TupleRank and Implicit Relationship Discovery in Relational Databases

Andy Huang Qiang Xue Jun Yang
Department of Computer Science, Duke University, Durham, NC 27708, USA
{huangxia,xue,junyang}@cs.duke.edu

Abstract

Google's successful PageRank brings to the Web an order that well reflects the relative importance of Web pages. Inspired by PageRank, we propose a similar scheme called TupleRank for ranking tuples in a relational database. Database tuples naturally relate to each other through referential integrity constraints declared in the schema. However, such constraints cannot capture more general relationships such as similarity. Furthermore, relationships determined statically from the database schema do not reflect actual query patterns that arise at runtime. To address these deficiencies of static TupleRank, we introduce the notion of query-driven TupleRank, which is based on a link structure that is dynamically constructed from a workload. Specifically, database tuples are considered to be related if they are joined together by a query result tuple. The main challenge in supporting query-driven TupleRank is how to record multi-way relations among tuples efficiently for a large workload. We develop techniques to compute query-driven TupleRank accurately and efficiently with low space requirement that is independent of the workload size. We further augment query-driven TupleRank so that it can better utilize the access frequency information collected from the workload. Preliminary experiment results demonstrate that TupleRank is both informative and intuitive, and they confirm the advantages of query-driven TupleRank over static TupleRank.

1 Introduction

The Google search engine [3] has brought to the enormous Web an order that well reflects the relative "importance" of Web pages. The ranking is called *PageRank* [11], which is based on a simple yet elegant idea: A Web page is "important" if many "important" Web pages link to it. PageRank can be computed efficiently over the link structure of the Web using an iterative procedure. The success of Google is the best testimony to the effectiveness of PageRank.

Inspired by PageRank, we propose to bring to relational databases a similar order, called *TupleRank*, which measures the relative importance of database tuples. In a relational database, there naturally exists a link structure induced by the referential integrity relationships between tuples. Based on this link structure, we can define TupleRank in the same way as PageRank. Intuitively, a tuple will have a high TupleRank value if it is referenced by many tuples with high TupleRank values. We call this definition of TupleRank *static* because it is based on the referential integrity constraints declared statically as part of the database schema.

However, explicitly declared database constraints often fail to capture all interesting relationships among database tuples. First, not all constraints can be conveniently declared in SQL. For example, referential integrity constraints are only a special case of inclusion dependencies. Inclusion dependencies indicate more general links between tuples that share values for some attributes. However, there is no direct way in SQL to declare general inclusion dependencies. Although it is possible to implement inclusion dependencies using `CHECK`'s and/or triggers, it is difficult to infer from `CHECK` and trigger definitions what constraints they enforce.

Secondly, database tuples could relate to each other in ways that are not reflected at all by constraints. For example, a tuple with an image attribute containing a picture of the Great Wall should be considered related to another tuple with a text attribute containing the string "`Great Wall`." In general, tuples may be related by any join condition that is meaningful to the application, which can be based on exact equality or fuzzy measures of similarity.

Finally, database constraints are static, and do not capture the dynamics of a real workload. Intuitively, tuples (or links between tuples) that are accessed (or traversed) frequently in a workload should be considered as more "important" than those that are not.

Therefore, we argue that the link structure of a database should be discovered dynamically from a query workload, rather than determined statically from the database schema. Specifically, tuples are considered to be related if they are joined together by a query in the workload. We can then compute TupleRank over this dynamically discovered link structure. We call this definition of TupleRank *query-driven* because the link structure is derived from a query workload. In contrast to static TupleRank, query-driven TupleRank can capture relationships that are not explicitly declared as database constraints. Furthermore, query-driven TupleRank reflects the tuple access pattern of the workload. This feature raises the interesting possibility of defining, for each user community of the database, a "customized" TupleRank computed according to the community's typical query workload.

The major challenge in implementing query-driven TupleRank is how to keep track of the joins of database tuples generated by a large workload. A single query result tuple may be derived by joining multiple database tuples. In general, a multiway relationship with arity greater than two cannot be modelled as links between pairs of tuples (which are binary relationships). A possible solution is to model query result tuples explicitly in the link structure and have them connect to joining database tuples, a trick reminiscent of the use of *connecting entity sets* in E/R design [8]. However, this simple solution is not scalable because the size of the link structure grows with the number of result tuples generated by the workload. To overcome this problem, we generalize the link structure to a *weighted TupleLink graph*, and develop a series of graph compaction techniques to reduce the size of the graph to $O(m^2)$, where m is the number of database tuples, without affecting the accuracy of TupleRank. In practice, we find the size of the graph to be much smaller, typically $\Theta(m)$.

In summary, the main contributions of our work are:

- The extension of PageRank to TupleRank for relational databases.
- The idea of inferring the link structure of a database dynamically from a query workload (rather than statically from the database schema), allowing the capture of the implicit relationships and access frequency information in the workload.
- Techniques for compacting the link structure, allowing it to be constructed incrementally from a workload using at most $O(m^2)$ space, and enabling efficient and accurate computation of query-driven TupleRank.

Preliminary experiments on two databases show that TupleRank is both interesting and informative, and is generally consistent with users' intuitive understanding of the relative importance of tuples. Experiment results also confirm the advantages of query-driven TupleRank over static TupleRank.

The remainder of this paper is organized as follows. Section 2 surveys related work. Section 3 discusses static TupleRank based on referential integrity constraints. Section 4 introduces query-driven TupleRank and techniques that enable its efficient and accurate computation. Preliminary experiment results are presented in both sections. Finally, Section 5 concludes the paper and discusses future work.

