Queries over Virtual Nested Objects

Liang Chen
University of California, San Diego
jeffchen@cs.ucsd.edu

Yannis Papakonstantinou
University of California, San Diego
yannis@cs.ucsd.edu

ABSTRACT

We describe a system that allows the easy specification and efficient support of queries specified as a set of attribute/ predicate/value triplets over virtual nested objects constructed from relational databases. For example, the Internet Movie Database (imdb) provides virtual nested objects such as "movies" (each one containing a movie tuple, multiple actor tuples and more) and "actors" (containing multiple movies). Queries may contain both boolean and text/fuzzy predicates and may be directed to one or more virtual nested object sets. We define ranked query semantics that capture the common requirements that (1) individual tuples within a nested object may only satisfy a subset of all the predicates and (2) an object that contains a tuple that satisfies more than one predicates should (all other things being equal) be ranked higher than an object where such predicates are satisfied over multiple tuples. The system fully utilizes the existing indices in the relational databases and combines special purpose algorithms with database accesses. Experiments demonstrate that the obtained performance is significantly better than the performance obtained by fully deferring query evaluation to SQL queries.

1. INTRODUCTION

Relational databases usually consist of a set of flatten tables and a number of joins between them because of the database normalization. Such joins reflect some semantic relationships. For example, Figure 1 shows the relational schema of the Internet Movie Database (IMDB). The join between the MOVIES table and ACTORS table through ACRIN represents *starring in*, an actor plays in a movie. Similarly, joins between ACTOR_TRIVIA and ACTOR represents that a trivia belongs to an actor. Tuples from different tables that are connected through these semantic joins can construct some nested objects. In IMDB, a "movie" object contains its own information, title and year, and the information of actors in the cast, i.e. name, trivia. An "actor" object contains his first name and last name, as well as the information

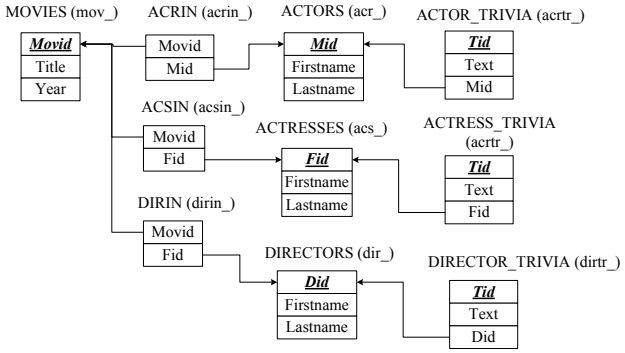


Figure 1: Relational Schema for IMDB

of the movies which he plays in.

There is a rich class of applications of querying nested objects in relational databases. Firstly, it is desirable to let users specify complex predicates nested within the objects. In IMDB database, when the user search the actor, he/she may forget actor's name, but only remember some trivia, which is nested within the actor. On the other hand, when the user search the movie, he/she may know not only the year of the movie, but also the details of the director. He/she definitely wants to put all the predicates within the movie object to confine the results. Secondly, users may need personalized nested objects over the same data set to satisfy their own searching needs. In DBLP, there is a number of ways to organize papers. For example, "people" object, grouping papers by authors, "topic" object, grouping papers by the research areas, "conference" object, grouping papers by the publication places, and "year" object, grouping papers by the publication time. For the users who want to search people and their research work, "people" object is much better; for the users who concentrate on papers on some specific areas, "topic" object is more suitable.

We consider the problem of evaluating queries over *virtual* nested objects in relational databases. The nested object is defined manually by specifying flatten tables and relationships (joins) between them. A query over the nested object is a set of boolean and keyword/fuzzy predicates on the atomic attributes within the object. Notice that nested objects are only conceptual and never fully materialized. We specifically emphasis *virtual* because there is usually a number of ways to organize the same data. Materializing one or fixed number of nested objects is insufficient to satisfy various users' needs, and meanwhile introduces information

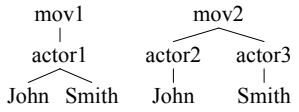


Figure 2: Two instances of the movie object

redundancies and difficulties of maintaining.

At first glance evaluating queries over virtual nested objects can be solved by the SQL query to reconstruct the nested objects with predicates. However, single SQL query fails to capture the ranking semantics shown in the following example.

EXAMPLE 1.1. *Consider the query that has two predicates: firstname="John" and lastname="Smith". Figure 2 shows two instances of the movie object. While mov1 is definitely the result, mov2 is also an answer to the query because actor tuples within mov2 also satisfy all the predicates. However, in mov1, the minimal number of actors to satisfy all the predicates is only one, whereas in mov2 this number increases to two. Intuitively, the the ranking of mov1 should be better than mov2, because it is more likely that the user looks for the movie with an actor whose name is "John Smith" than the movie with two actors whose names are "John" and "Smith" respectively.*

The above example is due to the many-to-many relationships in the relational database: a movie object may have multiple actors and predicates may be satisfied over a variable number of actors within it. More generally, ranked query semantics should capture the following two requirements: (1) individual tuples within a nested object may only satisfy a subset of all the predicates; (2) an object that contains a tuple that satisfies more than one predicates should (all other things being equal) be ranked higher than an object where such predicates are satisfied over multiple tuples.

In this paper, we present a system that allows the easy specification and efficient support of queries specified as a set of attribute/predicate/value triplets over virtual nested objects constructed from relational databases. We formalized the above ranking intuition as *Lowest Common Ancestor* (LCA). The system utilizes the existing indices within the relational databases and retrieves those tuples satisfying the predicates. Then join-based algorithm computes the LCAs for the predicates as well as their ranking scores, and further constructs satisfied objects. Our system is pipelined and as long as top K objects are generated, the execution terminates.

To our best knowledge, this is the first work to address the problem of evaluating ranked queries over virtual nested objects in relational databases. Key contributions of this paper ares summarized as follows:

- We formally define ranked query semantics that capture the common requirements of ranking nested objects: (1) individual tuples within a nested object may only satisfy a subset of all the predicates and (2) an object that contains a tuple that satisfies more than one predicates should (all other things being equal) be ranked higher than an object where such predicates are satisfied over multiple tuples.
- We propose a novel join-based LCA algorithm to compute LCAs for predicates in tree-structured nested objects. LCA is used in our system as an important ranking factor. Unlike the existing LCA algorithms in XML databases, the new algorithm guarantees that *lowest* LCAs for generated first, providing an efficient support for top K results.
- We propose a Top-K join algorithm specifically for our query semantics. Traditional SQL join returns the combinations of joined tuples. Instead, our query semantics requires the root of a nested object that can connects tuples satisfying the predicates. This difference results in a simplification of the algorithm complexity and more precise estimation of the upper bound of unseen results.
- We propose a pipelined architecture and implement it on top of PostgreSQL 8.3. We perform a detailed evaluation of the system with different parameters. Results show that the performance of our system is significantly better than fully deferring query evaluation to SQL queries.

The rest of the paper is organized as follows: Section 2 introduces the data model and query semantics. Section 3 discusses the ranking intuitions and formally defines the ranking metric with the semantic optimization. Section 4 shows the high level architecture of the system. Section 5 and Section 6 describes two core modules and algorithms of our system, which is experimentally evaluated in Section 7. Finally, Section 9 concludes the paper.

2. DATA MODEL AND QUERY SEMANTICS

Nested objects have many analogies to the nested tuples in nested relations, which is defined over nested relation schemes (NRSs) represented by the *scheme tree*[21]. In this paper, we adapt the scheme tree concept in our scenario to define the virtual nested objects over flatten tables.

Consider a relational database $\mathcal{D}$ that has relations $R_1, R_2, ...$ A *virtual scheme tree*, denoted by $\mathcal{T}$, is a labeled tree (V, E) such that

1. each leaf node n^A is labeled by an (atomic) attribute A.
2. each non-leaf node n^R is labeled by the primary key attribute(s) of a flatten relation R.
3. For each edge between a non-leaf node n^A and a leaf node n^R, A is the attribute of R; for each edge between two non-leaf nodes n^{R_i} and n^{R_j}, there is a join between R_i and R_j, i.e. $R_i \bowtie R_j$, which is specified manually.

In the paper, we simply use $R_i \bowtie R_j$ to denote the join between R_i and R_j, though there may be other relations "connecting" these two relations. Note that $R_i \bowtie R_j$ can be any joins beyond the primary key and foreign key relationships, as long as users feel there is some semantics relationship between R_i and R_j.

