UIUC Technical Report: UIUCDCS-R-2006-2766, UILU-ENG-2006-1806. August 2006

Best-k Queries on Database Systems

Tao Tao ChengXiang Zhai
Department of Computer Science
University of Illinois at Urbana-Champaign

Abstract

As exploratory queries become more and more popular, the study of how to select k items based on fuzzy matching and ranking of database tuples (i.e. top-k queries) has attracted much attention recently. However, taking the top-k tuples based on their scores computed independently *is inadequate for modeling some complex queries finding the* best k *tuples based on some selection criteria involving a global measure on multiple selected tuples (e.g., tuple redundancy or compatibility). In this paper, we introduce and study such best-k queries, and further model a database selection problem generally as a decision problem, in which a database system would respond to a query by selecting a subset of tuples that optimize a certain utility function defined globally. Accordingly, we present a general formal framework for database selection, which covers the boolean query search, the top-k query search, and the best-k query search all as special cases. We prove that finding answers to a general best-k query is an NP-hard problem. We propose an anytime algorithm, which allows a user to stop the algorithm at any time to achieve a flexible tradeoff between the result optimality and the running time, to execute a best-k query efficiently. Experiment results show that the algorithm can be used to efficiently answer a best-k query.*

1 Introduction

With a traditional database query language (e.g., SQL), a user must specify *precisely* the conditions to be satisfied by the selected tuples. This is appropriate when (1) the information in the database is precise and (2) the user knows exactly what to look for. While these are generally true in most traditional database applications, they are not true in many new applications that involve *exploratory* queries and imprecise database information such as multimedia and text databases [8]. Indeed, in such applications, it is often very difficult (even impossible) for a user to precisely specify the conditions in terms of data values for selecting data tuples, even though the user may have clear preferences among the candidate tuples. It is thus necessary to study how to assist a user in expressing such preferences and how to answer a query involving such preferences.

DocID	w_1	w_2	w_3	Date	SummaryLength
d_1	4	5	0	Jan-1-2003	50
d_2	1	4	1	Feb-10-2003	20
d_3	3	3	1	Dec-21-2003	15
d_4	4	3	1	Mar-3-2003	15
d_5	4	4	1	Apr-10-2003	40
d_6	0	1	5	May-12-2003	40
d_7	4	4	2	Apr-10-2004	40

Table 1. A sample text document database

For example, consider an overly-simplified database of text documents shown in Table 1. There are six attributes, including the document ID, the three words in our vocabulary (w_1, w_2, and w_3), the publication date, and the length of the summary of a document.

Suppose a user is interested in documents published before year 2004 about w_1 and w_2. He may use a Boolean query such as the following:

SELECT DocID
FROM Table t
WHERE before(t.Date, 2004) and t.$w_1 > 3$ and t.$w_2 > 3$

This query would return d_1 and d_5 as results, but not d_3 and d_4, though they intuitively may also be relevant. In such a case, it may be more appropriate to introduce a scoring function defined on all the three words and the query words (i.e., w_1 and w_2) and assume that the user simply prefers a document with a higher score by this function. For example, a simple scoring function can be $s(t,q) = \sum_{w_i \in q} t.w_i$ (i.e., the sum of frequency of each matched query word). We can then allow a user to specify the preferences through the following top-k query:

SELECT DocID
FROM Table t
WHERE before(t.Date, 2004)
ORDER(4) BY s(t,q)

That is, we will first select documents that have published before year 2004, score each document using function s, and return the top 4 documents with the highest scores according to s.

The top-k query problem has been studied extensively recently [9, 8, 3, 2, 6, 4]. In most studies, the focus was on how to efficiently execute a top-k query. For example, Fagin has proposed several efficient algorithms that can be used to answer a top-k query exactly [9, 8]. Chaudhuri and Gravano studied how to convert a top-k query to a range query that traditional databases can process [6]. Chang and Hwang studied how to minimize the probes when evaluating a top-k query with expensive predicates [4]. Agrawal proposed another angle of top-k problem [2], discussing how to combine different user preferences, when multiple users search for data tuples together.