2 Related Work

The idea of querying a relational database as a graph is proposed by Goldman et al. [9]. In their work on proximity search in databases, nodes of the graph represent tuples and attributes, and undirected weighted edges connect nodes having relationships between them. The relationships, however, are all based on static database constraints. In particular, only three kinds of edges are considered: an edge between a tuple and its attribute, an edge between two tuples with referential integrity relationship, and an edge between two tuples that belong to the same database table.

There is a flurry of recent work on supporting keyword-style searches in relational databases. Most of them are based on constraints declared in the database schema. Goldman et al. [9] propose the idea of proximity searches by locating a database subgraph whose nodes are within a certain distance to a query node. Bhalotia et al. [2] further incorporate in their BANKS search engine the concept of node prestige. However, their node prestige only depends on the in-degree of a node, and all edges in their database graph are derived from referential integrity constraints (they also mention inclusion dependency). Similar systems include DataSpot [6], which also uses edges based on the static database schema and user-defined associations, and DISCOVER [10], which uses the database schema graph to generate joins to answer keyword search queries.

3 Static TupleRank

PageRank views the Web as a directed graph, whose nodes represent Web pages and edges represent hyperlinks between Web pages. The PageRank of a page p, denoted $\text{PageRank}(p)$, is defined by:

$$\text{PageRank}(p) = \sum_{q \in B(p)} \frac{\text{PageRank}(q)}{N(q)}, \tag{1}$$

where $B(p)$ is the set of pages that point to p, and $N(q)$ is the number of hyperlinks on page q. PageRank can be computed through a simple iterative algorithm, whose convergence is guaranteed by the introduction of a damping factor $d \in (0,1)$. Let $\text{PageRank}^{(k)}(p)$ denote the PageRank of p at iteration k. We have

$$\text{PageRank}^{(k+1)}(p) = d \cdot \sum_{q \in B(p)} \frac{\text{PageRank}^{(k)}(q)}{N(q)} + (1-d). \tag{2}$$

Intuitively, $\text{PageRank}(p)$ corresponds to the probability that p is visited by a surfer in the *random surfer model* [11]. Damping factor d is the probability that a surfer follows one of the hyperlinks on the current page (instead of jumping directly to some random page).

Similarly, we define a *static TupleLink graph* based on the referential integrity relationships between tuples in a database. A static TupleLink graph is a directed graph whose nodes represent database tuples. There is a directed edge from tuple t_i to tuple t_j (and from t_j to t_i) if t_i's foreign key references t_j's primary key. We define the *static TupleRank* of a tuple t, denoted TupleRank(t), as follows:

$$\text{TupleRank}(t) = \sum_{t' \in B(t)} \frac{\text{TupleRank}(t')}{N(t')}, \tag{3}$$

where $B(t)$ is the set of tuples with outgoing edges to t in the TupleLink graph, and $N(t')$ is the number of outgoing edges from tuple t'. The TupleLink graph and TupleRank defined above are static, because the relationships between tuples are derived from static schema information, namely referential integrity constraints.

Let m be the total number of tuples in the database. Let $\mathbf{r}$ be a vector of size m, where $\mathbf{r}_i = \text{TupleRank}(t_i)$. Let $\mathbf{A}$ be an $m \times m$ matrix, where $\mathbf{A}_{ij} = 1/N(t_j)$ if there is an edge from t_j to t_i in the TupleLink graph, or 0 otherwise. Then, TupleRank can be computed by a simple iterative procedure using a damping factor d, just like PageRank:

$$\mathbf{r}^{(k+1)} = d\mathbf{A}\mathbf{r}^{(k)} + (1-d). \tag{4}$$

Experiments. We have computed static TupleRank for a relationship-rich geographical database, Mondial [1]. The database contains about ten thousand tuples storing information about countries, cities, rivers, seas, and other geographical entities. Relationships between these entities include, for example, "Chengdu is located in China," "Changjiang (spelled as 'Jangtse-Kiang' in Mondial) flows into East China Sea," etc. The schema of Mondial (with all referential integrity constraints) is shown in Figure 1.

To compute static TupleRank, we use a straightforward implementation based on the sparse matrix library `sparselib++` [7], using $d = 0.85$. The results are stored in a new database table *TupleRanks*(*TupleId*, *TupleRank*). Here, the *TupleId* of a database tuple consists of its table name and primary key value (which makes *TupleId* unique within the database). If no primary key is declared, a surrogate tuple identifier is used instead. For each original database table, we augment it with an additional *TupleRank* column by defining a join view over the original table and *TupleRanks*. Through this view, users can query the TupleRank of a tuple as an ordinary column.

Users of our system have found TupleRank to be quite interesting and informative. For example, the three top-ranking `Organization` tuples (with very close TupleRank values) are the Universal Postal Union, World Health Organization, and United Nations. The top-ranking `River` tuple is the Danube (Donau) river, probably because it crosses the largest number of countries. The two top-ranking `City` tuples in United States are New York City and Washington, DC. In general, most rankings are consistent with users' intuitive understanding of the relative importance of tuples in the geographical sense.

4 Query-Driven TupleRank

The main idea behind query-driven TupleRank is quite intuitive: Each query result tuple q obtained by joining database tuples $t_1, \ldots, t_n$ should be considered as an "evidence" that $t_1, \ldots, t_n$ are

