword search over relational databases can be classified as schema-based or graph-based [**?**] approaches. Schema-based techniques exploit the schema information to issue SQL queries with the same meaning as the original keyword queries. On the other hand, graph-based techniques treat relational databases as graphs. Solving keyword queries in this context requires the computation of specific structures over the graphs (e.g., Steiner trees). A few proposals have adopted information retrieval (IR) style solutions, where the main issue is building meaningful and integral information units from the data (which in a relational database is spread over a number of tables) to index and rank according to the given query. In [**?**], these atomic pieces of data are called tuple units.

Within the last few years, we have engaged into a long line of work in which we have tried to tackle a number of interesting problems in keyword searching over relational structures. We have introduced a principal model for keyword search over structured databases and developed a number of schema-based techniques and prototypes based on heuristic and machine learning techniques. Our KEYMANTIC [**?**][**?**] system has focused on finding a solution based on a bipartite graph matching model where user keywords were matched to database schema elements by using an extension of the Hungarian algorithm. In KEYRY [**?**][**?**] we extended KEYMANTIC by providing a probabilistic framework, based on a Hidden Markov Model (HMM) to compute the mapoping of keywords to database structures. Finally, in QUEST [**?**][**?**] we provided a complete end to end solution that the mapping is by means of HMM and the way these mappings are combined by means of Steiner Trees and a probabilistic framework provided by the Dempster-Shafer theory.

In this paper, we introduce and analyze a new proposal that extends some of the ideas introduced in the Tuple Unit approach. In particular, we advocate that IR techniques can be exploited to provide an implementation of keyword search over the relational data. To do it, the relational databases have to be viewed and materialized into documents which in turn can be queries and retrieved in the same way these IR tasks have done with regular documents. The advantage is that the IR community has developed reliable, and mature technologies

available also as open libraries (see for example Apache Lucene[1], Terrier [2]) and evaluated with robust methodologies (see for example the TREC initiative[3]) to be used for this purpose. These techniques are mature and allow users to easily retrieve keywords and rank large collections of documents. The challenge is now how to make relations in a database "documents" to be efficiently and effectively retrieved and ranked by a IR system. Our approach is still under development. The next section introduces, also by examples, the main issues we are addressing and the main features of our idea.

II. Leveraging IR for Querying Structured Data

IR techniques require the use of some elementary units. Traditionally this role is played by documents. Unfortunately, in relational databases, the logical units are fragmented both horizontally in different tuples of the same table, and vertically in different tuples across different tables. Thus, there is a need for a way to turn the distributed tuples into documents. For this, we introduce the notion *base joining trees of tuples (BT)* that can play the role of the documents and we show how they are to be created. However, in contrust to the traditional documents, the BTs can be further combined to create additional more complex documents, i.e., they serve a role similar to base tables.

A. From tuples to documents

To turn tuples to documents we extend the idea of tuple units. A tuple unit is a set of tuples in different tables connected throwback referential constraints [**?**]. We extend the idea of tuple units by materializing and indexing *base joining trees of tuples (BTs)* obtained from the application of the *full disjunction* operator to the database.

We use the notion of *joined tuple tree* introduced in [**?**]. Consider a database with n relations R_1,...,R_n. Each relation R_i is composed of A_{1i},...,A_{mi} attributes, a primary key and possibly foreign keys into other relations. The schema graph G is a directed graph that captures the foreign key relationships in the schema. G has a node for each relation R_i, and an edge $R_i \rightarrow R_j$ for each primary key to foreign key relationship from R_i into R_j. A *Joined Tuple Tree* T is a tree of tuples where each edge (t_i, t_j) in T, with $t_i \in R_i$ and $t_j \in R_j$ satisfies two properties: (1) $(R_i, R_j) \in G$, and (2) $t_i\ t_j \in R_i\ R_j$. The $size(T)$ of a joining tree T is the number of tuples in T.

