Hierarchical Result Views for Keyword Queries over Relational Databases *

Shiyuan Wang †
Department of Computer Science, UC Santa Barbara
Santa Barbara, CA, USA
sywang@cs.ucsb.edu

Junichi Tatemura
NEC Laboratories America
Cupertino, CA, USA
tatemura@sv.nec-labs.com

Arsany Sawires
NEC Laboratories America
Cupertino, CA, USA
arsany@sv.nec-labs.com

Oliver Po
NEC Laboratories America
Cupertino, CA, USA
oliver@sv.nec-labs.com

Divyakant Agrawal
Department of Computer Science, UC Santa Barbara
Santa Barbara, CA, USA
agrawal@cs.ucsb.edu

Amr El Abbadi
Department of Computer Science, UC Santa Barbara
Santa Barbara, CA, USA
amr@cs.ucsb.edu

ABSTRACT

Enabling keyword queries over relational databases (KQDB) benefits a large population of users who have difficulty in understanding the database schema or using SQLs. However, since there are different interpretations for a query, the results of KQDB that mix different possible answers still make it hard for users to consume and extract the information that is interesting to them. To help users with different preferences understand the query results and quickly locate the desired information, this paper proposes generating easy-to-navigate hierarchical result views for each interpretation to the query on the fly. We define the structures for organizing these hierarchical views, provide metrics for evaluating them in terms of user navigation efforts, and develop an efficient algorithm for generating the view structures. The effectiveness and efficiency of the proposed approaches are verified through an experimental study.

Categories and Subject Descriptors

H.2 [**Information Systems**]: Database Management; H.3 [**Information Systems**]: Information Storage and Retrieval

General Terms

Algorithms, Measurement, Human Factors

1. INTRODUCTION

Keyword queries over relational databases (KQDB) have enabled a similar search paradigm to web search on relational databases for users who do not know the database structure or do not have the expertise to formulate queries in structured query languages such as SQL. Several recent research proposals [2, 7, 1, 6] extend the web search paradigm based on keyword lookup and search for appropriate matches to answer user queries against structured data.

Although the goal of enabling keyword search on structured data appears to be straightforward and simple compared to web search, it gives rise to ambiguous results even though the data source is constrained due to its structure and is confined to a single organization. The main reason for this apparent ambiguity is that the implicit structure of the data source, which is explicitly captured in SQL, cannot be specified using keyword searches. We contend that the structural information should be provided to the user in a "usable" form at the time when query results are presented. We illustrate the underlying principles of this research proposal with a concrete example.

Consider the database schema shown in Fig. 1 for an E-commerce web site that sells books. A typical keyword query against a relational database will result in matching the keyword against all the attribute values in all tables. For all tables that have a match, the search would proceed further to find joinable relationships with other tables and process the query as a union of a set of SQL queries. For example, a keyword query "Jeff" and "Database" could result in multiple interpretations. One interpretation could be that the user is interested in finding all records of customers "Jeff" who have purchased a "Database" book. This interpretation corresponds to a SQL query which involves a join between *Customer* and *Item* tables. The other interpretation could be that the user is interested in the "Database" books authored by "Jeff". In this case, the underlying SQL query would involve a join between *Author* and *Item* tables.

The above example illustrates the ambiguity that results from an-

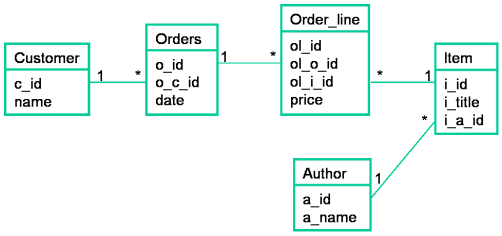


Figure 1: Example Database Schema Graph

*This work is partly supported by NSF Grant CNS-0423336.

†This work was partially done when Wang was an intern in NEC in 2008.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
KEYS'09, June 28, 2009, Providence, Rhode Island, USA.
Copyright 2009 ACM 978-1-60558-570-3/09/06 ...$5.00.

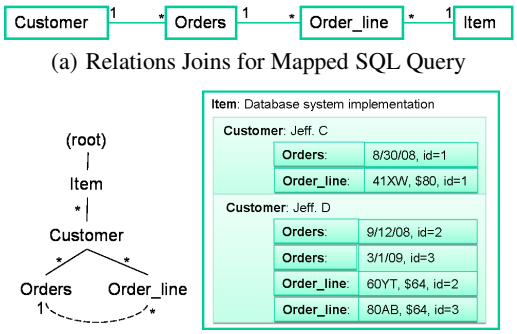
swering keyword queries by mapping them to a set of SQL queries. Furthermore, presenting the results indiscriminately as a set of matching tuples will make it extremely difficult for a user to extract the desired answer given that the answers, corresponding to multiple interpretations, are intermixed with each other.