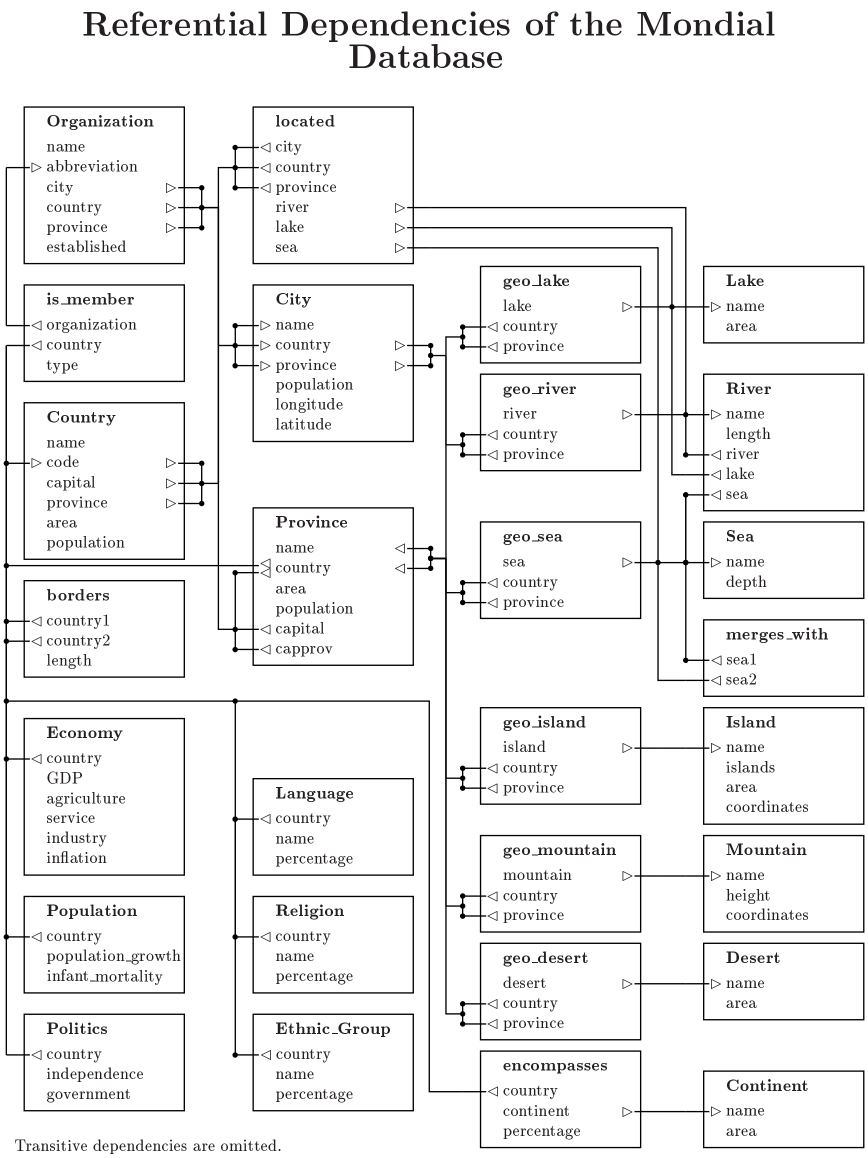

Figure 1: Mondial database schema (from [1]).

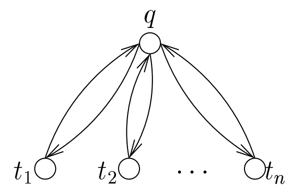

Figure 2: Example of a TupleLink graph capturing an n-way join relationship.

"related." To capture this idea in a TupleLink graph, we can treat the result tuple q as node that is connected to $t_1, \ldots, t_n$ (Figure 2). Formally, we define a *query-driven TupleLink graph* as a triple (T, Q, E), where

- T is a set of nodes that correspond to database tuples.
- Q is a set of nodes that correspond to query result tuples.
- E is a set of directed edges connecting nodes in $T \cup Q$.

A naive algorithm for constructing a query-driven TupleLink graph from a query workload works as follows:

- For each database tuple, create a node in T.
- For each query in the workload, and for each result tuple q returned by the query:
  - ◇ Create a node in Q.
  - ◇ Suppose that q is obtained by joining database tuples $t_1, \ldots, t_n$. Create a directed edge in E from q to each t_i $(1 \le i \le n)$, and from each t_i to q.

We can capture referential integrity relationships by manually adding to the workload queries that join foreign keys with corresponding primary keys. In practice, however, we expect such queries to arise naturally in a workload, so there is no need to deal with referential integrity relationships explicitly.

For now, we assume that the workload contains only simple select-project-join (SPJ) queries with no duplicate elimination. These queries usually constitute the bulk of common workloads. For these queries, it is clear that a result tuple is derived from joining database tuples. For other types of queries, such as those that use `EXCEPT`, it is less obvious which database tuples contribute to a result tuple. We plan to address other types of queries as future work. Work on lineage tracing [5] may provide a good starting point.

Given a query-driven TupleLink graph (T, Q, E), we can defined the *query-driven TupleRank* of a tuple t in either T or Q as follows:

$$\text{TupleRank}(t) = \sum_{t' \in B(t)} \frac{\text{TupleRank}(t')}{N(t')}, \tag{5}$$

where $B(t)$ and $N(t')$ have the same definitions as in the case of static TupleRank.

Although the query-driven TupleLink graph and TupleRank have very intuitive definitions, they pose serious implementation problems, because the size of the TupleLink graph increases linearly with the workload. As more queries enter the workload, more result tuples are generated, and $|Q|$ and $|E|$ can grow without any bound. In Section 4.1, we propose a series of techniques for compacting the TupleLink graph. In Section 4.2, we describe an algorithm for computing query-driven TupleRank by building a compact TupleLink graph that requires only $O(|T|^2)$ space,

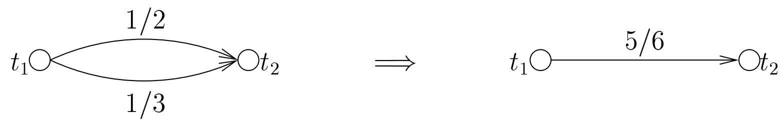

Figure 3: Example of edge merging.

independent of the size of the workload. In Section 4.2, we also introduce a further enhancement of query-driven TupleRank which does a better job utilizing the access frequency information collected from the workload. Preliminary experiment results are presented in Section 4.3.