Figure 3 gives an example of the scheme tree of movie object according to the fragment of IMDB relational schema in Figure 1. In Figure 3, star nodes represents the many-to-one relationship, e.g. there may be multiple tuples of ACTOR in a movie object. Later in the paper, we will see that star nodes play an important role in ranking optimization. We assume that star node should be identified by users if it

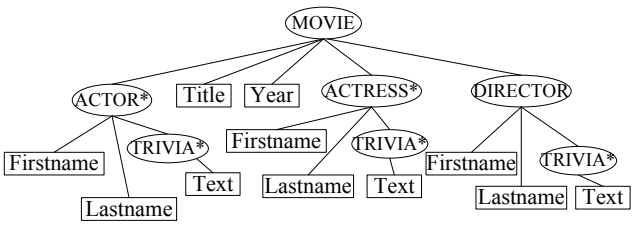


Figure 3: Virtual Scheme Tree for Movie Object

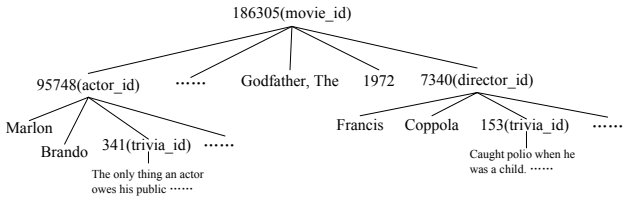


Figure 4: An instance of nested movie object according to $\mathcal{T}$

is a normal join or by the schema graph of the relational database if the join follows the primary key and foreign key relationships. Nevertheless, by default we set child to be the star node for every edge between two non-leaf nodes. It may only influence the performance, but not feasibility of our system.

An instance of $\mathcal{T}$ is a labeled tree and corresponds to a nested object. For the representation convenience and consistency with scheme tree, we also represent a nested object as the labeled tree, e.g. Figure 4. The label of a node in T is the value of the corresponding attribute in $\mathcal{T}$.

In the rest of this paper, we use $\mathcal{T}$ to denote the scheme tree and T to denote an instance (a nested object). n denotes a node in $\mathcal{T}$. Specifically, n^A denotes a leaf node where A is an (atomic) attribute and n^R denotes a non-leaf node where R is a flatten relation. v denotes a node in T, $\lambda(v)$ denotes the label of v in T. $parent(\cdot)$ operator gives the parent of a node. For a non-leaf node v_n, if $\lambda(v_n) \in \mathbf{dom}(R.PrimaryKey)$, we say v_n is an *alias* of n^R in T. For a star node n^R, there may be multiple aliases of n^R in a T. For example, a movie object can have arbitrary number of aliases of ACTOR.

A query over a scheme tree $\mathcal{T}$ is a set of predicates $P = \{p_1, p_2, ...\}$ each of which is on a single leaf of $\mathcal{T}$. The predicates can be either boolean or keyword ranked. The result of a query is a set of T's that satisfy the predicates.

1. For the boolean predicate p_b on the leaf node n^A of $\mathcal{T}$, an instance T satisfies p_b iff there exists a leaf v in T such that (1) $\lambda(v) \in \mathbf{dom}(A)$ and (2) $\lambda(v)$ satisfies p_b.

2. A keyword predicate p_k is a set of keywords $W = \{w_1, w_2, ...\}$ on leaf node n^A of $\mathcal{T}$. T satisfies p_k iff $\forall w \in W$, there exists a leaf node v in T such that (1) $\lambda(v) \in \mathbf{dom}(A)$ and (2) $\lambda(v)$ contains w.

Note that for the keyword predicate $W = \{w_1, w_2, ...\}$ and a satisfied instance T, there may be more than one leaves $v_1, v_2, ...$ in T each of which only contains a subset of keywords in W. In other words, keyword predicates can be thought as a set of conjunctive one-word-keyword predicate, i.e. $p_k = \bigwedge_i p_{k_i}$, $p_{k_i} = \{w_i\}$. In the following of this paper, we assume that each keyword predicate contains only one word.

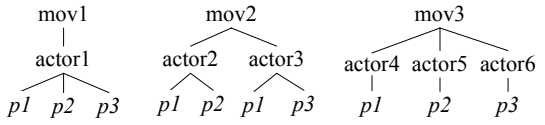


Figure 5: Different number of aliases satisfying the same predicates

3. RANKING

In this section, we show how to rank nested objects for a given query quantitatively. We first discuss intuitive and desirable constraints that ranking function is expected to satisfy. Then we introduce a semantic optimization by the constraints. Finally, we formally define our ranking metric.

The ranking of nested objects is motivated by the following observation: given a set of predicates on the descendants of n^R, if n^R is a star node, the minimal number of aliases of n^R that can satisfy all the predicates may be different for T's. Figure 5 shows an example: three predicates may be satisfied over one, two or three actors in the movie object. The intuition is that the *fewer* and *deeper* the aliases in T that satisfy all the predicates, the better the ranking score.

We formalize the above observation using LCA (*Lowest Common Ancestor*). Specifically, we borrow the concept of *Exclusive Lowest Common Ancestor* (ELCA) from [11, 32]. Let $v = LCA(v_1, \ldots, v_n)$ be the LCA for $v_1, \ldots, v_n$. For n sets of nodes $L_1, \ldots, L_n$, $LCA(L_1, \ldots, L_n) = \{v | v_1 \in L_1, \ldots, v_n \in L_n, v = LCA(v_1, \ldots, v_n)\}$, and v_i is called L_i's *occurrence* of v. Node v is called an ELCA of $L_1, \ldots, L_n$ iff $\exists v_i \in L_i, i = 1, \ldots, n$ such that (1) $v = LCA(v_1, \ldots, v_n)$ and (2) v_i is NOT a L_i's occurrence of $u \in LCA(L_1, \ldots, L_n)$ and u is the descendant of v. If nodes in L_i satisfy the predicate p_i, v is also called an ELCA of $P = \{p_1, \ldots, p_n\}$.

For a set of predicates $P = \{p_1, \ldots, p_n\}$ and an object instance T, let L_i^T denote leaves in T that satisfy p_i. $LCA(T; P)$ denotes the *lowest* ELCA(s) of $L_i^T, i = 1, \ldots, n$. $lev(\cdot)$ gives the depth of a node in T, and $f(T, P)$ gives the ranking score of T.

Constraint 1 Given two instances of $\mathcal{T}$, T_1 and T_2, $v_1 \in LCA(T_1; P)$, $v_2 \in LCA(T_2; P)$, $lev(v_1) > lev(v_2)$ then $f(T_1, P) > f(T_2, P)$.

Constraint 2 Given two instances T_1, T_2, assume $v_1 \in LCA(T_1; P)$, $v_2 \in LCA(T_2; P)$, and $lev(v_1) = lev(v_2)$. If $\forall P_i \subset P, |P| > 1, \sum_i lev(LCA(T_{v_1}; P_i)) > \sum_i lev(LCA(T_{v_2}; P_i))$ where T_{v_1} is the subtree of T_1 rooted at v_1 and T_{v_2} is the subtree of T_2 rooted at v_2, then $f(T_1, P) > f(T_2, P)$.

The intuition of Constraint 2 is that we hope to keep ELCA as low as possible, not only for P but also for subsets of P. ELCAs for the subsets of P give more fine granularity of ranking with respect to the number and depths of aliases. In Figure 5, there are three predicates on ACTOR. For the mov2 and mov3, although ELCAs for the three predicates are in the same level, intuitively, mov2 is better than mov3, because actor2 and actor3 in mov2 are "more qualified". In other words, while ELCAs for P in mov2 and mov3 are both roots, there are fewer aliases of ACTOR in mov2 that satisfy the subsets $P_1 = \{p_1, p_2\}$ and $P_2 = \{p_1, p_3\}$. Quantitatively, depths of ELCAs for $\{p_1, p_2\}$, $\{p_2, p_3\}$, $\{p_1, p_3\}$ in mov2 are 2,2,1 respectively, and in mov3

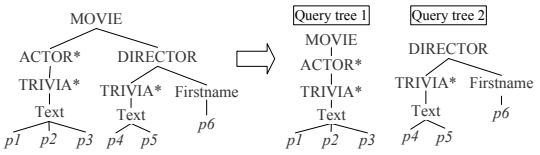


Figure 6: An example query and its query trees after partition

are 1,1,1. $\sum_i lev(LCA(T_{\mathsf{mov2}}; P_i)) = 2+2+1 > \sum_i lev(LCA(T_{\mathsf{mov3}}; P_i)) = 1+1+1$.

3.1 Semantic Optimization

The structure of the scheme tree $\mathcal{T}$ provides extra information that simplifies the ranking. Consider the query in Figure 6, $P = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. Let $P_1 = \{p_1, p_2, p_3\}$, $P_2 = \{p_4, p_5\}$ and $P_3 = \{p_6\}$.

- Since predicates in P_1 are under ACTOR and predicates in P_2 are under DIRECTOR, $\forall P_i \subseteq P$ (including P itself) that contains the predicates from both P_1 and P_2, for all the satisfied T's, $LCA(T; P_i)$ must be the root.
- For all the T's, the lower bound of $LCA(T; P_1)$ is the alias of Text, and the upper bound is the alias of MOVIE.
- For all the T's, the lower and upper bounds of $LCA(T; P_2)$ are both the alias of DIRECTOR, and cannot be higher. This is because: there is no star node from MOVIE to DIRECTOR. Thus, for any T, there cannot be two aliases of DIRECTOR each of which only satisfies a subset of P_2.