A major limitation of top-k queries is that the k tuples are selected based on a ranking of tuples according to scores that are computed for each tuple *independently*. Thus, for example, the score of each document in our example above is computed independent of other documents. This limitation makes top-k queries inadequate for expressing more complex selection preferences which may involve some *global* measure defined on multiple tuples. One such measure is semantic redundancy or novelty [19]. In a text database, many articles may be very similar, though not identical, so that users would like to see retrieved results as diversified as possible. It is clearly impossible to use a top-k query to express a user's desire for minimizing the redundancy among the returned results (by whatever redundancy measure appropriate for the application).

Another limitation of top-k queries is that the user must specify the number k, which sometimes may be better decided by the system. Consider again the example above, suppose we will display the summaries of all the selected documents (as a search engine often does) and our screen size can only display up to 100 words (e.g., with a PDA). We now not only have an additional global constraint (i.e., the sum of the summary lengths of the selected documents must not exceed 100), but also can automatically choose a k to maximize the amount of relevance information in the selected documents. Intuitively, in the example of Table 1, the optimal answers are $\{d_1, d_4, d_3, d_2\}$ because they all match the two query words w_1 and w_2 well and their summary lengths sum to 100. However, if we use the function s to rank the documents (as in the case of top-k queries), and select top k documents subject to the screen size constraint, the results would be $\{d_1, d_5\}$, since d_1 and d_5 would be ranked at top positions, and their sum of summary lengths is already 90, leaving no room for a third document.

The example above, while over-simplified, clearly illustrates two points: (1) Top-k queries are inadequate for database selection problems involving global constraints or preferences, which we will refer to as "best-k" query problems since our goal is to select k tuples according to some globally defined preferences. (2) Defining a user's preferences and constraints on the *whole* set of selected tuples, rather than on each individual tuple, is necessary to model best-k queries. Indeed, some preferences and constraints (e.g., screen size) are impossible to capture through functions defined solely on individual tuples. The example also shows that we need to define the database selection problem in a more general way in order to accommodate complex user preferences.

The need for querying a database with such complex preferences is abundant. In practice, a user often solves such a problem by post-processing the database search results manually or simply by-passes the problem by accepting the presumably non-optimal search results from a database with existing querying capabilities. As another example, consider a real estate dealer who wants to invest a certain amount of money on buying several houses and would like to spread the money to buy houses covering several different regions in a town. In this case, we would prefer the returned tuples to be as diversified as possible. Clearly, diversity is a global property that cannot be captured by a top-k query, and hence the problem is a best-k query problem.

In this paper, we study the best-k query problem together with the existing Boolean queries and top-k queries within a single unified database selection framework. In this framework, we model the database selection problem generally as a decision problem. Specifically, we assume that a database system would respond to a database query by selecting a subset of tuples that optimize a certain utility function defined on the tuples. The database selection problem is thus reduced to an optimization problem where the search space consists of all the subsets of tuples in the database and the objective function is a global utility function defined on any subset of tuples. We show that such an optimization framework covers the boolean query search, the top-k query search, and the best-k query search all as special cases, corresponding to different utility functions. Specifically, we show that the best-k queries can be regarded as a natural generalization of the top-k queries by allowing a user to specify an arbitrary utility function. The framework provides a roadmap for exploring complex database queries with different levels of complexity.

We further study how to efficiently evaluate a best-k query. We show that finding answers to a best-k query is an NP-hard problem and study efficient algorithms to find answers to a best-k query. Without any additional assumption on the utility functions, the only way to answer a best-k query would be a brute force solution, in which we enumerate all the possible subsets of items; clearly, such a solution is only useful for extremely small-scale best-k problems. We note that many practically interesting utility functions satisfy a monotonicity property, which can be exploited to dramatically prune the search space. Accordingly, we propose to use a branch-cutting strategy to prune the search space in both breadth-first and depth-first search. While these algorithms are efficient for obtaining exact optimal answers to a best-k query, the worst cases are still exponential. We further explore how to obtain approximate answers quickly. It turns out that the depth-first branch-cutting algorithm can naturally provide flexible tradeoff between the optimality of results and the running time of the query since it is an anytime algorithm, which provides an approximate solution at every step of search. We evaluate these algorithms on simulated data sets. The results show that, for any non-trivial best-k problems, the depth-first search strategy is more feasible and much faster than the breadth-first search strategy because the latter easily runs out of memory.