4.1 Weighted TupleLink Graph and Graph Compaction Techniques

The first step towards compacting the TupleLink graph is generalizing it to a *weighted TupleLink graph*. The basic idea is to associate a positive weight with each edge. The static TupleLink graph can be seen as a special case where all weights are 1. Formally, we define a weighted TupleLink graph as a quadruple (T, Q, E, w), where T, Q, E have the same definitions as before, and $w : E \to (0, \infty)$ is a function that assigns a positive weight to each edge in E. Furthermore, we allow nodes to connect to themselves via self-loop edges, and a pair of nodes to be connected by multiple edges. For convenience, we define $w(t)$, the weight of a node $t \in T \cup Q$, as the sum of weights on the edges coming out of t.

Now, we can define TupleRank on a weighted TupleLink graph (T, Q, E, w) as follows:

$$\text{TupleRank}(t) = \sum_{e \in \mathit{into}(t)} \frac{w(e) \cdot \text{TupleRank}(\mathit{from}(e))}{w(\mathit{from}(e))}, \tag{6}$$

where $\mathit{into}(t)$ is the set of edges coming into t in the weighted TupleRank graph, and $\mathit{from}(e)$ is the source node of edge e. Intuitively, the TupleRank of a node is distributed to all nodes that it points to, weighted by the edge and scaled by the total weight of outgoing edges.

Edge merging. Intuitively, if two edges share the same source and destination nodes, then we can replace these two edges by one whose weight is the sum of the weights of the original edges. The result TupleLink graph after edge merging should produce identical TupleRank values for all nodes as the original graph. An example of edge merging is shown in Figure 3. Formally, we have the following lemma.

Lemma 1 *Consider a weighted TupleLink graph $G(T, Q, E, w)$. Suppose that $e_1, e_2 \in E$ both go from t_1 to t_2, where $t_1, t_2 \in T \cup Q$. Define a second graph $G'(T, Q, E', w')$, where*

- *$E' = E - \{e_1, e_2\} \cup \{e_3\}$, where e_3 is a new edge that goes from t_1 to t_2.*
- *$\forall e \in E - \{e_1, e_2\} : w'(e) = w(e)$.*
- *$w'(e_3) = w(e_1) + w(e_2)$.*

Then, G and G' produce identical TupleRank values (as defined by Equation 6) for all nodes in $T \cup Q$.

Note that this lemma still holds when $t_1 = t_2$, i.e., both e_1 and e_2 are self-loops on the same node. The proof for this lemma can be found in the appendix.

Lemma 1 alone is not enough to compact a TupleRank graph since Lemma 1 does not reduce the number of nodes in the graph. Next, we propose a technique for removing nodes in the graph.

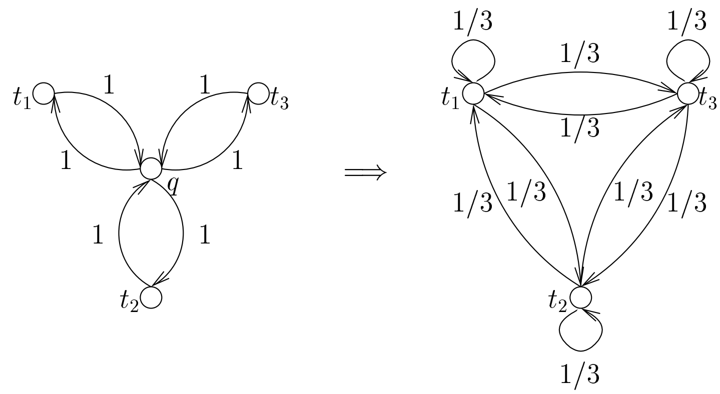

Figure 4: Example of node removal.

Node removal. Recall that a TupleLink graph is defined in terms of two sets of nodes: those that correspond to database tuples (T) and those that correspond to query result tuples (Q). The goal of node removal is to eliminate all nodes from Q without affecting the TupleRank values of nodes in T.

To motivate the next lemma, we use a very simple example of a query result tuple q that joins three database tuples t_1, t_2, and t_3, as shown on the left in Figure 4. Let R_1 denote the contribution to TupleRank(t_1) from edges that are not incident to/from q (not shown in the figure). R_2 and R_3 are defined similarly for t_2 and t_3. According to Equation 6,

$$\text{TupleRank}(t_1) = R_1 + \frac{1 \cdot \text{TupleRank}(q)}{3}; \tag{7}$$

$$\text{TupleRank}(t_2) = R_2 + \frac{1 \cdot \text{TupleRank}(q)}{3}; \tag{8}$$

$$\text{TupleRank}(t_3) = R_3 + \frac{1 \cdot \text{TupleRank}(q)}{3}; \tag{9}$$

$$\text{TupleRank}(q) = \frac{1 \cdot \text{TupleRank}(t_1)}{w(t_1)} + \frac{1 \cdot \text{TupleRank}(t_2)}{w(t_2)} + \frac{1 \cdot \text{TupleRank}(t_3)}{w(t_3)}. \tag{10}$$

Consider the graph on the right, where q and its incident edges are removed, and new edges between t_1, t_2, and t_3 and self-loops on them are added, all with weight 1/3. Note that in this graph, $w(t_1)$, $w(t_2)$, and $w(t_3)$ remain the same as before, because for each node, the weights on three outgoing edges add up to 1. According to Equation 6,

$$\begin{aligned}
\text{TupleRank}(t_1) &= R_1 + \left(\frac{\frac{1}{3} \cdot \text{TupleRank}(t_1)}{w(t_1)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_2)}{w(t_2)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_3)}{w(t_3)}\right); \\
\text{TupleRank}(t_2) &= R_2 + \left(\frac{\frac{1}{3} \cdot \text{TupleRank}(t_1)}{w(t_1)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_2)}{w(t_2)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_3)}{w(t_3)}\right); \\
\text{TupleRank}(t_3) &= R_3 + \left(\frac{\frac{1}{3} \cdot \text{TupleRank}(t_1)}{w(t_1)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_2)}{w(t_2)} + \frac{\frac{1}{3} \cdot \text{TupleRank}(t_3)}{w(t_3)}\right).
\end{aligned}$$