Any joined tuple tree of tuples T *consisting of at most n tuple* from each relation in $\mathcal{R}$ with $n \geq 1$ is called a *n*-level joining tree, denoted by T^n; a *1*-level joining tree is *base joining tree* and denoted by BT. The first interesting observation is that all possible BTs can be computed by means of the full disjunction (FD) [**?**] operator, an associative extension of the outer-join operator to an arbitrary number of relations; it has the main ability of maximally combining data from different relations while preserving all possible connections among the database tuples. BTs will be the elementary units of information that play the role of documents and are indexed and retrieved. The second interesting observation is that the *(n+1)*-level joined tuple trees are built upon those of *n*-level, which facilitates a map reduce implementation of their computation.

B. Joined Tuple Trees

A *keyword query* is a set of keywords $Q = [w_1, ..., w_m]$ specifying the user information need. An answer to the query Q is a n-level joining tree of tuples T containing all keywords w_i in Q.

Indexing Building. The computation of an answer requires an indexing system to efficiently retrieve the BTs containing all or part of the keywords in the query, to compute the possible joins between BTs, and to rank the answers obtained according to a score function that takes into account the selected BTs and the paths joining them. An approach computing firstly the best results reduces the answering time, potentially high due to the number of possible paths in a database.

To reach this goal we build two grouping tables for each level of joining tree n we want to compute[4]. The tables represent the possible ways two *n*-level trees can be groups in *n+1*-level trees. The grouping table ($KSET_{n+1}$) shows how *n*-level trees can be joined according to their primary keys, thus forming T^{n+1} trees. The second table $FKSET_n + 1$ is built upon the first one and shows the T^{n+1}s trees in $KSET_{n+1}$ joined with T^n trees, since they are the primary key in one primary / foreign key relation. The results are T^{n+2} trees.

Moreover, a number of inverted indexes are built on the $KSET_n$ and $FKSET_n$ tables. These inverted indexes, built and managed by means of Apache Lucene[5], are exploited for building the joining tables of the next level and, query-time, to find possible answers to the user queries.

Let us show through a running example how the data structures are created and used to provide query answers. Top-left part of Figure **??** shows a small database composed of three tables, representing *people* working in *Cities* and living in *Regions*. Note that *Cities* are located in *Regions*, and all these connections are coded in the database through primary / foreign key relationships connecting Person with Town and Region; Town with Region. Figure **??** shows also all possible BTs. Bottom part of the same Figure shows the *1*-level joining trees of tuples. For each BT B (Column KT), $KSET_1$ shows the BTs having one of the constituting tuples sharing a primary key with one of B constituting tuples (column TS_{KSET_1}). Column FKS_{KSET_1}) reports the BTs that references at least one of the elements in TS_{KSET_1}[6] through a primary / foreing key relation. We can easily see by construction that two joining tree of tuples $t_1, t_2 \in T^n$ are connected by a joining path if

[1] http://lucene.apache.org/core/

[2] http://terrier.org/

[3] http://trec.nist.gov/

[4] In the following, we suppose that an identifier is defined for each BT

[5] lucene.apache.org/

[6] We refer to TS_{KSET_1} as the TS column in table $KSET_1$. A similar notation has been adopted for the other joining tables.