Our contribution in this paper is three-fold:

- We formulate a new database searching problem — the best-k selection. This best-k is not only generally enough to cover most previous work, such as traditional binary database search and top-k queries, but it also suggests new database problems.

- We propose a framework to address the best-k selection. This framework further suggests a query language, which is a natural extension of of the SQL language and the top-k queries.

- We design efficient algorithms to select best-k items from databases. In particular, we propose the *ordered enumeration tree and develop several pruning techniques based upon this tree. Experiments show the effectiveness of these pruning techniques.*

2 A Unified View of Database Queries

In order to formally define the best-k problem, we model a database query generally as a decision problem, in which a database system would respond to a selection query by choosing an optimal subset of data tuples from the space of all possible subsets in the database. We show that such a decision-theoretic view of database queries gives us a unified database selection framework in which we can formally study the new best-k query problem together with the existing Boolean queries and top-k queries. We now present this framework in detail.

Without loss of generality, we represent a database abstractly as a single table, which, in reality, can be a view generated from several different tables or even one computed on the fly. For example, in Table 2, there is a single primary ID for every data tuple d_1 to d_n, and each tuple has m different attributes, from attribute$_1$ to attribute$_m$.

tuples ID	attribute$_1$	attribute$_2$	...	attribute$_m$
d_1	a_{11}	a_{12}	...	a_{1m}
d_2	a_{21}	a_{22}	...	a_{2m}
...	...	...	...	...
d_n	a_{n1}	a_{n2}	...	a_{nm}

Table 2. A single table database

Formally, we define our data set as $D = \{\vec{d_1}, ..., \vec{d_n}\}$, and each tuple $\vec{d_i}$ in D is an m degree vector representing its m attribute values. Let $T = \{\vec{t_1}, ..., \vec{t}_{|T|}\} \subseteq D$ be any subset of D and Q be any user query.

We define the following query-based *utility function* $\mathcal{U}$, which maps any subset of data tuples to a real value for decision making.

Definition 1: Utility function
$\mathcal{U}$ is called a utility function if $\mathcal{U} : T \times Q \rightarrow R^+ \cup \{0\}$, where $R^+ \cup \{0\}$ represents all non-negative real numbers.

We assume that a larger $\mathcal{U}$ value indicates a better utility, thus a better choice of data tuples. According to the decision theory [17, 10], the optimal choice of subset is the one that maximizes its utility function value. We thus can define a best-k query generally as the following decision problem:

Definition 2: Best-k query
A best-k query is to find the best tuple set T^*, which maximizes the user utility function $\mathcal{U}$. Formally,

$$T^* = \arg\max_{T \subseteq D} \mathcal{U}(T, Q) \tag{1}$$

Note that the parameter k does not explicitly appear in this definition; it can be either defined by a user through the query Q, or automatically derived by the database system from Q and the utility function $\mathcal{U}$.

Equation 1 is our *decision function*. Clearly, the utility function $\mathcal{U}$ is the essential part of our decision function and directly affects the results of a database query. Indeed, the semantics of a best-k query is mostly captured by the utility function $\mathcal{U}$, and different instantiations of this function lead to different types of database queries. We now show that the Boolean queries and top-k queries can both be formally defined in this framework as special cases of the best-k queries.

2.1 Boolean queries and top-k queries

Let Q be a Boolean query. Consider the following utility function:

$$\mathcal{U}(T,Q) = |T| \prod_{i=1}^{|T|} \delta(\vec{t_i}, Q),$$

where $\delta(\vec{t_i}, Q)$ is an indicator function defined as follows:

$$\delta(\vec{t_i}, Q) = \begin{cases} 1 & \text{if tuple } \vec{t_i} \text{ satisfies Q;} \\ 0 & \text{otherwise} \end{cases}$$

It is straightforward to prove that a best-k query with this utility function precisely captures the semantics of the Boolean query Q.

Proposition 1: A Boolean query Q returns the same results as the corresponding best-k query with $\mathcal{U}(T,Q) = |T| \prod_{i=1}^{|T|} \delta(\vec{t_i}, Q)$.