It is easy to see that the above three equations are equivalent to Equations 7–9, if we simply substitute the definition of TupleRank(q) in Equation 10 into Equations 7–9. For any other node t in the graph (not shown in the figure), TupleRank(t) and $w(t)$ should have identical definitions as before, since the set of edges incident to/from t has not changed. Therefore, the definitions of R_1, R_2, and R_3 also remain the same as before.

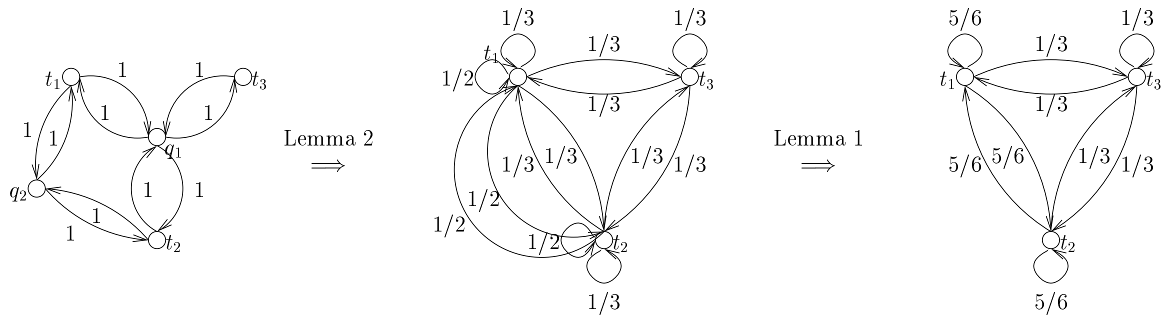

Figure 5: Example of graph compaction.

In general, we can remove a n-way join result tuple q by adding edges between all pairs of database tuples that q connects to, as well as edges from each such tuple to itself. All new edges should have weight $1/n$. Formally, we have the following lemma, which has been generalized for the case where edges do not necessarily have unit weights.

Lemma 2 *Consider a weighted TupleLink graph $G(T, Q, E, w)$. Suppose that $q \in Q$ has n outgoing edges to tuples $t_1, \ldots, t_n$ with weights $w_1, \ldots, w_n$, respectively, and n incoming edges from $t_1, \ldots, t_n$ with same weights $w_1, \ldots, w_n$, respectively. Define a second graph $G'(T, Q', E', w')$, where*

- $Q' = Q - \{q\}$.
- $\bigtriangledown E = \{(q, t_i) \mid 1 \leq i \leq n\} \cup \{(t_i, q) \mid 1 \leq i \leq n\}$.
- $\triangle E = \{(t_i, t_j) \mid 1 \leq i \leq n, 1 \leq j \leq n\}$.
- $E' = E - \bigtriangledown E \cup \triangle E$.
- $\forall e \in E - \bigtriangledown E : w'(e) = w(e)$.
- $\forall e(t_i, t_j) \in \triangle E : w'(e) = (w_i \cdot w_j)/(\sum_{1 \leq k \leq n} w_k)$.

Then, G and G' produce identical TupleRank values (as defined by Equation 6) for all nodes in $T \cup Q'$.

Note that this lemma still holds in the case of self-joins, i.e., $t_i = t_j$ for some i and j. The proof for this lemma can be found in the appendix.

Starting with any weighted TupleLink graph $G(T, Q, E, w)$, we can apply Lemma 2 repeatedly to remove all nodes from Q, one at a time, while adding more edges to E between nodes in T. Then, we can apply Lemma 1 repeatedly to merge edges in E until only one edge remains for each pair of nodes in T. In the result graph, $Q = \varnothing$ and $|E| \leq |T|^2$. A very simple example of this graph compaction process (starting with a graph with $|Q| = 2$) is illustrated in Figure 5.

4.2 Compact TupleLink Graph Construction and TupleRank Computation

We now present an algorithm for computing query-driven TupleRank by incrementally building a compact TupleLink graph from a query workload. Suppose the database contains m tuples $t_1, \ldots, t_m$. The weighted TupleLink graph can be encoded by an $m \times m$ matrix $\mathbf{A}$. We also maintain the access frequency of each tuple in a frequency vector $\mathbf{f}$ of size m, which can be used to enhance TupleRank computation, as we shall discuss later in this section.

- Initially, set all entries of $\mathbf{A}$ and $\mathbf{f}$ to 0.

- For each query in the workload, and for each result tuple q returned by the query:
  - ◇ Suppose that q is obtained by joining n database tuples $t_{i_1}, \ldots, t_{i_n}$.
  - ◇ For each integer $u \in [1, n]$, and for each integer $v \in [1, n]$, update $\mathbf{A}$ as follows: $\mathbf{A}_{i_u,i_v} \leftarrow \mathbf{A}_{i_u,i_v} + 1/n$, and $\mathbf{A}_{i_v,i_u} \leftarrow \mathbf{A}_{i_v,i_u} + 1/n$.
  - ◇ For each integer $u \in [1, n]$, increment $\mathbf{f}_{i_u}$ by 1.
- Finally, we scale $\mathbf{A}$ to ensure that entries in each column sum up to 1 (unless the column consists entirely of 0's): $\mathbf{A}_{i,j} \leftarrow \mathbf{A}_{i,j} / \sum_{1 \leq k \leq m} \mathbf{A}_{i,k}$ (provided that $\sum_{1 \leq k \leq m} \mathbf{A}_{i,k} \neq 0$).
- We also scale $\mathbf{f}$ to ensure that all entries sum up to 1: $\mathbf{f}_i \leftarrow \mathbf{f}_i / \sum_{1 \leq k \leq m} \mathbf{f}_k$.