Even for a single SQL query with a large result size, presenting the results in a more comprehensible way is desirable. Two recent proposals organize SQL query results in hierarchical category structures [3, 4], upon which a user can navigate along the parent-child hierarchy and selectively explore subsets of the results under certain levels of nodes. Each non-leaf node in the structure represents a filtering condition on the attribute values which is applied to the results from the parent node. The decision and organization of different filtering attribute values is made by summarizing user preferences based on past query workload. For example, the results of a "housing" query can be organized by neighborhood in the first level, and then categorized by price ranges in each neighborhood.

Unfortunately, the above approach does not help users consume the ambiguous results generated from mapping a keyword query to a set of SQL queries. Because the set of mapped SQL queries does not have a clear interpretation compared to a well defined SQL query. Moreover, the interpretation may not be clear for a single mapped SQL query. Typical keyword query processing just finds a join chain among a set of matching tables, and returns all tuples from all the tables in the join chain without discrimination of condition information and desired information. However, a user generally experiences two stages when she looks for her desired information: In the first stage, she collects all the conditions and picks the ones that may be of interest. In the second stage, she finds her desired information based on the selected conditions. We refer to this type of behavior as the *two stage navigation pattern*.

To help users consume the results and extract the desired information by effective condition pruning, we organize the results in a navigable hierarchical view for each interpretation of the keyword query. Each mapped SQL query could have multiple result views corresponding to different interpretations. For the example query of "Jeff, Database", if a user has specific database books in mind as search conditions for the customers "Jeff" who purchased these books, organizing the results in the form of Fig. 2(b) would satisfy her intention. On the other hand, organizing the results in the form of Fig. 3(a) may be more useful if the user wants to check the details of the order and order_line of customers "Jeff" step by step. Finally, if the user has some target customers "Jeff" as search conditions to find what database books they have purchased, organizing the results in the form of Fig. 3(c) is more appropriate.

We refer to such structures as *schema trees*. An example of a schema tree and the corresponding hierarchical result view is shown in Fig. 2(b) and Fig. 2(c). We refer to the nodes with condition information, such as *Customer* and *Item*, as *predicate nodes*, and the other nodes, such as *Orders* and *Order_line*, as *non-predicate nodes*. For each mapped SQL query, we generate candidate schema trees based on the two stage navigation pattern. A *basic* schema tree generation algorithm (BA) would put all predicate nodes in a single path as the top branch of a tree which can be checked together, and then grow non-predicate nodes from the existing branches of the tree. However, some schema trees are more appropriate for fast retrieval of information than others, hence we propose a navigation cost model for measuring schema trees in quantified user navigation costs. Based on the navigation cost model, we propose a *navigation order aware* schema tree generation algorithm (NOA) which is more efficient than the *basic* schema tree generation algorithm. The navigation order aware algorithm generates a small set of the most cost saving schema trees that represent different query

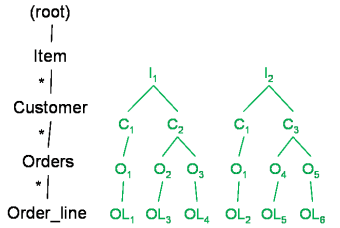


(a) Relations Joins for Mapped SQL Query

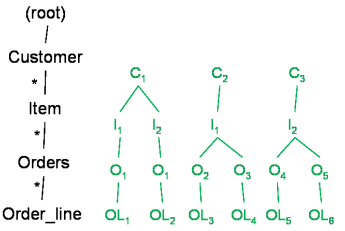
(b) Schema Tree (c) Result View

Figure 2: An Example of Mapped SQL Join And Hierarchical Result View

interpretations in order of increasing costs. Then the corresponding hierarchical result views are shown to a user in the same order. The clearly structured views make it possible for users to quickly prune and decide the answers of interest.



(a) T_1 (b) Instance Tree According to T_1



(c) T_2 (d) Instance Tree According to T_2

Figure 3: Tow Examples of Schema Trees And The Corresponding Instance Organizations

2. KEYWORD QUERIES OVER RELATIONAL DATABASES

This section gives an overview of the underlying formulation and the basic approach for processing KQDB.