Proof: First, consider $\mathcal{U}(T_2) > 0$ and $T_1 = T_2 \cup \{\vec{t_i}\}$. If $\delta(\vec{t_i}, Q) = 1$, $\mathcal{U}(T_1) > \mathcal{U}(T_2)$; If $\delta(\vec{t_i}, Q) = 0$, then $\mathcal{U}(T_1) = 0 < \mathcal{U}(T_2)$. Thus, $\mathcal{U}(T,Q)$ is maximized when we add to T as many tuples that satisfy Q, which is precisely the semantics of a Boolean query. QED.

This utility function can be easily generalized to capture a more general type of constrain-based selection queries, where we would like to obtain as many tuples that satisfy the query constraint as possible.

Definition 3: Constraint query
A constraint query is to find the maximum tuple set T^*, which satisfies the (possibly complex) constraint specified by Q. Formally,

$$T^* = \arg\max_{T \subseteq D} |T| \delta(T,Q)$$

where $\delta(T,Q)$ is an indicator function that takes the value 1 if T satisfies the constraint specified in Q and 0 otherwise.

Note that a Boolean query is a special case of the general constraint query where $\delta(T,Q) = \prod_{i=1}^{|T|} \delta(\vec{t_i}, Q)$, i.e., the evaluation of the satisfaction status of T can be performed efficiently by checking each tuple $\vec{t_i}$. In general, however, a constraint query may have complex constraints that cannot be evaluated in this way.

Let us now consider a top-k query with the preference function being $r : T \rightarrow R^+ \cup \{0\}$. The selected k tuples based on this query (ignoring the ranking of them, which is mostly a way to present the results) would be equivalent to the results from a corresponding best-k query with the following utility function:

$$\mathcal{U}(T,Q) = \delta(|T| \leq k) \delta(T,Q) \sum_{i=1}^{|T|} r(\vec{t_i}, Q).$$

Proposition 2: A top-k query Q with a preference function r returns the same results as the corresponding best-k query with

$$\mathcal{U}(T,Q) = \delta(|T| \leq k) \delta(T,Q) \sum_{i=1}^{|T|} r(\vec{t_i}, Q).$$

In this proposition, we use another single variable δ function, which basically maps boolean value "true" to 1 and "false" to 0. We briefly prove the the proposition below.

Proof: First, we observe that the Boolean part of the top-k query captured by $\delta(T,Q)$ would allow us to include only the tuples that satisfy the Boolean constraints. Second, within this subset of tuples, we can have no more than k tuples actually included in the results due to the constraint $\delta(|T| \leq k)$. Third, since the value of r is non-negative, we would always prefer adding more tuples to the results. Finally, if we replace a tuple t in T with another tuple t' with a larger r value (i.e. $r(t', Q) > r(t, Q)$), we would always increase the utility value (i.e., $\mathcal{U}(T,Q) < \mathcal{U}(T,Q)$). Thus, the optimal subset can be constructed as follows: (1) Use the Boolean part of the query to select a working set of tuples; (2) Rank the tuples in the working set by $r(\vec{t}, Q)$; (3) Take up to k top-ranked tuples as the optimal subset. Clearly, this is exactly what a top-k query is expected to return. QED.

Note that a top-k query implies a step of ranking the returned results based on r, and this is not imposed by the treatment of a top-k query in the decision-theoretic framework because we believe that it is important to separate the selection step from the ranking step as the latter is more related to the *presentation* of the results. Indeed, the separation would allow us to consider other alternative ways of presenting the results, e.g., showing clusters of tuples or visualizing the results. Also, in a more general best-k query, the primary utility function is defined on the space of subsets, thus a ranking function defined on an individual tuple may not always be meaningful. In this paper, we focus on the selection step of best-k queries, leaving the study of how to present the selected results as future research work.

We have shown that the new best-k queries cover the existing Boolean queries and top-k queries as special cases. An examination of the utility functions derived from these two types of queries would reveal that we have made an implicit assumption that the utility value of a selected tuple is independent of each other. This assumption helps to decompose a complex utility function into simpler component functions that can be efficiently evaluated on each individual tuple. However, the independence assumption is not always true, especially when a user's preferences involve some *global* measure defined on multiple tuples. An example of this is semantic redundancy, which is only meaningful for multiple tuples, hence violates the independence assumption. We now look into more general best-k utility functions without making such an independence assumption.