Let $\mathbf{r}$ be a vector of size m, where $\mathbf{r}_i = \text{TupleRank}(t_i)$. We then have the following equivalent formulation of query-driven TupleRank:

$$\mathbf{r} = \mathbf{A}\mathbf{r}. \tag{11}$$

¿From Lemmas 1 and 2, the following theorem follows naturally:

Theorem 1 *Given a database and a query workload, construct a query-driven TupleLink graph (T, Q, E) using the naive algorithm discussed at the beginning of Section 4, and construct a matrix $\mathbf{A}$ using the algorithm discussed in Section 4.2. Then, the TupleRank definition in Equation 5 based on (T, Q, E) is equivalent to the TupleRank definition in Equation 11 based on $\mathbf{A}$.*

Again, TupleRank can be computed by a simple iterative procedure with a damping factor d, starting from $\mathbf{r}^{(0)} = (1, 1, \ldots)$:

$$\mathbf{r}^{(k+1)} = d\mathbf{A}\mathbf{r}^{(k)} + (1 - d). \tag{12}$$

We introduce another enhancement of TupleRank which makes better use of the access frequency information stored in $\mathbf{f}$. Recall that we can regard the damping factor d as the probability that a database user chooses to follow a link from the current tuple. With probability $1 - d$, the user chooses a random tuple to revisit next. Equation 12 above simply assumes that the next tuple is chosen uniformly at random (similar to PageRank). However, with the access frequency information from the workload, we may assume instead that the next tuple is chosen according to the frequency vector $\mathbf{f}$. Thus, starting with $\mathbf{r}^{(0)} = (1/m, 1/m, \ldots)$, we have

$$\mathbf{r}^{(k+1)} = d\mathbf{A}'\mathbf{r}^{(k)} + (1 - d)\mathbf{f}, \tag{13}$$

where $\mathbf{A}' = \mathbf{A} - \text{diag}(\mathbf{A})$, and further scaled to ensure that entries in each column sum up to 1.

The reason for zeroing out the diagonal entries is the following. ¿From the construction algorithm, we can easily see that the diagonal entries of $\mathbf{A}$ also capture the tuple access frequency information as does $\mathbf{f}$. Since Equation 13 already makes explicit use of the frequency information through $\mathbf{f}$, it makes sense to remove this information in $\mathbf{A}'$ to avoid overuse. We find that TupleRank can make more effective use of the frequency information with $\mathbf{f}$ than with $\mathbf{A}$, because the diagonal entries of $\mathbf{A}$, representing the self-loops on nodes in the TupleLink graph, tend to increase the total weight of outgoing edges from a node, making it harder for the node to contribute its TupleRank to others. On the other hand, the contribution from $\mathbf{f}$ can be propagated to others more easily.

4.3 Implementation

In implementing query-driven TupleRank, the first issue that must be addressed is how to determine which database tuples contribute to a query result tuple. As mentioned earlier in Section 4, for simple SPJ queries with no duplicate elimination, it is clear that a result tuple is derived from joining database tuples. Given one such query in the workload, we can rewrite its `SELECT` clause to return the *TupleId* values (as defined earlier in Section 3) for tables in the `FROM` clause. Using the Mondial schema as an example, the query

```
SELECT Country.name, Economy.GDP
FROM Country, Economy
WHERE Country.code = Economy.country
AND Economy.industry > 50;
```

can be modified into

```
SELECT 'Country' || Country.code,
       'Economy' || CAST (Economy.id AS CHAR(10)),
FROM Country, Economy
WHERE Country.code = Economy.country
AND Economy.industry > 50;
```

The first two output columns uniquely identify the joining `Country` and `Economy` tuples. `Economy.id` is a surrogate key since no primary key is declared for `Economy`. For queries involving subqueries, we often can flatten them into simple unnested select-project-join queries and then use the approach to identify the joining tuples. As mentioned earlier, we plan to address other types of queries as future work.

In practice, the algorithm presented in Section 4.2 can be implemented using the following approaches:

1. *Using a copy of the production database.* We continuously ship queries and updates from the production database to a copy of it. Updates are applied verbatim on the copy. Queries are rewritten to return *TupleId* values (as described above) and then processed on the copy. The returned *TupleId* values allow us to update **A** and **f**.

2. *Directly on top of the production database.* A layer can be implemented directly on top of the database API (e.g., JDBC) to intercept application queries. We augment these queries to return *TupleId* values before passing them to the database for evaluation. The returned *TupleId* values allow us to update **A** and **f**. Before returning to the application, we must remove these extraneous *TupleId* values from the result.

For both approaches, we can periodically recompute TupleRank using **A** and **f**, and then reset them. Compared with the first approach, the second approach does not re-execute queries and updates, and it does not require additional resources; however, it may potentially slow down applications. Currently, we have implemented only the first approach. Experiment results using this implementation are described in the next section.

Country	static	query-driven (W_{ref})	query-driven (W_2)	query-driven (W_3)
France	6	6	7	5
Switzerland	11	11	12	7
Belgium	28	28	29	14
Guinea	81	81	83	50
Haiti	137	137	1	37
French Guiana	192	192	192	184

Table 1: TupleRank of French-speaking countries in Mondial.