We assume a keyword query $KQ = \{K_1, K_2, ...K_m\}$ implies conjunctive search conditions $(K_1 \wedge K_2 \wedge ... \wedge K_m)$, and each keyword K_i can find (partial) matches in at least one column in the database. The *database schema* is represented as an undirected graph G_s, in which each node is a relation in the database, and each edge represents a joinable relationship between two attributes in a pair of relations (e.g. primary key–foreign key). Formally, define a relation node n as $R(A_1, A_2, ..., A_k)$, in which $A_j (1 \leq j \leq k)$ are its attributes. Define a *joinable relationship edge* as $(R_l, R_r, A_l, A_r, card_l, card_r)$ where $A_l \in R_l$, $A_r \in R_r$ are joinable attributes of the two relations R_l and R_r, and $card_l$, $card_r =$

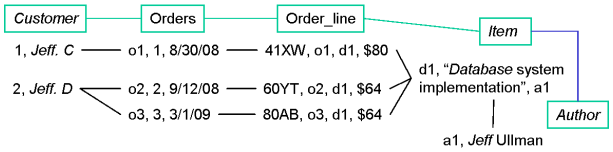


Figure 4: A Joining Network of Tuple Sets

$[1|*]$ (one-to-many or many-to-one) capture the cardinality relationship between the instances of A_l and A_r. For example in Fig. 1, one *Customer* has multiple *Orders*, so $card_l = 1$ and $card_r = *$.

The results of a keyword query are obtained from the union of the results of a set of SQL join queries $\{Q\}$. Based on [7, 1], we outline the important steps for processing KQ: (1) **Matching keywords to tuples**. Find a set of tuples $\{t_i\}$ which together match all the keywords in KQ, and group them by keyword and relation. In our example, a customer set {"Jeff. C", "Jeff. D"}, a book item with the title "Database system implementation" and an author named "Jeff Ullman" are found. (2) **Identifying joining network of tuple sets**. Connect every pair of tuple sets (t_l, t_r) where $t_l \subseteq R_l \wedge t_r \subseteq R_r$, and $\exists$ a joinable relationship edge $(R_l, R_r, A_l, A_r, card_l, card_r)$ in G_s. As shown in Fig. 4, these pairs of tuple sets form a joining network, in which all boundary nodes are the keywords matched tuples. (3) **Enumerating all possible SQL join queries**. Note that the joining network can be partitioned into several joining trees of tuple sets, where each keyword has exactly one match node in a joining tree. Denote the database schema subgraph corresponding to a joining tree as G. Then form a SQL query by joining the relations along the paths of G and specifying keywords matched columns as selection predicates. For example, by traversing from *Customer* {"Jeff. C", "Jeff. D"} to *Item* "Database system implementation" in Fig. 4, we get a database schema subgraph as Fig. 2(a), and a SQL query which filters out some *customer*s and *item*s, and joins relations *Customer*, *Orders*, *Order_line* and *Item* together.

3. HIERARCHICAL RESULT NAVIGATION

Our goal is to generate hierarchical result views as structural interpretations for each enumerated database schema subgraph G. In this section, we first define such a structure called a *schema tree*. Then we discuss how to enumerate different schema trees, and efficiently generate "good" schema trees on which the user efforts for navigation are reduced according to a navigation cost model.

3.1 Schema Tree Definition

Definition 1. A *schema tree* is a tree structure with a virtual root that is generated from a database schema subgraph G. It has two properties: (1) *Correctness*. It preserves the relations and relationships between relations in G. (2) *Functionality*. Its structure specifies all constraint information for organizing a data instance into a hierarchical view.

As in a database schema subgraph G, each node in a schema tree T represents a relation, denoted as $n(A_1, A_2, ..., A_k)$. The edges in T are of two types:

(1) *Structural Join Edge* s (parent-child edge, represented using a solid line) by which the hierarchical structure is built, such as the *Item-Customer* edge in Fig. 2(b). Formally represent s as s(*pn*,*cn*, $path$,$card$,$constraint$), in which pn, cn are parent and child nodes respectively, $path$ represents the ordered sequence of the binary join relationships that justifies the join between pn and cn, $card = [1|*]$ denotes the relative cardinality of cn under pn, and $constraint = [\{as\}|null]$ specifies the possible selection constraint that pn's ancestor edge as might enforce on the instances of cn due to the overlap of the $path$s of as and s. For the example of *Item-Customer* edge in Fig. 2(b), pn and cn are *Item*, *Customer* respectively, $path$= {(*Item*,*Order_line*)(*Order_line*,*Orders*) (*Orders*,*Customer*)}, $card = *$, and $constraint = null$. For the example of *Orders-Order_line* edge in Fig. 3(a), $constraint = s(Item, Customer)$, because although an *order* may have several *order_line*s, only the *order_line* belonging to the particular *item* are of interests.