The structure of $\mathcal{T}$ provides the information of lower and upper bounds of ELCAs for P and P's subsets. For some subsets of P, their ELCAs are in the fixed level for all the T's. We only need to concentrate on those subsets whose ELCAs may be different in T's for the ranking purpose. To incorporate this idea, we propose the concept of *partition node*. A partition node n^P is a star node in $\mathcal{T}$ such that there is no other star node along the path between the root and n^P. P is partitioned into a set of subsets $P_i, i = 1, \ldots, m$ such that

1. $P_i, i = 1, \ldots, m$ are non-overlapping partitions of P.
2. For each P_i, there exists a partition node n^{P_i} in T such that $\forall p_k \in P_i$, p_k is on the descendant of n^{P_i}.

PROPOSITION 3.1. *Given a set of predicates P over $\mathcal{T}$ and its partitions, if $P_i \subset P$ contains predicates from more than one partitions, ELCAs for P_i are in the fixed level in all T's.*

For the query in Figure 6, P is partitioned into two subsets $P_1 = \{p_1, p_2, p_3\}$ and $P_2 = \{p_4, p_5\}$ by the partition nodes. For the predicates left after the partition, i.e. p_6, they are treated as filter predicates, because for all the satisfied T's, there is only one alias in T that satisfies p_6.

In summary, for the query that consists of more than one partitions, ELCAs for all the predicates is not an effective and efficient ranking metric. Instead, ELCAs for each partition should be considered individually. The final ranking score should be the combination of scores for all the partitions. In this paper, we simply use the summation. That is: $f(T, P) = \sum_{i=1}^{m} f(T, P_i)$ where $P_i, i = 1, \ldots, m$ are partitions of P.

3.2 Ranking Score

To incorporate all the above discussions, we define *Ranking-Triple Tuple* $(lev, slev, score)$ as the ranking score of T for P_i. $lev = lev(v)$ where $v = LCA(T; P_i)$, $slev = \sum_k w_k \times lev(LCA(T_v; P_{i_k}))$, $P_{i_k} \subset P_i, |P_{i_k}| > 1$ where w_k is the weight of subset P_{i_k}. w_k is used to give preferences to some subsets, e.g. subsets with larger size. In this paper, we let $w_k = 1$ for all P_{i_k}'s. *score* is the accumulative IR ranking score given by the keyword predicates on textual leaves. Only scores of textual nodes in the subtree T_v is counted. Intuitively, *lev* and *slev* reflect "tightness" of predicates in T, and *score* reflects textual relevance.

Using absolute depths for *lev* and *slev* unfairly favors those partitions whose predicates are deeply nested just because some part of the scheme tree has more nesting than another. To remedy this problem, we normalize *lev* and *slev* as follows:

$$lev = \frac{lev(v)}{lev(n)}, \quad v = LCA(T; P_i), n = LCA(\mathcal{T}; P_i)$$

$$slev = \frac{\sum_i lev(LCA(T_v; P_{i_k}))}{\sum_i lev(LCA(\mathcal{T}_n; P_{i_k}))}, \quad P_{i_k} \subset P_i, |P_{i_k}| > 1$$

$n = LCA(\mathcal{T}; P_i)$ is the LCA for P_i in the scheme tree, and is also the lower bound of the ELCAs in all T's. Similarly, $slev = \sum_i lev(LCA(\mathcal{T}_n; P_{i_k}))$ is the sum of depths of LCAs for P_{i_k}'s in subtree $\mathcal{T}_n$. (In the following of this paper, ranking-triple tuples are represented without normalization for the convenience of algorithm discussions.)

The final ranking score of T for all the predicates is given by

$$f(T, P) = (\sum_{i=1}^{m} lev_i, \sum_{i=1}^{m} slev_i, \sum_{i=1}^{m} score_i)$$

For any two ranking-triple tuples $rt_1 = (lev_1, slev_1, score_1)$ and $rt_2 = (lev_2, slev_2, score_2)$, the order of them is given by the following: if $lev_1 > lev_2$, then $rt_1 > rt_2$. If $lev_1 = lev_2$, $slev_1 > slev_2$, then $rt_1 > rt_2$. Similarly, *score*'s are only compared when *lev* and *slev* are the same.

3.3 Algorithmic Perspective

In practice, it is hard to compute the ranking-triple tuple of T for P_i directly. Instead, since we have efficient algorithm (as described later) to compute ELCAs of P_i, each ELCA in T (no matter if it is lowest) is considered as the lowest one, and its ranking-triple tuple is computed. If there are more than one ELCAs in T for P_i, the ranking-triple tuple of T is the value of the *minimal* ELCA. For example, in Figure 5, if there is another movie mov4 which has actors actor1, actor2, and actor3 at the same time, then actor1 and mov4 are both ELCAs for P_i. In such case, ranking-triple tuple is determined by the actor1 because if it is the lowest one. More precisely, if T contains two ELCAs $v_1(lev_1, slev_1, score_1)$ and $v_2(lev_2, slev_2, score_2)$, then the ranking-triple tuple of T for P_i is: (1) $(lev_1, slev_1, score_1)$, if (i) $lev_1 > lev_2$ or (ii) $lev_1 = lev_2$ and $slev_1 > slev_2$; (2) $(lev_1, slev_1, score_1 + score_2)$, if $lev_1 = lev_2, slev_1 = slev_2$.

4. SYSTEM OVERVIEW

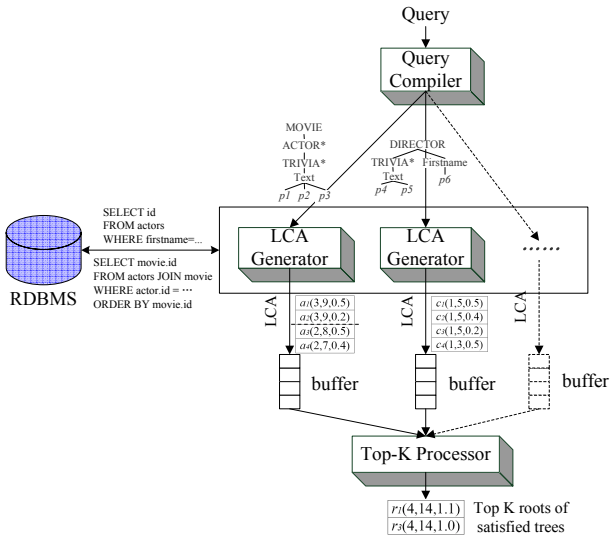


Figure 7: System Architecture

In this section, we give an overview of our system.The high level picture of system is shown in Figure 7. Since all T's are only conceptual and not materialized, the query evaluation can be viewed as the process of reconstructing satisfied T's in the descending order by their ranking scores. T's are constructed bottom up: first, leaves that satisfy the individual predicates are evaluated through the SQL engine. Then, ELCAs for each partition is computed. Finally, root aliases that connect to at least one ELCA for each partition are the roots of satisfied T's. In the following of this section, we walk through all the modules of the system and formally define their inputs and outputs.

The Query Compiler Module inputs all the predicates P, and outputs one or more *query trees*. A query tree contains the predicates of one partition and filter predicates on the corresponding leaves. Figure 6 shows the two query trees after compilation. Algorithm 1 gives the pseudo code of partitioning. Given the scheme tree $\mathcal{T}$ and a set of predicates P, the algorithm generates the partition of P.

For each query tree, a LCA Generator (LCAG) is created. LCAG takes the query tree as input and outputs a set of ELCAs with their ranking-triple tuples. LCAG first sends a SQL query that encapsulates the predicate to the relational engine and gets a set of leaves of T's that satisfy the predicate. Then, LCAG traverses the query tree bottom-up, performs join-based LCA algorithm to generate ELCAs, and computes their ranking-triple tuples progressively. Join-based algorithm guarantees that lowest ELCAs are generated first, an important feature that Top-K Processor requires.

ELCAs associated with their ranking-triple tuples are fed into a buffer between the LCAG and Top-K Processor. Recall that T's ranking score is the sum of its ranking-triple tuples for all the partitions, satisfying the monotonicity. Thus, there is a potential to exploit top K algorithm to avoid scanning complete ELCAs for all the partitions. Taking buffers as input, Top-K Processor outputs roots of T's such that they connect to at least one ELCA from each buffer. We propose a threshold algorithm to generate top K roots efficiently.

Algorithm 1 Partition Algorithm

```
1: Input: Schema tree T
2: Output: Partitions of P
3:
4: i ← 0
5: Push the root of T into queue Q
6: while Q is not empty do
7:   v ←Q.pop()
8:   if v is a star node then
9:     if some predicates are v's descendants then
10:      create a new partition P_i
11:      add all the predicates under v to P_i
12:      i ← i + 1
13:    end if
14:  else
15:    Push all the child nodes of v into Q
16:  end if
17: end while
18: return P_i, i = 1, ..., m
```

Essentially, LCAGs and Top-K Processor can be thought as the producer-consumer model, and can execute simultaneously. As long as TopK Processor outputs top K results, all the LCAGs can also stop.

5. LCA GENERATOR MODULE

In this section, we elaborate our join-based algorithm that generates ELCAs for a partition and show how to compute their ranking-triple tuples progressively. The problem is defined as follows: given n predicates $P = \{p_1, ..., p_n\}$ in a partition, L_i denotes a list of leaves of T's that satisfy p_i. The problem is to compute $ELCA(L_1, ..., L_n)$ and their ranking-triple tuples.