2.2 Complex best-k queries

The value of $\mathcal{U}(T, Q)$ indicates the usefulness of returning the subset T in response to the query Q from the user's perspective. When a user judges how useful a set of selected tuples is, one would generally consider multiple factors and have some preference function for each of them. For example, in the example about retrieving documents from a text document database, the user cares about at least 3 factors: (1) the relevance of documents; (2) the publication date; and (3) the summary lengths. The total utility of T is thus a combination (or more precisely often a tradeoff) of all these factors.

To model these intuitions formally, we first define the space of user preference functions, Ψ_Q (or Ψ in short), as the set of all the s preference functions that a user may be interested in, i.e., $\Psi = \{\psi_1, ..., \psi_s\}$. A function $\psi \in \Psi$ measures the utility of T on some utility aspect and gives a mapping from T to a preference value in $R^+ \cup \{0\}$ with a larger preference value indicating a better utility. Thus the utility function can now be written as

$$\mathcal{U}(T, Q) \quad = \quad u(\psi_1(T, Q), ..., \psi_s(T, Q)),$$

where u is a composition function to combine all preference function together. We now take a further look at the individual preference functions ψ_i's. Intuitively, the total preference value of all the tuples in T is some aggregation of the values of individual tuples.

The simplest case is when the aggregated value is just a sum of the values of each individual tuple.

$$\psi(T, Q) = \sum_{i=1}^{|T|} m(\vec{t_i}, Q)$$

where $m(\vec{t_i}, Q) \in R^+ \cup \{0\}$ is a measure function defined on an *individual* tuple, which we refer to as a *1st-degree measure function*.

For example, the summary length preference function falls into this case, where $\psi(\vec{t_i}) = \vec{t_i}.summarylength$, and we can imagine that a global overall utility function defined on top of this preference function can be $\mathcal{U}(T, Q) = \delta(\psi(T, Q) \leq 100)$, expressing the constraint of displaying 100 words at most.

While many preference functions can be decomposed as a simple summation, there are cases when we need to consider the *interactions* of the individual tuples when aggregating their values. For instance, the redundancy among a set of selected text documents must be defined based on some of their interactions. One possibility is the following definition, which involves a *2nd-degree measure function* $m(\vec{t_i}, \vec{t_j}, Q)$.

$$\psi(T, Q) = \sum_{1 \leq i,j \leq |T|, i \neq j} m(\vec{t_i}, \vec{t_j}, Q)$$

where $m(\vec{t_i}, \vec{t_j}, Q) \in R^+ \cup \{0\}$ is a measure function defined on a *pair* of individual tuples.

An attribute that normally can be captured through a first-degree measure function may need higher-degree functions in special cases. Consider the price attribute of a book. When a user wants to find a set of books with a total price no more than \$100, a first-degree measure function would be sufficient. However, if a discount can be obtained when buying multiple books together, we may need higher-degree measure functions. While higher degree measures may be necessary, the first degree and second degree measures can serve as good approximations.

2.3 Best-k query is NP-hard

When a utility/preference function involves tuples interaction, even if the preference function can be defined based on first-degree measure functions, the decision problem is often an NP-hard problem. For example, choosing T from $D\ s.t.$
$\psi(T) = \sum_{\vec{t} \in T} \vec{t}.summarylength \leq C$ is a known NP-hard problem (i.e., subset sum). Below we show that choosing an optimal subset to optimize a preference function involving a 2nd-degree measure function is also NP-hard.

Proposition 3: A best-k query problem of choosing a size-k subset T from an n-tuple set D to minimize $\sum_{i,j} m(\vec{t_i}, \vec{t_j})$ is NP-hard.

Proof: We prove by reducing the optimization problem to a set independent problem. For any given graph G with n nodes(N_1,...,N_n), we set up a new weighted graph $\tilde{G}$ also with n nodes($\tilde{N}_1$,...,$\tilde{N}_n$). If two nodes N_i and N_j in G are connected, we add a 1-weight edge between $\tilde{N}_i$ and $\tilde{N}_j$; otherwise a 0-weight edge. Consider the nodes in $\tilde{G}$ as tuples, and the edge weights as $m(\vec{t_i}, \vec{t_j})$ values between tuples. If we can solve our minimum subset problem in polynomial time, we then can judge whether the minimum k-node subset in $\tilde{G}$ has a total weight of 0. Consequently, we solve the set independent problem of the original graph G in polynomial time. This creates contradiction since the set independent problem is an NP-complete problem. Hence the best-k problem involving a 2nd-degree measure function is NP-hard. QED.