(2) *Referential Join Edge* r (non-parent-child edge, represented using a dashed line) by which direct joinable relationships in G are preserved, such as *Orders-Order_line* edge in Fig. 2(b). Formally represent r as $r(ln, rn, A_l, A_r, card_l, card_r)$, which corresponds to a joinable relationship edge (R_l,R_r,A_l,A_r,$card_l$,$card_r$) in the original database schema graph, and ln, rn correspond to relations R_l and R_r. For the example of *Orders-Order_line* edge in Fig. 2(b), ln and rn are *Orders*, *Order_line* respectively, A_l and A_r are primary and foreign key *o_id*, $card_l = 1$ and $card_r = *$.

3.2 Generating New Schema Trees

To ensure the correctness of generated schema trees, we first transform G to an initial tree T_{init}, the flatest tree where all nodes are under a virtual root and connected by referential join edges as illustrated in Fig. 5. Then we apply an operator called $nest$ to construct structural join edges based on T_{init}, hence producing new schema trees.

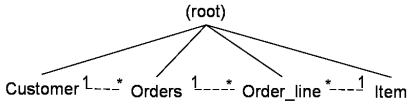
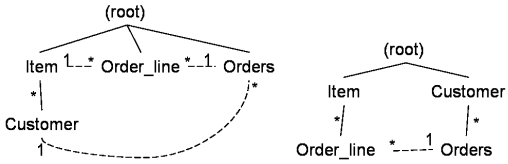


Figure 5: T_{init}

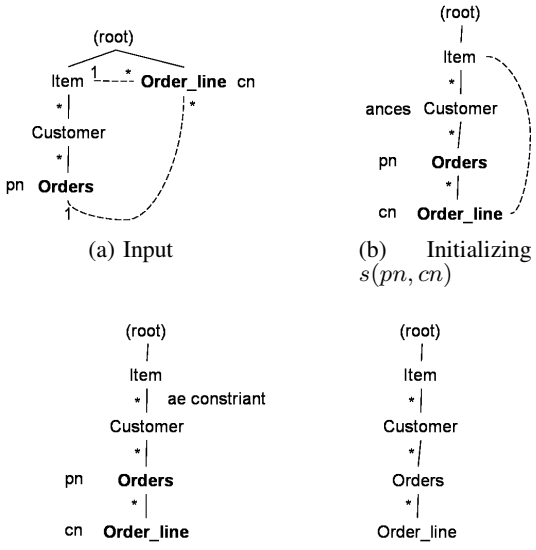
The $nest$ operator is in the form of $nest(pn, cn, path)$, in which cn is a free node, pn can be a free or non-free node, and $path$ specifies the particular ordered binary join sequence from pn to cn. When $path$ is obvious, we simply denote $nest(pn, cn, path)$ as $nest(pn, cn)$. We consider a node to be *free* if it is not attached to a structural join edge. If pn is a free node, it constructs a new subtree with itself as the root and cn as the child, such as $nest$(*Item*, *Customer*) in Fig. 6(a). If pn is not a free node, after initializing a new structural join edge s, we check the relationships between cn and each of the ancestors of pn, and update the tree structure accordingly as follows: (1) Remove the referential join edge between cn and the ancestor of pn if it exists, because this relationship is already captured in the hierarchy. (2) Add ancestor constraint reference to the structural join edge between pn and cn, s, if its $path$ overlaps with the $path$ of the ancestor edge. Fig. 7 demonstrates the processing steps of $nest$(*Orders*, *Order_line*) where *Orders* is a non-free node. In step (c), we remove the referential join edge between *Item* and *Order_line*, and add the ancestor constraint $s(Item, Customer)$.



(a) T_3 (b) T_4

Figure 6: More Example Schema Trees

Clearly, a schema tree T generated from T_{init} by applying $nest$ is correct: Because in the $nest$ process, no relation is removed, and



(a) Input

(b) Initializing $s(pn, cn)$

(c) Redundant referential join edges removed & Checking for constraint

(d) Output

Figure 7: $nest$(Orders, Order_line)

no relationship is removed except when it is already captured in the tree hierarchy.