5.1 Join-based LCA algorithm

Computing $ELCA(L_1, ..., L_n)$ has attracted much attention in XML keyword search[11, 32]. However, two issues make the existing algorithms infeasible in our scenario. Firstly, existing algorithms reply on the node encoding in XML tree, e.g. Dewey Id or pre-order/post-order. Node encoding is not available until T's are materialized. Secondly, generated ELCAs follow the XML document order, which does not provide an efficient support for top-K processing. In the existing algorithms, nodes containing one keyword are sorted by their encodings. Then nodes are scanned sequentially and either stack or index is used to compute ELCAs. According to existing encodings, e.g. Dewey ID or pre-order/post-order, this sequence follows the document order. Thus, the sequence of generated ELCAs also follows the document order: in Figure 8, ELCAs in the Subtree 1 are first generated, and then Subtree 2, and so on. Since ELCAs in the rightmost subtree can be in any level, we have to wait until all the ELCAs are generated in order to get *lowest* ones. In other words, Top-K Processor has to be blocked until all the LCAGs finish execution, because Top-K processing requires the input be ordered.

The key idea of join-based algorithm is that in order to guarantee *lowest* ELCAs are generated first, all the nodes are processed bottom up, level by level, and all the ELCAs in the same level are generated at one time. Consider two

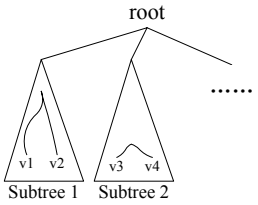


Figure 8: Subtrees under the root

nodes v_1 and v_2 in L_1 and L_2 respectively. If $parent(v_1)$, $parent(v_2)$ are aliases of n^R, then $parent(v_1) \cap parent(v_2)$ is the common ancestor(s) for v_1 and v_2. Recall that labels of two nodes' parents are the primary keys of R, which are the identities of parents. Notice that in relational database, one node may refers to multiple parents, e.g. one actor node can have multiple movies as its parents.

More precisely, let $par(L_i) = \{parent(v)|v \in L_i\}$, and $par^k(L_i) = par(par^{k-1}(L_i))$. If nodes in $par^{x_i}(L_i)$, $i = 1 \ldots n$ are all aliases of n^R, then $S_1 = \bigcap_{i=1}^{n} par^{x_i}(L_i)$ is a set of ELCAs for P at the level k, where k is the depth of n^R in $\mathcal{T}$. In the next level upward, $S_2 = \bigcap_{i=i}^{n} par(par^{x_i}(L_i) - S_1)$ is another set of ELCAs for P at the level $k-1$. This process repeats until reaches the upper bound of ELCA for the partition. And we obtain the all ELCAs in an descending order of their depths.

Algorithm 2 Join-based LCA Algorithm

```
1: Input: L_{1}, L_{2}, L_{3} whose nodes are aliases of n^R
2: Output: RS_k { RS_k is a set of LCAs at level k}
3:
4: X = {{1},{2},{3},{1,2},{2,3},{1,3}}
5: L_{1,2} ← ∅, L_{1,3} ← ∅, L_{2,3} ← ∅
6: C_i denotes the cursor of L_i, i ∈ X
7: ū denotes the upper bound of LCA for p_1, p_2, p_3 in T
8: k ← lev(n^R)
9: while k >= lev(ū) do
10:   L_i' ← ∅, C_i ← 0, i ∈ X
11:   while some C_i has not reached the end do
12:     Find the minimal value v of L_i(C_i), i ∈ X
13:     X_0 ← ∅
14:     for i ∈ X do
15:       if L_i(C_i) == v then
16:         X_0 ← X_0 ∪ i
17:         C_i ← C_i + 1
18:       end if
19:     end for
20:     if X_0 == {1,2,3} then
21:       RS_k ← RS_k ∪ {v}
22:     else if k > depth(ū) then
23:       L'_{X_0} ← L'_{X_0} ∪ {v}
24:     end if
25:   end while
26:   for i ∈ X do
27:     L_i ← par(L_i')
28:   end for
29:   k ← k − 1
30: end while
31: return RS_k, k = 1, 2, ... if RS_k is not empty
```

par operator can be evaluated through SQL queries. Assume L is a list of nodes which are aliases of n^{R_i} and $parent(n^{R_i}) = n^{R_j}$. Then $par(L)$ is given by:

SELECT DISTINCT	R_j.id
FROM	R_i JOIN R_j
WHERE	R_i.id IN L
ORDER BY	R_j.id

Starting from the lowest level, multi-list sort-merge join algorithm computes ELCAs for P level by level. The pseudo code is shown in Algorithm 2 which only illustrates the case of 3 predicates, for the sake of easy understanding. More predicates are similar. Unlike the traditional sort-merge join algorithm, multi-list join algorithm not only computes the nodes shared by all the lists $L_i, i = 1, \ldots, n$, but also those shared by a subset of lists, $L_{x_1}, \ldots, L_{x_j}$ where $\{x_1, \ldots, x_j\} \subset \{1, \ldots, n\}$. These nodes are moved into new list $L_{\{x_1 \ldots x_j\}}$ as intermediate results. The reason of maintaining $L_{\{x_1 \ldots x_j\}}$ is that ranking-triple tuple not only involves ELCA for P, but also ELCAs for the subsets $P_i \subset P, |P_i| > 1$. Node $v_x \in L_{\{x_1 \ldots x_j\}}$ is the ELCA for the subset $P_j = \{p_{x_1}, \ldots, p_{x_j}\}$, and records information of ELCAs for P_j's subsets. If v_x's ancestor u_x joins other nodes at higher level and u_x is the ELCA for P, then depths of ELCAs for P_j and P_j's subsets in T_{u_x}, i.e. $lev(LCA(T_{u_x}; P_j))$ and $lev(LCA(T_{u_x}; P_{j_k})), P_{j_k} \subset P_j$, can be given directly by v_x, which avoids extra computing.

For the partition whose predicates are on different leaves of $\mathcal{T}$, e.g. query tree 2 in Figure 6, we start from predicates in the lowest level and traverse the query tree bottom up. If in current level more than one lists have nodes that are aliases of the same n^R, compute ELCAs among corresponding lists and keep them as intermediate results. In the query tree 2 of Figure 6, we start from lowest predicates p_4 and p_5. At TRIVIA level, aliases of TRIVIA may be ELCAs for p_4 and p_5. So join is performed on lists L_4 and L_5, and ELCAs for $\{p_4, p_5\}$ are moved into $L_{\{4,5\}}$. In the next level upward, aliases of DIRECTOR can be ELCAs for the three predicates, so the join is performed on the four lists L_4, L_5, L_6 and $L_{\{4,5\}}$. Since DIRECTOR is the upper bound of this partition, we do not need to keep ELCAs for subsets of $\{p_4, p_5, p_5\}$ (line 22 in Algorithm 2).

It must be explained that although join-based algorithm generates ELCAs in the descending order of their depths, ELCAs in the same level are not ordered by their ranking-triple tuples automatically. Rather, they are ordered by the node labels (i.e. primary keys of a flatten table), because of the sort-merge join. Here we insert a block point for the LCAG: ELCAs are not fed into the buffer until all the ELCAs in the same level are generated and sorted. The reason for the block point will become more clear in Section 6.

5.2 Computing Ranking-Triple Tuple Progressively

In implementation, each list is attached a tag and corresponds to a subset $P_i \subset P$. Nodes in the lists are associated with three variables: (1) lev_i, depth of the ELCA for P_i, (2) $levSet_i$, a set of depths of ELCAs for P_i's subsets, i.e. $P_{i_k} \subset P_i, |P_{i_k}| > 1$, and (3) $score_i$, the accumulative IR relevance score. For the *par* operator, three variables of a node are passed directly to its parent(s).

Consider two lists $L_{P_i}, P_i \subset P$ and $L_{P_j}, P_j \subset P$ whose nodes are aliases of n^R in level k. Let $P_{ij} = P_i \cup P_j$. Assume $v_i \in L_{P_i}, v_j \in L_{P_j}$, and $\lambda(v_i) = \lambda(v_j)$. Then v_i (and v_j) is

the ELCA for P_{ij}. So v_i (and v_j) is moved from L_{P_i} (and L_{P_j}) to $L_{P_{ij}}$. Its new variables lev_{ij}, $levSet_{ij}$ and $score_{ij}$ in $L_{P_{ij}}$ should be updated as follows:

1. If $P_i \subset P_j$ (or $P_j \subset P_i$), then $lev_{ij} = lev_j$ (or $lev_{ij} = lev_i$), because $P_{ij} = P_j$ (or $P_{ij} = P_i$) and v_j (or v_i) is already the ancestor of the ELCA for P_{ij}; else, $lev_{ij} = k$.

2. $levSet$ now should contain depths of ELCAs for the subsets of P_{ij}. $\forall P^{'} \subset P_{ij}, |P^{'}| > 1$, let $levSet(P^{'})$ denote the depth of ELCA for $P^{'}$. Then

   (a) if either $P^{'} \subset P_i$ or $P^{'} \subset P_j$, but not both, then $levSet_{ij}(P^{'}) = levSet_i(P^{'})$ or $levSet_{ij}(P^{'}) = levSet_j(P^{'})$.