4.4 Experiments

The first series of experiments are conducted on Mondial, with the primary goal of evaluating the intuitiveness of query-driven TupleRank. Mondial is a large and complex database, so we focus on tracking TupleRank for a small number of tuples as we adjust the query workload. Specifically, we track six `Country` tuples that correspond to the six French-speaking countries (according to Mondial, which may not be complete): France, Switzerland, Belgium, Guinea, Haiti, and French Guiana. Static TupleRank for these `Country` tuples are shown in the second column of Table 1. Instead of showing their actual TupleRank values, we show their *ranks* among all `Country` tuples, which are more meaningful. For example, France is ranked sixth among all countries, while French Guiana is ranked 192nd.

We start with a query workload W_{ref} consisting of join queries along all referential integrity relationships declared in Mondial's schema. We compute the query-driven TupleRank for W_{ref} and find the result ranking (shown in the third column of Table 1) to be identical to that of static TupleRank, as expected.

Next, we construct a second workload W_2 by adding a huge number (1000) of the following query to W_{ref}:

```
SELECT * FROM Language WHERE country = 'RH';
```

where `RH` is the country code representing Haiti. These queries drive up the access frequency of the selected `Language` tuple, but the access frequencies of all other database tuples remain unchanged as in W_{ref}. We show the query-driven TupleRank computed for W_2 in the fourth column of Table 1. Note that Haiti becomes the top-ranking `Country` tuple, even though its access frequency has not changed. The reason is that the selected `Language` tuple is related to the `Country` tuple for Haiti through a referential integrity constraint. Thus, Haiti receives a boost in TupleRank through the selected `Language`. This result clearly demonstrates that query-driven TupleRank is frequency-aware, yet it offers more than a simple ranking based on access frequency alone.

On the other hand, TupleRank values for other French-speaking countries remain practically unchanged (some of them drop by one because Haiti has risen to the top). A closer look at Mondial's schema (Figure 1) reveals the problem: There is no link between two countries that speak the same language. Although there are two `Language` tuples with identical `Language.name`, each linking to one of the two countries, these two `Language` tuples do not link to each other. This link can be captured by a self-join of `Language` on `Language.name`, but it is beyond what referential integrity constraints can capture.

Fortunately, query-driven TupleRank comes to rescue. To capture the relationship between countries speaking the same language, we construct a third workload W_3 by adding the following query (once is enough) to W_2:

```
SELECT * FROM Language L1, Language L2 WHERE L1.name = L2.name;
```

The query-driven TupleRank values for W_3 are shown in the last column of Table 1. Note that the ranks of French-speaking countries all improve consistently. Compared with static TupleRank, Haiti still receives the most significant boost; after all, it is most closely related to the particular `Language` tuple being accessed 1000 times. On the other hand, the boost is not as much as in the case of W_2, because now the boost is also distributed to other French-speaking countries. The last two experiments together demonstrate the inadequacy of static TupleRank and the need for discovering relationships from query workloads.

We have also experimented with the TPC-W database. TPC-W [12] is a benchmark designed to evaluate the performance of a complete Web e-commerce solution. The benchmark specification details database layout and sizing parameters, the set of database transactions used by the server, the set of possible Web interactions, and the design of a load generator. The server implements a small but functional online bookstore. The load generator emulates the behavior of customers interacting with the Web site through HTTP requests. We use a Java implementation of the TPC-W benchmark from University of Wisconsin [4] to generate the workload for query-driven TupleRank. Since TPC-W is a synthetic database, it is not meaningful to report TupleRank for randomly generated tuples. Therefore, the primary goal of our TPC-W experiments is to evaluate the efficiency of query-driven TupleRank. The results are shown in Table 2. All experiments use the TPC-W browsing mix since we are interested primarily in queries. We vary the number of EB's (emulated client browsers running concurrently) as well as the number of items for sale. A larger number of EB's means that more queries, and consequently more result tuples, will be generated. On the other hand, although a larger number of items implies a larger database, the number of result tuples is actually smaller. The reason is that the workload contains a number of queries that compute the join between orders and best-selling items. With fewer items, each item will have more orders, so the join result will be larger.

Computation of TupleRank is done on a Sun Blade 100 workstation with a 500MHz UltraSPARC-IIe processor and 256MB of RAM. Overall, we see that the matrix $\mathbf{A}$ is extremely sparse, because each database tuple usually joins with a constant number of tuples. Hence, in practice, the space requirement of our algorithm tends to be $\Theta(m)$, where m is the number of database tuples. Size of the workload has no significant effect on either space or time required by TupleRank computation. Even for large databases (30 EB's/$100,000$ items) and workloads (more than $6,042$ queries), TupleRank computation completes in less than 3.2 seconds on a fairly underpowered machine.

5 Conclusion and Future Work

To conclude, we have proposed a scheme called TupleRank for ranking tuples in a relational database, based on the same intuition as Google's PageRank. We have also introduced the notion of query-driven TupleRank, which is based on a link structure that is dynamically constructed from a workload. To the best of our knowledge, we are the first to propose using such a query-driven link structure for determining relationships among database tuples. As we have shown,

Number of EB's/items	10/100,000	30/100,000	10/1,000	30/1,000
Number of Database tuples	340,748	773,665	217,114	649,370
Number of queries generated	2,454	6,042	2,608	8,104
Number of result tuples	32,402	107,496	531,585	4,491,969
Number of non-zero entries in **A**	23,400	36,084	295,803	894,699
Sparsity of **A**	99.99998%	99.99999%	99.99994%	99.99999%
TupleRank computation time (sec)	2.1	3.1	11.2	38.6

Table 2: Result of experiments on TPC-W.

query-driven TupleRank can capture the implicit relationships and access frequency information in a query workload, both of which are beyond the capabilities of traditional approaches based on static schema information. Finally, we have developed techniques to compute query-driven TupleRank accurately and efficiently. In the following, we briefly outline some future directions.