Clearly, when interactions are involved in the utility function, even if it involves only 1st-degree measure functions, the best-k query problem is generally NP-hard. Thus it is necessary to explore heuristics and study approximate algorithms, which is detailed in Section 4.

3 Best-k Query Languages

In the previous section, we defined the best-k query problem conceptually as a decision problem with its semantics

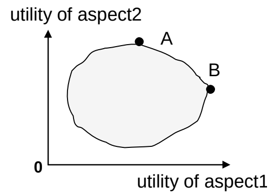

Figure 1. Multiple optimal points

mostly captured by a utility function. In this section, we discuss query languages for best-k queries that can be used to describe a utility function.

In general, a utility function for database selection may involve a combination of multiple preference functions corresponding to different aspects of utility. While each preference function alone can usually uniquely suggest an optimal subset, we will have to impose some trade-off between different preference aspects when we combine multiple preference functions. As shown in Figure 1, a plane is defined by two different preference aspects. Point A and point B are both optimal on one aspect, but neither of them is optimal on both. Choosing between A and B would have to rely on the tradeoffs. Clearly, such trade-off can only be given by a user, and different users generally have different tradeoffs.

A simple strategy that a user often follows in resolving potential conflicts between different utility aspects is to choose one single primary utility aspect to optimize and impose some thresholds for all other (secondary) utility aspects. We refer to the preference function for the primary utility aspect as the *primary preference function* and all other secondary preference functions *constraint preference functions*. In general, a constraint preference function simply imposes a constraint on the selected tuples X, so it is an indicator function whose value is 1 when X satisfies the constraint and 0 otherwise. A typical form of such a function is $\psi(X) = \delta(m(X) > c)$, where c is some constant threshold for some utility measure $m(X)$. For example, $\psi(X) = \delta(\sum_{t \in X} t.price < 100)$ imposes the constraint that the total price of the selected tuples must be below \$100.

When the utility function is a combination of a primary preference function and a number of constraint preference functions, we have $\mathcal{U}(X) = u(X) \prod_{i=1}^{r} \delta_i(X)$, where $\delta_i(X)$ is a constraint preference function and $u(X)$ is the primary preference function that we want to maximize. Clearly, the optimal subset X chosen according to such a utility function must satisfy all the constraints as well as maximize $u(X)$.

In some case, a constraint function can be further written as a product of constraints over individual tuples: $\delta_i(X) = \prod_{j=1}^{|X|} \delta_i(\vec{t_j})$. Suppose we have l such local constraint functions and $r - l = m$ global constraint functions correspondingly, the utility function can now be written as

$$\mathcal{U}(X) = u(X) \prod_{i=1}^{l} \prod_{j=1}^{|X|} \delta_i(\vec{t_j}) \prod_{i=l+1}^{r} \delta_i(X) \quad (2)$$

$$= u(X) \prod_{i=1}^{l} \prod_{j=1}^{|X|} \delta_i(\vec{t_j}) \prod_{i=1}^{m} \Delta_i(X) \quad (3)$$

where we denote $\Delta_i = \delta_{l+i}$ for easy narration in the rest of the paper.

A best-k query to express such a utility function can be specified as follows, assuming that the user is interested in the attributes $a_1, ..., a_n$ of the selected tuples.

SELECT $a_1, ...a_n$
FROM Table t
WHERE $\delta_1(t)$, ..., $\delta_l(t)$
MAXIMIZE u(X)
SUBJECTTO $\Delta_1(X)$, ..., $\Delta_r(X)$

Clearly, when the preference functions u and Δ_1,...,Δ_m are missing, our best-k query would degenerate to a standard Boolean query. If $l + 1 = r$ (i.e., only one global constraint) and $\Delta_1(X) = \delta(|X| \leq k)$, and our function $u(X)$ is $u(X) = \sum_{i=1}^{|X|} u(\vec{t_i})$, the best-k query would be a regular top-k query without the ranking part.