   (b) if both $P^{'} \subset P_i$ and $P^{'} \subset P_j$, then $levSet_{ij}(P^{'}) = \min\left(levSet_i(P^{'}), levSet_j(P^{'})\right)$.

   (c) if $P^{'} = P_i$, (1) if $P_i \subset P_j$, $levSet_{ij}(P^{'}) = min\left(lev_i, levSet_j(P^{'})\right)$; (2) else, $levSet_{ij}(P^{'}) = lev_i$. Similarly, if $P^{'} = P_j$, (1) if $P_j \subset P_i$, $levSet_{ij}(P^{'}) = min\left(lev_j, levSet_i(P^{'})\right)$; (2) else, $levSet_{ij}(P^{'}) = lev_j$.

   (d) if neither $P^{'} \subset P_i$ nor $P^{'} \subset P_j$, then $levSet_{ij}(P^{'}) = k$.

3. $score_{ij} = score_i + score_j$.

When join algorithm is performed level by level, P_{ij} keeps growing, and finally $P_{ij} = P$. Then node $v \in L_{P_{ij}}$ is the ELCA for P and v's ranking-triple tuple can be given directly by its three variables:

$$\begin{aligned} lev &= lev_{ij} \\ slev &= \sum_{P'} levSet(P^{'}) \qquad P^{'} \subset P, |P^{'}| > 1 \\ score &= score_{ij} \end{aligned}$$

5.3 Execution Example

Now we walk through the algorithm by an example, showing how to compute the ELCAs and their ranking-triple tuples. The sample data is shown in Table 1. Consider the query that has three keyword predicates, $p_1 = \{$"cannes"$\}$, $p_2 = \{$"venice"$\}$, $p_3 = \{$"2000"$\}$, on the Text of TRIVIA of ACTOR, as the query tree 1 in Figure 6.

LCAG first sends three SQL queries to the relational engine and gets three lists of satisfied leaves with their IR ranking scores, as shown in Figure 9(a). Three values in the parentheses represent lev, $levSet$ and $score$ respectively ("[]" denotes the empty set).

Since all the nodes in three lists are aliases of Text of ACTOR, multi-list join is performed on $L_{\{1\}}, L_{\{2\}}, L_{\{3\}}$. t_2, t_3 can be joined between $L_{\{1\}}$ and $L_{\{3\}}$, so they are removed from these two lists and put into a new list $L_{\{13\}}$. Similarly, t_1, t_6 are removed from $L_{\{2\}}$ and $L_{\{3\}}$ and put into $L_{\{23\}}$. Now there are four lists as shown in Figure 9(b). Notice that nodes in $L_{\{13\}}$ and $L_{\{23\}}$ are ELCAs for $\{p_1, p_3\}$ and $\{p_2, p_3\}$ respectively. The first value in the parentheses is the depth of the ELCA, and the third value is the accumulative IR score.

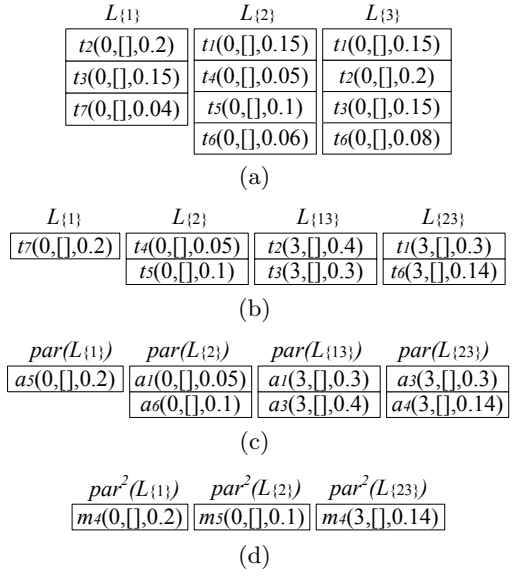


Figure 9: List Status in Execution Example

Next, par operator is applied on the four lists to get their parents. lev, $levSet$ and $score$ are passed directly to their parents. $par(L_{\{1\}})$, $par(L_{\{2\}})$, $par(L_{\{13\}})$ and $par(L_{\{23\}})$ are shown in Figure 9(c). Nodes in the four lists are aliases of ACTOR and join algorithm is performed again. a_1 can be joined between $L_{\{2\}}$ and $L_{\{13\}}$ and thus is moved into L_{123}. Its new lev is the depth of the current level, so $lev = 2$. $levSet$ in L_{123} should contain depths of ELCAs for subsets $\{12\}, \{23\}, \{13\}$. ELCAs for $\{13\}$ can be given by a_1 in $par(L_{\{13\}})$. That is: $levSet(\{13\}) = lev_{\{13\}} = 3$. $\{12\}$ and $\{23\}$ are subsets contained neither in $\{13\}$ nor $\{2\}$, so $levSet(\{12\}) = 2$, $levSet(\{23\}) = 2$. New $score$ is the sum of $score$'s from two joined nodes: $score = 0.05 + 0.3 = 0.35$. a_3 can also be joined between $par(L_{\{23\}})$ and $par(L_{\{13\}})$ and thus is moved into L_{123}. Its lev, $levSet$ is updated in a similar way: $lev = 2$, $levSet(\{13\}) = 3$, $levSet(\{23\}) = 3$, $levSet(\{12\}) = 2$, $score = 0.3 + 0.4 = 0.7$. So $L_{\{123\}} = \{a_1(2, [2, 3, 2], 0.35), a_3(2, [2, 3, 3], 0.7)\}$. Since nodes in $L_{\{123\}}$ are ELCAs for all the predicates, a_1 and a_3 are fed into the buffer between LCAG and Top-K Processor. The ranking-triple tuples of a_1 and a_3 can be computed directly from three variables: $lev_{a_1} = 2$, $slev_{a_1} = 2+3+2 = 7$, $score_{a_1} = 0.35$; $lev_{a_3} = 2$, $slev_{a_3} = 2+3+3 = 8$, $score_{a_3} = 0.7$. Notice that within this level, a_1 is generated first, although its ranking-triple tuple is less than a_3.

In the next step, par operator is applied again on the nodes left in the lists, as shown in Figure 9(d). Now nodes in the lists are aliases of MOVIE. m_4 can be joined between $L_{\{1\}}$ and $L_{\{23\}}$, and thus is moved into the $L_{\{123\}}$ and further to the buffer. Its ranking-triple is: $lev_{m_1} = 1$, $slev_{m_1} = 1 + 3 + 1 = 5$, $score_{m_1} = 0.34$. Since MOVIE is the root level of the nested object, the algorithm terminates. All the ELCAs have been fed into the buffer by the order of the ELCAs' depths.

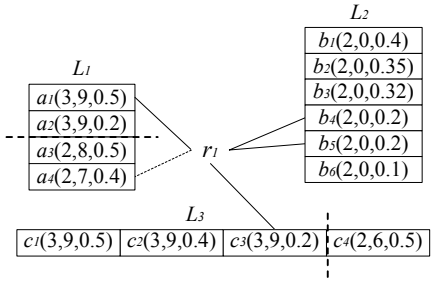
6. TOP-K PROCESSOR MODULE

Top-K Processor takes lists of ELCAs as input, and outputs those T's whose roots can connect to at least one ELCA

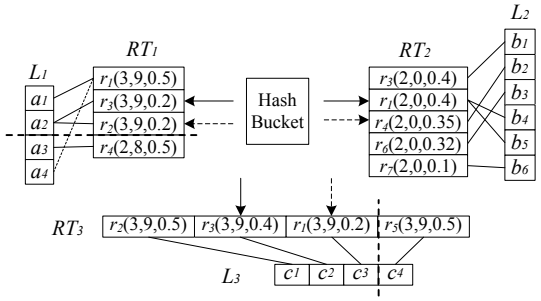
Table 1: Sample Data

Movid	Mid
m_1	a_1
m_1	a_2
m_2	a_3
m_3	a_3
m_4	a_4
m_4	a_5
m_5	a_6

Tid	Text	Mid
t_1	2000: Received the Career Golden Lion at the Venice Film Festival.	a_3
t_2	Member of the jury at the Cannes Film Festival in 2000.	a_3
t_3	President of the famous film festival in Cannes since 2000.	a_1
t_4	... best "first time" young actor/actress at the Venice Film Festival.	a_1
t_5	Member of the jury at the Venice Film Festival in 1991.	a_6
t_6	... Olivier Theatre Award in 2000 for ... in The Merchant of Venice ...	a_4
t_7	... 6 times as Best Director and received 3 nominations from Cannes ...	a_5



(a) Nodes from three lists joined through r_1



(b) Snapshot of Top-K algorithm execution

Figure 10: Running Example of Top-K Module

from each list. In database, connecting the root to an ELCA is a join operation. Thus, essentially, finding top K T's is the top K join problem[18, 14]. However, given the semantic difference between SQL join and our query, there is a big opportunity for improvement. In SQL join, the result of n-relation join is a set of combinations of tuples from n relations. Figure 10(a) shows a join among nodes from L_1, L_2 and L_3. By the SQL join semantics, there are 4 results: $(a_1, b_4, c_3), (a_1, b_5, c_3), (a_4, b_4, c_3), (a_4, b_5, c_3)$. However, the 4 combinations correspond to one tree rooted at r_1 which is our expected result. Traditional join introduces extra complexity if we concentrate the whole tree rather than the combinations of its leaves. Those top-K join algorithms that reply on the SQL join semantics is not optimal for the tree pattern result.