- We are currently investigating potential applications of TupleRank. Besides its benefit to database users in providing an interesting and informative ranking, we think that TupleRank also can be used in designing cache replacement and replication policies. Intuitively, tuples with higher TupleRank values are more worthy of caching or replicating. We conjecture that TupleRank might be a better metric than plain access frequency because tuples with higher TupleRank values tend to have better "connections" to other tuples and therefore can potentially benefit more accesses.

- Currently, we recompute TupleRank and reset **A** and **f** periodically. A better approach is to update **A** and **f** incrementally and gradually decrease the effect of past queries on them. Another difficult but interesting problem is how to compute TupleRank incrementally given small perturbations to **A** and **f**.

- As mentioned in Section 4, we plan to investigate how to use non-SPJ queries in the workload to identify more relationships among database tuples. Also, we plan to experiment with the second implementation approach discussed in Section 4.3.

- Our current graph compaction techniques still ensure the accuracy of TupleRank. The next natural question is whether we can trade the accuracy of TupleRank for further space and time reductions.

References

[1] The Mondial database. `http://www.informatik.uni-freiburg.de/~may/Mondial/`.

[2] G. Bhalotia, C. Nakhe, A. Hulgeri, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proc. of the 2002 Intl. Conf. on Data Engineering*, San Jose, California, February 2002.

[3] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proc. of the 1998 Intl. World Wide Web Conf.*, Brisbane, Australia, April 1998.

[4] H. W. Cain, R. Rajwar, M. Marden, and M. H. Lipasti. An architectural evaluation of Java TPC-W. In *Proc. of the 2001 Intl. Symp. on High-Performance Computer Architecture*, January 2001.

[5] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 471–480, Roma, Italy, September 2001.

[6] S. Dar, G. Entin, S. Geva, and E. Palmon. DTL's DataSpot: Database exploration using plain language. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 645–649, New York City, New York, August 1998.

[7] J. Dongarra, A. Lumsdaine, R. Pozo, and K. Remington. A sparse matrix library in C++ for high performance architectures. In *Proc. of the Second Object-Oriented Numerics Conference*, pages 214–218, 1994.

[8] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book*. Prentice Hall, Upper Saddle River, New Jersey, 2002.

[9] R. Goldman, N. Shivakumar, S. Venkatasubramanian, and H. Garcia-Molina. Proximity search in databases. In *Proc. of the 1998 Intl. Conf. on Very Large Data Bases*, pages 26–37, New York City, New York, August 1998.

[10] V. Hristidis and Y. Papakonstantinou. DISCOVER: Keyword search in relational databases. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 670–681, Hong Kong, China, August 2002.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd. The PageRank citation ranking: Bringing order to the Web. Technical report, Stanford University, November 1999.

[12] Transaction Processing Performance Council. TPC-W benchmark specification. `http://www.tpc.org/wspec.html`.

A Proof of Lemma 1

It suffices to consider node t_2. In G, let $R(t,S)$ be the TupleRank contribution to node t from edges other than those in the edge set S, namely,

$$R(t,S) = \sum_{e \in into(t)-S} \frac{w(e)\cdot \text{TupleRank}(from(e))}{w(from(e))}.$$

Let $R'(t,V)$ be the similar contribution in graph G' and $\text{TupleRank}'(t)$ the TupleRank for the node t in G'. According to the construction of the new graph, we have

$$w'(t_1) = w(t_1) - w(e_1) - w(e_2) + w'(e_3) = w(t_1).$$

Then

$$\begin{aligned}\text{TupleRank}'(t_2) &= R'(t_2,\{e_3\}) + \frac{w'(e_3)\cdot \text{TupleRank}'(t_1)}{w'(t_1)} \\ &= R'(t_2,\{e_3\}) + \frac{[w(e_1)+w(e_2)]\cdot \text{TupleRank}'(t_1)}{w(t_1)}.\end{aligned}$$

Therefore, all the TupleRank computation formulae remain the same in graph G'. The two graphs thus produce identical TupleRank values.

B Proof of Lemma 2

It suffices to consider the nodes q and t_i, $i = 1, \ldots, n$. We will use the notations introduced in the proof of Lemma 1. According to the definition of TupleRank and the construction of the new graph, we have

$$\text{TupleRank}(q) = \sum_{j=1}^{n} \frac{w_j \cdot \text{TupleRank}(t_j)}{w(t_j)}, \tag{14}$$

$$\text{TupleRank}(t_i) = R(t_i, \{e(t_q, t_i)\}) + \frac{w_i \cdot \text{TupleRank}(q)}{w(q)} \tag{15}$$

and

$$\text{TupleRank}'(t_i) = R'(t_i, \{e(t_j, t_i) \,|\, j = 1, \ldots, n\}) + \sum_{j=1}^{n} \frac{w_i \cdot w_j \cdot \text{TupleRank}'(t_j)}{w(q) \cdot w'(t_j)}. \tag{16}$$

Now let us verify that TupleRank(t) for all $t \in T \cup Q'$ satisfies the TupleRank formulae for the new graph. Substitute (14) into the right-hand side of (16), and we get

$$R'(t_i, \{e(t_j, t_i) \,|\, j = 1, \ldots, n\}) + \frac{w_i \cdot \text{TupleRank}(q)}{w(q)}.$$

It is easy to verify that $w'(t_i) = w(t_i)$. Therefore, $R'(t_i, \{e(t_j, t_i) \,|\, j = 1, \ldots, n\}) = R(t_i, \{e(t_q, t_i)\})$, and (16) remains valid with the old TupleRank values. Because the TupleRank solution to a graph is unique, the TupleRank values for $T \cup Q'$ in G' should be identical to those in G.