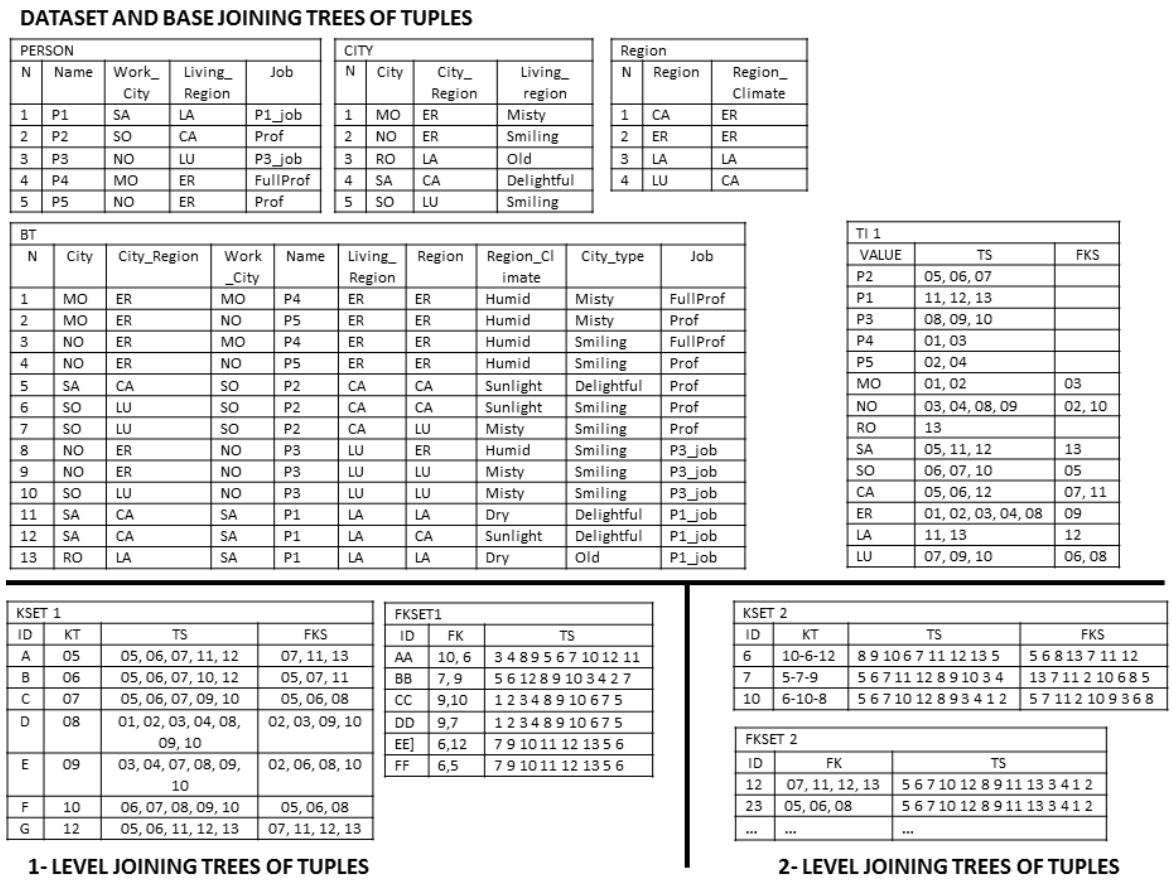

Fig. 1. The reference database

there is an entry in the table where (a) $t_1 and/or t_2 \in KT_1$ and (b) $t_1 and/or t_2 \in TS_{KSET_1}$. This property is true for each level n.

Example. The first entry in $KSET_1$ shows that BTs 5, 6, 7, 11, and 12 can be connected with each other via one of the tuples constituting BT 5. Moreover, tuples in these BTs are referenced via primary / foreign key relationships from tuples in BT 7, 11, and 13.

Table $FKSET_1$ includes an entry for each element in FKS_{TI_1} that is referencing an element in TS_{KSET_1} via a primary / foreing key relation. We can easily see that two joining tree of tuples $t_1, t_2 \in T^n$ are connected by a joining path if there is an entry in the table where $t_1 and t_2 are \in TS_{KSET_1}$. The joining path is the union of t_1, t_2 and the elements in FKS_{TI_1}.

Example. The BTs 8 and 6 forms a joining tree of tuples with BT 10 according to the first entry in $FKSET_1$.

For the sake of clarity, Figure **??** shows also table TI^1, containing the inverted indexes on primary and foreign keys of the original tuples in the database In particular, the TI_1 dictionary contains the primary keys of all tuples composing the BTs. For each primary key, TS_{TI_1} lists the BTs where the key appears, and FKS_{TI_1} shows the BTs (if any) where there is one of the constituting tuples referred as primary key in a primary / foreign key relationship.

Example. The first entry in TI_1 shows that it is possible to retrieve the value "P2" in the BTs identified as the values 5,6,7. Moreover "P2" is never a primary key in a primary / foreign key relationship.

Note that the index and joining tables related to T^{n+1} joining trees of tuples can be easily generated via the inverted index and the tables defined for T^n. Furthermore, the computation of a high number of indexes and joining tables is not needed: after few levels (1) we cannot obtain new groups (typically 8-9 in our experiments); (2) the data we are joining are to far for carrying out some interesting semantics for the user. Finally, the joining tables are built by means of inverted indexes. Therefore, the time required for the computation of these data structures is low, existing tools are able to process large amount of data, and, in any case, the operation is performed off-line (the data structures can be updated periodically).