Note that the total number of trees that can be enumerated by applying $nest$ is exponential with respect to the total number of nodes. However, enumerating all these trees is not practical for the purpose of one-time query navigation, because we are only concerned with "good" schema trees that are easy for users to navigate and to quickly locate the desired information. So a measure of such "good" schema trees is required for ranking and an effective algorithm should only output the higher ranked trees.

3.3 Navigation Cost Model

The structure of a schema tree confines the paths along which a user can navigate on the corresponding instance trees. So our metric for evaluating schema trees uses a navigation cost model, which accounts for user navigation efforts taken during each traversal of an instance node or an edge. A tree with a smaller navigation cost is supposed to be easier to navigate than trees with larger costs.

Let nodes n_1, n_2,..., n_p correspond to relations R_1, R_2,..., R_p respectively. For a given schema tree T, define a *navigation path* as $(n_i[r_i|s_i])^* n_p$ $(1 \leq i < p)$, where $[r_i|s_i]$ is either a referential join edge r_i or a structural join edge s_i between n_i and n_{i+1}. Correspondingly, define the *navigation order* for the relations in the navigation path as $(R_1 R_2 ... R_p)$. The data instances we consider in the following are the result instances of different relations that participate in a mapped SQL query. So the data cardinality here is the size of result instances of a relation.

Assume users are able and prefer to take the shortest navigation paths to check the predicates and locate the desired subset of answers. We now elaborate on the *two stage navigation pattern* presented in Section 1:

1. Starting from a root of a subtree or a free node whose corresponding relation has the smallest data cardinality, locate predicate nodes in shortest paths.
2. From the last level nodes of the first stage, *selectively* explore unvisited non-predicate data nodes as follows:

   If all predicate nodes are located in a *single path* in the first stage (*single path condition*), pick only *one* branch of predicate data nodes and check the corresponding non-predicate data nodes. Otherwise, check *all* unvisited non-predicate data nodes to ensure the correct correspondences between predicates of different paths.

For example, navigation on T_1 of Fig. 3(a) starts from *Item* obviously. In the first stage, it locates all predicate nodes *Item* and *Customer* in a single path. In the second stage, it visits *Orders* and *Order_line* below a certain *Customer*. Navigation on T_3 of Fig. 6(a) also starts from *Item* because *Item* is both a root of a subtree and has the smallest data cardinality. Then the navigation paths on T_3 and T_1 are the same in the first stage, but the navigation path on T_3 in the second stage goes through referential join edges. When navigating a referential join edge, a user has to do some additional work to connect the instances of the two end relation nodes by the same result ids. In contrast to T_1 and T_3, navigation on T_4 of Fig. 6(b) cannot locate the predicate nodes *Item* and *Customer* in a single path, so in the second stage it needs to check all *Order_line* and *Orders* to ensure the correspondences between *Item* and *Customer*.

Based on the above two stage navigation pattern, define the total navigation cost for a SQL query Q on a given schema tree T as $cost(T, Q) = cost_1(T, Q) + \alpha * cost_2(T, Q)$, where $cost_1(T, Q)$, $cost_2(T, Q)$ are the navigation costs in the first and second stage respectively, and α is the selectivity of the data explored in the second stage which is determined by the single path condition. $cost_1(T, Q)$ and $cost_2(T, Q)$ are computed by adding up the costs for navigating from all the first level nodes to the last level nodes in the navigation paths of the first and second stage respectively. For example, $cost_1(T_1, Q)$ is the cost for navigating from all *Item*s to all corresponding *Customer*s in Fig. 3(b).