An important observation is that for the tree pattern join, all the nodes connect to a central node r, an alias of the root of $\mathcal{T}$. Thus, we maintain the root aliases as intermediate results and those roots that join to one ELCA from each list are the final results. Let L_i denote the buffer fed by the LCAG, RT_i denote the list of roots of ELCAs in L_i, and r^i denote the root r in RT_i. Then join among $RT_i, \ldots, RT_m$ generates roots of satisfied T's.

The requirement of top-K algorithm is that RT_i is ordered by the roots' scores so that results with highest scores can be generated first. Roots in RT_i inherit ranking-triple tuples from their descendants in L_i. If r^i in RT_i has more than one descendants in L_i (i.e. T has more than one ELCAs for partition i), the score of r^i is determined only by those ELCAs whose $(lev, slev)$ is the minimal (see Section 3.3). In other words, if r^i is newly derived by a node in L_i, only nodes in the same level in L_i may affect its score and position in RT_i. This is the very reason we insert a block point in LCAG module: when all the ELCAs in the same level are fed into the buffer, their corresponding roots are derived, sorted and put into RT_i. Then all the nodes fed after the L_i's block point cannot change the scores of existing roots in RT_i anymore.

EXAMPLE 6.1. *In Figure 10(a), value in the parentheses is the ranking-triple tuple of that node and doted lines denote block points. Although both a_1 and a_4 connect to r_1^1, the score of r_1^1 in RT_1 is only determined by a_1, i.e. $(3, 9, 0.5)$, because a_1 is in lower level. For the r_1^2 in RT_2, it is connected by b_4 and b_5 which have the same $(lev, slev)$. So the score for r_1^2 is $(2, 0, 0.4)$. Furthermore, as long as r_1^1, r_3^1, r_2^1 are fed into the RT_1, their positions in RT_1 are fixed and cannot change by a_4.*

Given a set of ordered $RT_i, i = 1, \ldots, m$, the algorithm works as follows: (1) Maintain a cursor for $RT_i, i = 1, \ldots, m$, and let t_i be the score of the root right after the cursor in RT_i. Each time retrieve one root r_k^i from RT_i. RT_i is chosen in a round-robin way before the number of roots in result set is less than K. After that, RT_i whose t_i is minimal is chosen. (2) Put r_k^i into the hash bucket. Let r^0 denote a root in the bucket. If there is a matched root r_k^0 in the bucket, add the score of r_k^i to the r_k^0. If r_k^0 has been matched $m-1$ times (there is no match when put into the bucket first time), move it from the bucket to the result set.

The key of threshold top-K algorithm is to estimate the upper bound of the scores of unseen results so that existing results whose scores are greater than the upper bound can be outputted without blocking.

- For roots that have not been seen in any RT, their upper bound can be estimated as $\sum_i^m t_i$.
- Roots in the bucket can be grouped into $2^m - 2$ groups $B_S, S \subset \{1, \ldots, m\}$. All the roots in B_S have been seen in $RT_j, j \in S$. Let o_S denote the maximum score of roots in B_S. Then the upper bound of roots in B_S is estimated as $o_S + \sum_{j \notin S} t_j$. The upper bound of roots in the bucket is: $MAX_{S \subset \{1,\ldots,m\}} \left(o_S + \sum_{j \notin S} t_j\right)$.

Since $o_S + \sum_{j \notin S} t_j \geq \sum_{i \in S} t_i + \sum_{j \notin S} t_j = \sum_i^m t_i$, we only need to consider the roots in the bucket. Therefore, the upper bound of unseen results is estimated by $MAX_{S \subset \{1,\ldots,m\}} \left(o_S + \sum_{j \notin S} t_j\right)$.

EXAMPLE 6.2. *Figure 10(b) shows a snapshot of an execution of the top-K algorithm. Scores of nodes in L_i are given in Figure 10(a). Solid arrows denote current positions of cursors. Two roots in each RT have been seen and put into the bucket. r_3 appears in all the RT's and is put into the result set. Let r_i^* denote the r_i in the result set. The score of r_3^* is $r_3^1.score + r_3^2.score + r_3^3.score = (3,9,0.2) + (2,0,0.4) + (3,9,0.4) = (8,18,1.0)$. Within the bucket, $B_{\{12\}} = \{r_1^0\}$, $B_{\{3\}} = \{r_2^0\}$. Current score of r_1^0 is $r_1^1.score + r_1^2.score = (3,9,0.5) + (2,0,0.4) = (5,9,0.9)$, and r_2^0 is $c_1.score = (3,9,0.5)$. $t_1 = r_2^1.score = (3,9,0.2)$, $t_2 = r_4^2.score = (2,0,0.35)$ and $t_3 = r_1^3.score = (3,9,0.2)$. So the upper bound of $B_{\{12\}}$ is estimated as $r_1^0.score + t_3 = (5,9,0.9) + (3,9,0.2) = (8,18,1.1)$ and upper bound of $B_{\{3\}}$ is $r_2^0.score + t_1 + t_2 = (3,9,0.5) + (3,9,0.2) + (2,0,0.35) = (8,18,1.05)$. Thus, the upper bound of all the unseen results is (8,18,1.1) which is greater than the score of r_3^*. So r_3^* has to be blocked. In the next iteration, r_2^1, r_4^2 and r_1^3 are put into the bucket. r_1^0 is now moved into the result set. Its score is $(8,18,1.1)$. Within the bucket, r_2^0 is moved from $B_{\{3\}}$ to $B_{\{13\}}$ and its score is added by $r_2^1.score$. $B_{\{12\}}$ is empty, and $B_{\{2\}} = \{r_4^0\}$. Now $t_1 = r_5^1.score = (2,8,0.5)$, $t_2 = r_6^2.score = (2,0,0.32)$ and $t_3 = r_4^3.score = (2,6,0.5)$. So the upper bound of $B_{\{13\}}$ is estimated as $r_2.score + t_2 = (8,18,1.02)$ and upper bound of $B_{\{2\}}$ is estimated as $r_4^0.score + t_1 + t_3 = (6,13,1.62)$. Thus the upper bound of unseen results is $(8,18,1.02)$, which is greater than r_3^* but less than r_1^*. Hence r_1^* can be outputted and r_3^* continues to be blocked. In the next iteration, when the cursor of RT_2 moves to r_6^2, $t_2 = r_7^2 = (2,0,0.1)$. The estimation of r_2^0's upper bound is updated as $(8,18,0.8)$. So the r_3^* can be outputted.*

One thing worth to be mentioned is that our algorithm provides a tighter upper bound estimation. Existing top-K join algorithms estimate the upper bound as: $max_i\left(t_i + \sum_{j\neq i} t_j^1\right)$ where t_j^1 denotes the score of the maximum root in RT_j. $\forall S \subset \{1,\ldots,m\}$, $\left(o_s + \sum_{j\notin S} t_j\right) \leq \left(\sum_{j\in S} t_j^1 + \sum_{j\notin S} t_j\right) \leq \left(t_k + \sum_{j\neq k} t_j^1\right)$ where $k \notin S$. For example, when cursors point to the third positions, the estimated upper bound would be $r_1^1 + r_6^2 + r_2^3 = (8,18,1.32)$. Both r_1^* and r_3^* needs to be blocked. The reason for the tighter upper bound estimation is that considering the tree pattern result, we maintain roots directly. For the roots that *partially* connect to nodes in some lists L_i's, only scores from those unconnected lists are estimated.

7. EXPERIMENTS

7.1 Experiment Setup

In the experiments, we use the IMDB data set[1]. Original data is in text files and is converted into relational tables. Database schema is similar to Figure 1 except that we also include QUOTE and BIOGRAPHY of ACTOR, ACTRESS and DIRECTOR. The total size of all the relations is around 450M. All the experiments are performed using PostgreSQL 8.3 on a Debian 2.40GHz PC with 1G memory. Algorithms in the paper are implemented in Java, and connect to the database through JDBC. Indices are created

[1]http://www.imdb.com/interfaces/

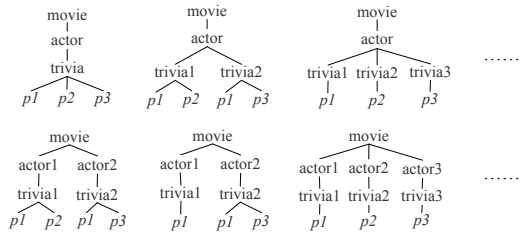


Figure 11: Possible Patterns of SQL Queries

on all attributes that can be queried. Boolean predicates are evaluated on the B-trees as normal SQL query. Keyword predicates are evaluated through the full-text module of PostgreSQL 8.3. Full-text module of PostgreSQL 8.3 retrieves tuples containing the keyword with their IR relevance scores.