Selection. Answering a keyword query means to select the n-level joining trees of tuples containing all keyword. Our idea is to base this task on the analysis of the indexes and joining tables. Let us consider a keyword query $Q = [w_1, ..., w_m]$, where for sake of simplicity, keywords refers only to primary

BT									
N	City	City_Region	Work_City	Name	Living_Region	Region	Region_Climate	City_type	Job
3	NO	ER	MO	P4	ER	ER	Humid	Smiling	FullProf
5	SA	CA	SO	P2	CA	CA	Sunlight	Delightful	Prof
7	SO	LU	SO	P2	CA	LU	Misty	Smiling	Prof
9	NO	ER	NO	P3	LU	LU	Misty	Smiling	P3_job
11	SA	CA	SA	P1	LA	LA	Dry	Delightful	P1_job

Fig. 2. An answer to the query P4, SO, NO

or foreign keys in the original database[7]. Let us start from the first level. Table TI_1 allows the system to map keywords $w_m \in Q]$ into sets of joining trees $B_m \in BT$ containing them. Joined tuple trees mapped by all keywords, i.e. $b \in B_m \forall m$, are answers to Q. Further answers can be found in $KSET_1$. In particular, an inverted index on the elements in TS_{KSET_1} allows to easily retrieve if there are rows in TS_{KSET_1} able to complete answer a query. Once identified the rows, the solution is computed by adding (if not already included) the Ts in KT_1 (needed to guarantee the existence of a connected tree) and removing the Ts in TS_{KSET_1} not needed. A similar approach can be applied to $FKSET_1$ and to the remining of TS_{KSET_n} and $FKSET_n$.

Example. Let us suppose that a user is interested in Person P4 working in City MO and living in ER. The inverted indexes in TI_1 show that keyword P4 can be found in BTs 1 and 3; keyword MO in BTs 1 and 2; keyword ER in BTs 1, 2, 3, 4, and 8. Only BT 1 is common to all keywords, and this is one possible answer for the keyword query. We can found other answers checking the TS values in tables $KSET_1$, $FKSET_1$, $KSET_2$, $FKSET_2$, and so on. For example, entry D in $KSET_1$ contains BT 1, and 8, that is another answer to the keyword query.

Let us suppose now that a user formulates the keyword query P4, SO, NO. TI_1 shows that P4 can be found in BTs 1 and 3; SO in BTs 6, 7, and 10; LA in BTs 11 and 13. Figure **??** shows that these keywords does not share any common BT. However, column TS of entry 7 in table $KSET_2$ contains BTs 3, 7, and 11. Therefore, the union of these BTs and the ones in column KT are an answer for the keyword query, as shown in Figure **??**.

Ranking. The strategy for ranking the answers takes into account three perspectives. The first evaluates the quality of the BTs selected as partial answers. The BTs are created by means of the Full Disjunction and this operators maximises the possible connections among the tables, without taking into account the original semantics the Designer had in mind when he created the database schema. This means that, even if "syntactically" correct, not all BTs have the same importance for the user. A specific score will be assigned to each BT, reflecting this knowledge. We plan to compute the score by combining two values: the importance of the tables involved in the full disjunction and of the path joining them. In [**?**] a method based on entropy and foreign key / primary key relationships for computing the importance of tables and paths is proposed.

The second perspective evaluates the ability of a BT as a "document" to satisfy a keyword query. The usual parameters adopted in IR (term frequency, inverse document frequency, document size, ...) can be adopted for this purpose.

Finally, the third perspective evaluates the distance among the BTs which have been selected to form an answer. The higher the n-level of the trees involved the lower should be the answer score.

III. Conclusion

We have described a new approach that leveraging on big data technologies and information retrieval for computing answers to keyword queries over relational databases. We provide a technique enabling keyword search on a unified single-table representation of the content of a relational database, built based on the notion of full-disjunction [**?**]. We described how indexes can be used to answer the keyword queries and re-generate the relations from the retrieved documents.

[7]Finding for each keyword the primary key of the tuple where the keyword can be found is an easy task.