The *basic* method for computing the cost on a navigation path is as follows: Compute the cost for navigating from a first level instance node and then sum up the costs for all such instances. The cost for navigating from an instance node tn at a certain step is computed recursively by adding the unit costs for visiting tn, traversing the edges between tn and the next step instance nodes tn_j and the costs for navigating from each tn_j. Let c_n, c_s and c_r be the unit cost of visiting an instance node, traversing a structural join edge and a referential join edge respectively (we set $c_n < c_s < c_r$ based on the convenience of visiting). Formally define the cost for navigating from an instance tn at the $(i-1)$th step to the instance nodes tn_j at the ith step of the navigation path, $cost(tn, T, Q) = c_n + \sum_j (cost(tn_j, T, Q) + c_{s/r})$, in which tn is an instance of R_{i-1}, tn_j is an instance of R_i, $\exists s/r(R_{i-1}, R_i) \in T$ and $c_{s/r} = c_s$ or c_r depending on the type of the edge between R_{i-1} and R_i. For example, navigating from *Item* I_1 to *Customer* C_1 and C_2 on T_1 in Fig. 3(b) has cost $c_n + (c_n + c_s) + (c_n + c_s) = 3c_n + 2c_s$. The first stage navigation, which means navigating from *Item* I_1 to *Customer* C_1 and C_2, and navigating from I_2 to C_1 and C_2, has cost $cost_1(T_1, Q) = 3c_n + 2c_s + 3c_n + 2c_s = 6c_n + 4c_s$.

The basic cost computing method is intuitive, but not efficient. So we propose an *optimized* cost computing method based on the following observations: First, the number of nodes at a given level of navigation is equal to the number of instances joining all the relations in a prefix of the navigation order, beginning with the navigation start relation and ending at the current relation. Second, the number of edges at a given level of navigation is equal to the number of downstream nodes. For example in T_1 of Fig. 3(b), the number of *Customer*s at the second level of the first stage navigation is $|join(Item, Customer)| = 4$, and the number of edges $|s(Item, Customer)|$ is the same. The number of the first level nodes *Item* is the cardinality of the *Item* relation, $|join(Item)| =$

2. Third, on joining the same set of relations, the cardinality of join instances is irrelevant to the navigation order of these relations, such as $|join(Item, Customer)|$ = $|join(Customer, Item)|$ = 4 on T_1 and T_2. We use the first two observations to transform the recursive computation in the instance level to the relation level. We use the third observation to share the cardinalities of join instances for computing the navigation costs of different trees. For T_1 and T_2, the first stage navigation costs are only decided by the cardinalities of *Item* and *Customer*. It is then obvious that the first stage navigation cost of T_1 is smaller than that of T_2, and actually the same holds for the total navigation costs of T_1 and T_2.

3.4 Generating Easy-to-Navigate Schema Trees

Analyzing the *basic* schema tree generation algorithm in Section 1 according to the navigation cost model, we notice two key inefficiency issues: (1) It generates redundant trees, e.g. two trees with the same navigation order but different navigation costs may be generated, such as T_1 of Fig. 3(a) and T_3 of Fig. 6(a). In this case, we just need to keep the tree with the smaller cost by giving preference to structural join edges instead of referential join edges. (2) It does not prioritize the generation order of trees with respect to increasing costs, so a user could see a larger cost tree before seeing a smaller cost tree, such as T_2 before T_1.

To prioritize the generation order of trees, we use a technique similar to the optimized cost computing method and refer to it as *prioritizing generation*. Define *path cardinality* to be the sum of the cardinalities of join instances of different levels of relations in a given navigation path. We use path cardinality, especially the path cardinality of the first stage navigation path, as an estimation of the navigation cost, and schedule generating trees in increasing path cardinality. For example, the path cardinality of the first stage navigation on T_1 is $path\_cardinality_1(T_1)$ = $|join(Item)|$ + $|join(Item, Customer)|$ = 2 + 4 = 6. The first stage path cardinality on T_2 is $path\_cardinality_1(T_2)$ = $|join(Customer)|$ + $|join(Item, Customer)|$ = 3 + 4 = 7. So T_1 is generated before T_2. Another heuristic technique based on the path cardinality, which we refer to as *shallow tree first*, is that in the second stage of navigation, shallow branches tend to have smaller path cardinalities than deeper branches, so shallow branches are created before deeper branches.

Thus we propose a *navigation order aware* schema tree generation algorithm (NOA) as the following:

1. Compute and sort path cardinalities for different predicate single paths, and build the top branches of different trees in sorted order.
2. $nest$ non-predicate nodes under the bottommost predicate node of the top branches in increasing tree heights, i.e., for each of the existing trees, $nest$ any of the free non-predicate nodes into the bottommost predicate node or any non-free non-predicate node.