For the join-based LCA algorithm, *par* operator can be evaluated through a SQL query, as mentioned in Section 5. However, in practice, this process is very slow due to the overhead introduced by JDBC. To overcome this shortcoming, we re-implement the child-parent relationship on B-trees residing directly on the file system. In other words, we re-store tables ACRIN, ACSIN and DIRIN outside the database. So *par* operator is evaluated directly on those disk-resident B-trees, avoiding the overhead of database transaction and JDBC.

7.2 Baseline

For the purpose of performance comparison, we present the baseline approach in this section. Given a nested scheme tree, SQL query is able to reconstruct those satisfied nested objects from flat tables using join. However, single SQL query fails to incorporate ranking semantics. As we saw in Section 3, ranking opportunity comes from the star node in $\mathcal{T}$: since there can be arbitrary number of aliases in T's, there are a number of possible relationships between predicates in different T's. Single SQL query can only reflect one pattern of predicate relationships. Thus, a straightforward approach to incorporate the ranking is issuing multiple SQL queries: one SQL query for each possible pattern. SQL queries corresponding to tighter relationships are issued first. Figure 11 shows some (but not all) possible patterns for the query tree 1 in Figure 6 (Text node is skipped).

If the first few queries can generate enough results (top K), this process can terminate. However, it also faces the potential danger that there is no result for "tight patterns", and outputting is delayed until all the queries are issued. Furthermore, the number of possible patterns increases very fast w.r.t the number of predicates. Let's consider the simplest case where three predicates p_1, p_2, p_3 is on the ACTOR nested in a MOVIE object, as shown in Figure 5. Since there is only star node, ACTOR, enumerating all the patterns is simply the problem of finding *minimal covers*. A minimal cover of a set is a cover for which removal of any single member destroys the covering property. In Table 2, a_i denotes different actor aliases in a movie object, and a_i satisfy the predicates within the brackets after a_i. We can see for this query, number of patterns is the number of minimal covers, and the number of the aliases is the number of elements in a minimal cover.

For a more complex query that has multiple star nodes,

Table 2: Enumerations of all the patterns of three predicates on ACTOR

# of elements	Cover
1	$a_1[p_1, p_2, p_3]$
2	$a_1[p_1, p_2], a_2[p_3]$
	$a_1[p_2, p_3], a_2[p_2]$
	$a_1[p_1, p_3], a_2[p_3]$
	$a_1[p_1, p_2], a_2[p_2, p_3]$
	$a_1[p_1, p_2], a_2[p_1, p_3]$
	$a_1[p_2, p_3], a_2[p_1, p_3]$
3	$a_1[p_1], a_2[p_2], a_3[p_3]$

e.g. query tree 1 in Figure 6, this is a recursive process. In the query tree of Figure 6, there are two star nodes, ACTOR and TRIVIA. For the top star node ACTOR that has predicates P nested in it, there are N possible ways for split P in the ACTOR level where N is the number of the minimal covers of P. And for each minimal cover, there are k ACTOR aliases in the MOVIE object each of which satisfies an element (which is a subset $P_i \subset P$) of this cover. Within each alias of ACTOR, this process repeats to enumerate all the patterns that satisfy $P_i, P_i \subset P$: there are a number of minimal covers splitting P_i, and for each minimal cover there are k TRIVIA aliases to satisfy it. Number of minimal covers is already exponential[13]. Number of different query patterns can only be larger for the query with more star nodes because of the multiplication effect.

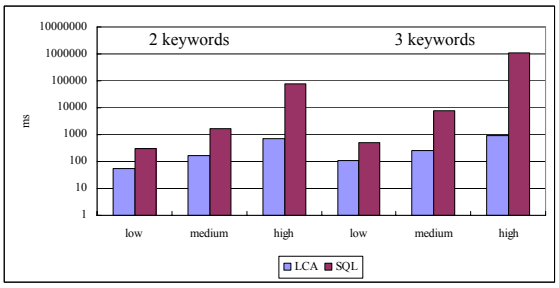
7.3 Search Performance

We focus on the query execution time in this paper. In the experiments, queries are sets of keyword predicates on TRIVIA. We choose keyword predicates because keyword frequencies reflect the selectivity of predicates directly. Queries are classified into three groups, namely *low*, corresponding to keywords with frequency lower than 100, *medium*, corresponding to keywords with frequency between 100 and 1000, and *high*, corresponding to keywords with frequency greater than 1000. Within each range, 40 queries are randomly selected. Values of each range in the following figures are average time over 40 queries, each repeated 10 times.

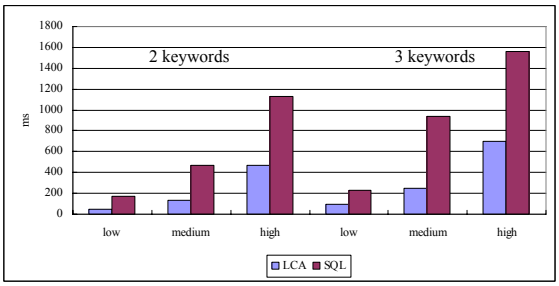
In Figure 12(a), queries contain two or three keyword predicates respectively in a single partition. *LCA* denotes our algorithms and *SQL* denotes the native approach. As we can see, to generate complete results in one partition, the performance of LCA algorithm is orders of magnitude better than that of naive approach, especially for less selective predicates. Figure 12(b) shows the performance of generating top 20 results. LCA algorithm is roughly 2-4 times faster than SQL approach. For single partition, top-K processing doesn't take effect. So the speedup in Figure 12(b) justifies the advantage of join-based LCA algorithm: lowest LCAs are generated first.

Experiment results for more than 3 predicates are not shown here. As mentioned above, the number of possible patterns increases extremely fast. Four predicates will result hundreds of patterns. Furthermore, Figure 12 already shows the trend that the speedup of our system for queries with more predicates is much larger than the queries with fewer predicates.

To validate the scalability of the LCA algorithm, we perform the experiments on varying number of keywords in one



(a) Complete results



(b) Top 20 Results

Figure 12: Predicates in One Partition, varying frequencies

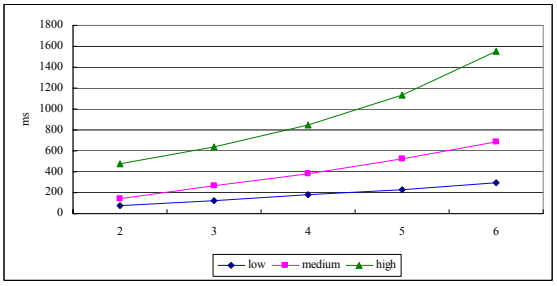


Figure 13: Top 20 results, varying number of predicates

partition. The result is shown in Figure 13. We can see that execution time increases nearly linearly w.r.t the number of predicates. As mentioned earlier, number of different tree patterns increases exponentially w.r.t number of predicates. Without prior knowledge of which patterns can generate results, SQL approach has to try all the patterns one by one until top K results are generated. This brings two serious problems: (1) atomic predicates have to be re-evaluated for each new query. (2) SQL engine wastes a lot of time on empty-result queries. In the experiments, we observe that the more predicates the query has, the less likely the tight patterns can generate top K queries, because of the selectivity. On the other hand, LCA algorithm computes patterns progressively (through join-based LCA algorithm) and only those patterns that are encountered in the data are maintained. In other words, the complexity is proportional to the data set, instead of the number of patterns.

Figure 14 shows the experiments on more than one partitions, evaluating the Top-K algorithm. For two partitions,

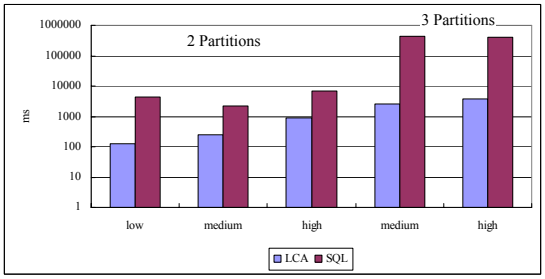


Figure 14: Top 20 results, 2 and 3 partitions

predicates are on TRIVIA's of ACTOR and ACTRESS, each of which has two keyword predicates. For three partitions, predicates are on TRIVIA's of ACTOR, ACTRESS and DIRECTOR, where the first two partitions have two keyword predicates and the last one has one keyword predicate. Experiment on three partitions is only performed on high and medium frequency, because a large amount of low frequency keyword predicates fail to generate any results. As shown in the figure, the speedup of our algorithms for top K result is larger than one partition. This is due to (1) the effect of the Top-K algorithm. Top-K processing does not take effect on only one partition; (2) expanded SQL query; (3) multiplication effect on number of patterns for multiple partitions. If the first partition has n patterns and the second partition has m patterns. Then altogether there are $m \times n$ patterns. In other words, it is very likely that more time is wasted on empty-result queries. A good demonstration of this is that in Figure 14, execution time for some lower frequency predicates are even longer than that of higher frequency predicates.