Step 1 prioritizes the generation order of trees. Step 2 reduces the number of candidate parents for $nest$, thus reducing the number of redundant trees. To illustrate the algorithm for our running example, at step 1, we compare the path cardinalities of $Item - Customer$ and $Customer - Item$, and decide to build the first top path first because it has smaller path cardinality. At step 2, we $nest$ *Orders* and *Order_line* at the same level under *Customer* and generate the tree in Fig. 2(b), then still from the top branch $Item - Customer$ we $nest$ *Orders* into *Customer* and $nest$ *Order_line* into *Orders* (or $nest$ *Order_line* first), increasing the tree height by one and generating T_1 of Fig. 3, and so on. Let $|P|$ be the number of predicate nodes, and $|V|$ be the total number of nodes in the database schema subgraph G. The upper bound of the number of schema tree generated by this algorithm is $|P|! * (|V| - |P|) * 1 * (|V| - |P| - 1) * 2 * ... * 1 * (|V| - |P| + 1)$=$|P|! * (|V| - |P|)! * (|V| - |P| + 1)!$. Note that this is a very loose upper bound, because it includes duplicate trees. However, in the implementation of step 2, we prune the duplicate trees. Furthermore, $|P|$ and $|V|$ are usually very small in practice.

4. EXPERIMENT EVALUATION

This section evaluates the effectiveness of the navigation cost model and the tree generation algorithms as compared to real user navigation behaviors by a user study, and verifies the efficiency of our approaches by comparing different cost computing and algorithm settings on a synthetic data set. The implementation was done in Java. All the experiments were executed on a Dell PC of 1.4M HZ processor and 768M memory running Windows XP SP2.

4.1 Effectiveness of Satisfying User Preference

Note that the problem of result ambiguity, which is the focus of this work, is different from existing proposals for keyword queries over relational databases [2, 7, 1, 6]. The queries we are handling do not have clear interpretations as in the cases of well defined SQL queries. Thus we decided to do a user study solely on the approaches we propose instead of comparing with existing result presentation approaches, such as [3, 4] which address the problem of presenting the results of well defined SQL queries.

To evaluate the effectiveness of the navigation cost model for ranking and prioritizing tree generation of the navigation order aware algorithm according to user preferences, we implemented a user interface to show different hierarchical organizations of the data representing different schema trees for a single query. The chosen schema trees consist of all the trees generated from the navigation order aware algorithm and the trees without free nodes generated from the basic algorithm. The trees are shown in pairs for easy comparison. The program computes a user's rank of trees based on pairwise comparison results.

We use the IMDB movie data set [8]. We chose five relations *Movie*, *Director*, *Rating*, *Genre* and *Keyword*, and built a relational schema in which the latter four relations are connected by *Movie*. All primary-foreign key relationships are built on the titles of movies. We designed four queries as user tasks shown in Table 1 and indexed by QI. The user studies were conducted on 10 subjects (graduate students). Each user task requires navigating and locating his/her desired information within the query results on each tree, and judging which tree is easier for such navigation.

Table 1: Queries for User Study Tasks

QI	Query
1	director name like 'Frank', movie rank > 7.5
2	movie title like 'Frank', movie rank > 7.5
3	movie rank > 8.5, movie plot like 'prison'
4	movie with title like 'Family' and produced in 2008

Two measures are used for comparing the subjects' ranks and the ranks given by our cost model: (1) Normalized Average Rank Distance (NARD), the average absolute difference of the two ranks on the same trees normalized by the maximum rank difference. NARD is in the range of [0,1], the smaller the better. (2) Majority Voted Rank (MVR) Precision, an approach inspired by the measure of page relevance in the user study of [5], the larger the better. The user rank of a tree, MVR, is computed by taking the majority of subject ranks. MVR Precision is the percentage of matches between MVR and the ranks given by the cost model. From the re-

sults shown in Table 2 (small NARD and high MVR Precision), we can see that the cost model mimics the real navigation behaviors reasonably.

Table 2: Ranking Precision of Cost Model

QI	NARD	MVR Precision
1	0.22	0.75
2	0.0	1.0
3	0.18	1.0
4	0.0	1.0

We evaluated the two techniques used in the navigation order aware algorithm i.e. prioritizing generation and shallow tree first as follows: (1) *Best Tree Evaluation*, measure the precision (percentage) that the algorithm and the subjects agree on the best trees for different navigation orders. (2) *Shallow Tree Evaluation*, measure the precision (percentage) that the algorithm and the subjects agree on the *shallow tree first* heuristic among the trees satisfying the single path condition. Since the trees generated for QI1 and QI3 have more distinct navigation orders because more relations are predicate nodes, and the trees generated for QI4 have more non-predicate nodes, we did evaluation (1) on QI1 and QI3, and evaluation (2) on QI4. The results are shown in Table 3 and Table 4. The precision results here are not as good as that of the cost model. One reason is that prioritizing is based on the intuitions from path cardinality which does not completely compute the navigation costs before hand. Another reason is that users' personal preferences might not always be consistent with the computed costs.

Table 3: Precision of Best Trees

QI	Precision
1	0.5
3	0.75

Table 4: Precision of Shallow Tree First

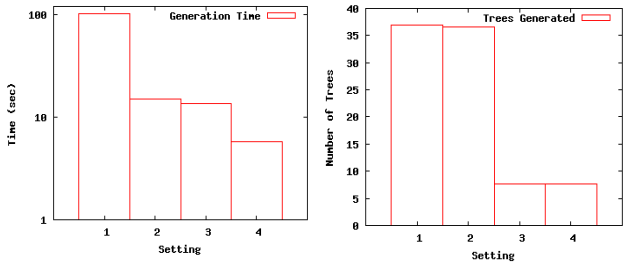
QI	Precision
4	0.5

4.2 Efficiency of Tree Generation

To evaluate the efficiency of the proposed tree generation algorithm and cost computing method, we form four settings by combining one algorithm from the algorithm set {basic (BA), navigation order aware (NOA)} and one method from the cost computing method set {basic (BC), optimized (OC)}: (1) BA+BC, (2) BA+OC, (3) NOA+BC, (4) NOA+OC.

We conducted the evaluation on a well-known synthetic data set TPCW [9]. The TPCW scaling factors: number-of-items and number-of-EB, are set to 10K and 20 respectively. The size of the database is 70MB. We use synthetic SQL queries which correspond to meaningful keyword queries as input. These queries were generated as follows: First, asked a volunteer to create 12 query templates and specify regular expressions for the possible conditions. Then, we created 10 queries from each query template by filling in the conditions with randomly picked values from the database based on the regular expressions. We filtered out queries with small result sizes. This results in 109 queries. Each query contains no less than 2 selection conditions and joins of 2 to 4 relations. The average result size of these queries is 4914. We report the averages of the schema trees generated and the total generation time for each of the four settings.

The evaluation results are shown in Fig. 8. Comparing the average time taken by any algorithm + OC method to that of the same



(a) Total Generation Time (b) #Trees Generated

Figure 8: Comparison of Tree Generation

algorithm + BC method, it is clear that OC method with the cardinalities of join instances dramatically saves time. For example, the total generation time of the fourth setting is 5.73s, compared to 13.43s in the third setting as shown in Fig. 8(a). The advantage of OC method is even more obvious when more trees are generated, for example compare the total generation time of the second setting (15.12s) with that of the first setting (102.37s). NOA outperforms BA by generating less trees (see Fig. 8(b)) and thus reducing the time taken in tree generation.

5. CONCLUSION

We have observed the problem of result ambiguity in keyword queries over relational databases. In this paper, we propose presenting different result interpretations in different navigable result views. We represented each result interpretation as a light-weight navigable hierarchical structure called a *schema tree*. We measured "good" schema trees using a cost model which is derived from general user navigation behaviors rather than past query workload. Based on the cost model, we have proposed an efficient algorithm to generate a best set of schema trees, each of which captures a different interpretation for the keyword query.

6. REFERENCES

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

[2] Gaurav Bhalotia, Arvind Hulgeri, Charuta Nakhe, Soumen Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using banks. In *ICDE*, pages 431–440, 2002.

[3] Kaushik Chakrabarti, Surajit Chaudhuri, and Seung won Hwang. Automatic categorization of query results. In *SIGMOD Conference*, pages 755–766, 2004.

[4] Zhiyuan Chen and Tao Li. Addressing diverse user preferences in sql-query-result navigation. In *SIGMOD Conference*, pages 641–652, 2007.

[5] Taher H. Haveliwala. Topic-sensitive pagerank. In *WWW '02: Proceedings of the 11th international conference on World Wide Web*, pages 517–526, 2002.

[6] Vagelis Hristidis, Luis Gravano, and Yannis Papakonstantinou. Efficient ir-style keyword search over relational databases. In *VLDB*, pages 850–861, 2003.

[7] Vagelis Hristidis and Yannis Papakonstantinou. Discover: Keyword search in relational databases. In *VLDB*, pages 670–681, 2002.

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

[9] http://www.tpc.org/tpcw/.