8. RELATED WORK

There has been much work on keyword search over structured data. DISCOVER[17], DBXplorer[1] and BANKS[6] are first three systems presented to support keyword search in relational databases. Their query semantics are similar: the query is a set of keywords and the results are sets of tuples that contain all the keywords and can be connected through the primary and foreign keys. Later work generally follows this semantics and further focuses on two aspects: efficiency and effectiveness. [14] incorporates IR-style ranking and proposes algorithms to return top K results efficiently. [23, 25] focus on the effectiveness and take into account more IR heuristics in the ranking function. [25] also proposes a Top-K algorithm which handles with non-monotonic ranking function. [20] studies the theoretical aspect of the keyword search problem. Beside relational databases, keyword search in graph databases has also been studied, e.g. [19, 12, 10]. The semantics of the query in graph is very similar to the relational databases: results are sets of connected nodes that contain all the keywords.

Keyword search over structure data exploits the relationship between keywords, and returned results may reflect some complex objects. However, given the semantics of the keyword search on structured data, the results can be arbitrary patterns. While it takes the advantage of schema-free search, it loses the control of result patterns and the opportunity to specify what those keywords refer to. Instead, in our system, queries are over nested objects with fixed formats, and every predicate is on a fixed attribute. For keyword predicates, although keywords may disperse across multiple tuples, these tuples are all aliases referring to the same entity, e.g. ACTOR in the execution example in Section 5.3. We believe this is an effective mechanism because when searching users usually know exactly what object they are looking for and what entities the predicates refer to. Our system gives users the opportunity of specifying the searched object and predicates on it, but not confining the concrete pattern. Another negative aspect of the keyword search semantics is that there may be redundancy in the result set. Consider two keywords w_1, w_2. If there is such a structure in the data $[a_1, a_2] \leftarrow c \rightarrow [b_1, b_2]$ where a_1, a_2 contains w_1 and b_1, b_2 contains w_2, then there will be four results $a_1 \leftarrow c \rightarrow b_1$, $a_1 \leftarrow c \rightarrow b_2$, $a_2 \leftarrow c \rightarrow b_1$ and $a_2 \leftarrow c \rightarrow b_2$, which reflect nearly the same semantics.

Keyword search in XML also attracts much attention. First set of work, e.g. [11, 31, 32, 24, 28, 16, 22], takes LCA's variations (e.g. ELCA, MLCA, SLCA) as query semantics and proposes different LCA algorithms. Another set of work tries to extend the XQuery with keyword search operators, augmenting IR ranking mechanism in XML, e.g. [3, 2, 29, 22]. Some recent work addresses the XML keyword search in new applications, e.g. over virtual views[27]. Although our ranking metric also relies on the LCA computation, as analyzed in Section 5, our LCA algorithm is fundamentally different with all the existing algorithms. These algorithms sort the matched nodes by the node encoding and scan the nodes sequentially. According to current encodings, e.g. Dewey ID or pre-order, this sequence is the same as the document order, and thus generated LCAs also follow the document order. There is no mechanism to guarantee that *lowest* LCAs are generated first. Instead, join-based algorithm scans the nodes bottom up, and LCAs in the lowest level are generated first, providing an efficient top-K support in terms of the ranking semantics.

Top-K query in relational databases has been widely studied recently. Existing work attacks the problem from different dimensions: monotonic ranking functions [9, 7, 5, 26], non-monotonic ranking functions [30, 25], existence of materialized views [15, 8, 4]. These work mainly focuses on the functions that combine multiple values from attributes of relation(s) and doesn't involve other operations. More related work to our scenario is the Top-K join problem[18, 14], which considers the traditional SQL join semantics. Although our ranking function also involves the join operation, the result semantics is different: it is no longer the combinations of joined tuples, but the central root that connects joined tuples.

9. CONCLUSION

Data in relational databases may have to be split and stored in a number of flatten tables because of the database normalization requirement. Tuples from different tables that are connected by joins can represent some complex/nested objects. It is highly desirable to support querying of such nested objects constructed the the flatten tables. In this paper, we present a system that allows users to specify their own virtual nested objects and issue queries over it easily. Our query semantics capture the fact that while the object format is fixed, predicates on one entity within the object can be satisfied over multiple aliases, e.g. ACTOR in the MOVIE object. According to this semantics, we propose

a ranking metric for the nested object. The metric satisfies the desirable constraints: the fewer and deeper aliases that satisfy the predicates, the better the ranking score. We proposes a pipelined architecture and two novel algorithms to support the query semantics and ranking mechanism efficiently. Join-based algorithm is "orthogonal" to existing LCA algorithms in terms of node processing order and provides guarantee that lowest LCAs are generated first. Top-K join algorithm is specifically tailored for our query semantics and thus is superior to existing methods for our queries. Experiments verify that our system outperforms naive SQL evaluations significantly.

10. REFERENCES

[1] S. Agrawal, S. Chaudhuri, and G. Das. Dbxplorer: A system for keyword-based search over relational databases. In *ICDE*, pages 5–16, 2002.

[2] S. Amer-Yahia, C. Botev, and J. Shanmugasundaram. Texquery: a full-text search extension to xquery. In *WWW*, pages 583–594, 2004.

[3] S. Amer-Yahia, E. Curtmola, and A. Deutsch. Flexible and efficient xml search with complex full-text predicates. In *SIGMOD Conference*, pages 575–586, 2006.

[4] N. Bansal, S. Guha, and N. Koudas. Ad-hoc aggregations of ranked lists in the presence of hierarchies. In *SIGMOD Conference*, pages 67–78, 2008.

[5] H. Bast, D. Majumdar, R. Schenkel, M. Theobald, and G. Weikum. Io-top-k: Index-access optimized top-k query processing. In *VLDB*, pages 475–486, 2006.

[6] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[7] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD Conference*, pages 346–357, 2002.

[8] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, pages 451–462, 2006.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[10] K. Golenberg, B. Kimelfeld, and Y. Sagiv. Keyword proximity search in complex data graphs. In *SIGMOD Conference*, pages 927–940, 2008.

[11] L. Guo, F. Shao, C. Botev, and J. Shanmugasundaram. Xrank: Ranked keyword search over xml documents. In *SIGMOD Conference*, pages 16–27, 2003.

[12] H. He, H. Wang, J. Yang, and P. S. Yu. Blinks: ranked keyword searches on graphs. In *SIGMOD Conference*, pages 305–316, 2007.

[13] T. Hearne and C. Wagner. Minimal covers of finite sets. *Discrete Mathematics*, 5(3):247–251, July 1973.

[14] V. Hristidis, L. Gravano, and Y. Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[15] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: A system for the efficient execution of multi-parametric ranked queries. In *SIGMOD Conference*, pages 259–270, 2001.

[16] V. Hristidis, N. Koudas, Y. Papakonstantinou, and D. Srivastava. Keyword proximity search in xml trees. *IEEE Trans. Knowl. Data Eng.*, 18(4):525–539, 2006.

[17] V. Hristidis and Y. Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

[18] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[19] V. Kacholia, S. Pandit, S. Chakrabarti, S. Sudarshan, R. Desai, and H. Karambelkar. Bidirectional expansion for keyword search on graph databases. In *VLDB*, pages 505–516, 2005.

[20] B. Kimelfeld and Y. Sagiv. Finding and approximating top-k answers in keyword proximity search. In *PODS*, pages 173–182, 2006.

[21] M. Ley. *The Nested Universal Relation Database Model*. Springer, 1992.

[22] Y. Li, C. Yu, and H. V. Jagadish. Schema-free xquery. In *VLDB*, pages 72–83, 2004.

[23] F. Liu, C. T. Yu, W. Meng, and A. Chowdhury. Effective keyword search in relational databases. In *SIGMOD Conference*, pages 563–574, 2006.

[24] Z. Liu and Y. Chen. Identifying meaningful return information for xml keyword search. In *SIGMOD Conference*, pages 329–340, 2007.

[25] Y. Luo, X. Lin, W. Wang, and X. Zhou. Spark: top-k keyword query in relational databases. In *SIGMOD Conference*, pages 115–126, 2007.

[26] N. Mamoulis, K. H. Cheng, M. L. Yiu, and D. W. Cheung. Efficient aggregation of ranked inputs. In *ICDE*, page 72, 2006.

[27] F. Shao, L. Guo, C. Botev, A. Bhaskar, M. M. M. Chettiar, F. Y. 0002, and J. Shanmugasundaram. Efficient keyword search over virtual xml views. In *VLDB*, pages 1057–1068, 2007.

[28] C. Sun, C. Y. Chan, and A. K. Goenka. Multiway slca-based keyword search in xml data. In *WWW*, pages 1043–1052, 2007.

[29] M. Theobald, R. Schenkel, and G. Weikum. An efficient and versatile query engine for topx search. In *VLDB*, pages 625–636, 2005.

[30] D. Xin, J. Han, and K. C.-C. Chang. Progressive and selective merge: computing top-k with ad-hoc ranking functions. In *SIGMOD Conference*, pages 103–114, 2007.

[31] Y. Xu and Y. Papakonstantinou. Efficient keyword search for smallest lcas in xml databases. In *SIGMOD Conference*, pages 537–538, 2005.

[32] Y. Xu and Y. Papakonstantinou. Efficient lca based keyword search in xml data. In *EDBT*, pages 535–546, 2008.