4 Algorithms for Executing Best-k Queries

The main challenge of best-k query problem is its NP-hard property. In this section, we study the algorithms for executing best-k queries, especially we address the global constraints parts. We first formulate the problem as a constrained optimization problem. The equation 3 and its query language in Section 3 naturally suggest a *constrained optimization* approach for executing best-k queries, where all $\delta_i(X)$ $(i = 1, ..., r)$ functions serve as the constraint part and the $u(X)$ works as an objective function. We then have the following formulation:

$$\begin{array}{ll} \text{max} & u(X) \\ \text{subject to} & \delta_1(X) \\ & ... \\ & \delta_r(X) \end{array}$$

As discussed in the previous section, the constrained optimization problem is clearly NP-hard. Interestingly, this optimization problem can be further cast as a 0-1 integer programming problem. We define a binary variable x_i for each tuple d_i in the working set. Let $x_i = 1$ if d_i is selected ($i.e.$ $d_i \in T$) and $x_i = 0$ otherwise. If $u(X)$ is p-degree measure preference function, it can be expressed by:

$$u(X) = \sum_{i_1, i_2, .., i_p} u(d_{i_1}, d_{i_2}, ..., d_{i_p}) x_{i_1} x_{i_2} ... x_{i_p},$$

where the product $x_{i_1}x_{i_2}...x_{i_p} = 1$ iff every $x_{i_j} = 1$ $(j = 1...p)$ meaning that all corresponding d_i's are selected. All δ functions can be converted in the same way. For instance, if ψ is q-degree measure, function $\delta(\psi(X) < c)$ can be converted as

$$\begin{aligned}\delta(X) &= \delta(\psi(X) < c) \\ \Longleftrightarrow & \sum_{i_1,i_2,..,i_q} \psi(d_{i_1}, d_{i_2}, ..., d_{i_q})x_{i_1}x_{i_2}...x_{i_q} < c.\end{aligned}$$

0-1 integer programming is a classic NP-hard problem. Considering the scale of typical database problems, finding efficient algorithms for executing a best-k query is quite challenging. Below, we propose a strategy for answering a best-k query efficiently.

4.1 Working set construction

To make an NP-hard problem tractable, our first strategy is to reduce the data size. Indeed, in extreme cases, when the size of data is very small, even a brute-force enumeration would be feasible. We now discuss how we can reduce the data size by constructing a working set.

In general, a database query may involve both local and global constraints. The processing of global constraints is more complex than that of local constraints. We thus address them separately in two steps: First, we construct a working set mostly based on local constraints. Second, we obtain a best-k solution from the working set by considering the global constraints as well. The benefit of this two-step approach is twofold: (1) We can reuse the existing well-established database techniques to deal with the (local) preferences over individual tuples. (2) We can efficiently remove the noninteresting data tuples, and substantially reduce the search space when processing global constraints.

In order to construct a working set, we would first explore standard relational database techniques, and generate the SQL "WHERE" clause using preference functions defined solely on individual tuples.

SELECT $a_1, ...a_n, b_1, ..., b_s$
FROM Table t
WHERE $\delta_1(t)$, ..., $\delta_l(t)$

We introduce additional SELECT attributes $b_1, ..., b_s$ to ensure that $a_1, ...a_n$ and $b_1, ..., b_s$ cover all attributes involved in $\delta 1(t)$, ..., δ_l and $\Delta_1, ..., \Delta m$, since some of them are needed by Δ's in the next step.

Depending on the size of the constructed working set in this way and whether the WHERE clause is missing, we may have to use the primary preference $u(X)$ to further reduce the size of the working set heuristically. Because generally the primary interest of a user is to maximize the objective function, if a data tuple has a very small objective value, a user is unlikely very interested in it even if it may turn out to be a part of the true optimal solution. We can thus rank data tuples in the descending order of objective values and cut off on some threshold. Clearly, this step can be done through the existing top-k searching techniques.

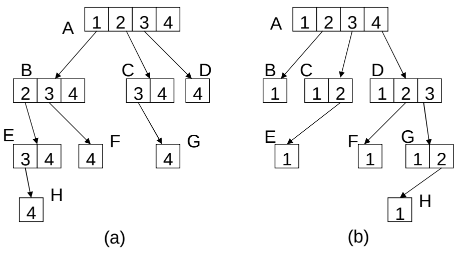

Figure 2. (a) A complete ordered set enumeration tree; (b) A complete reverse ordered set enumeration tree.

Once the working set is constructed, we conceptually obtain a new database table T from the original one t, and our task is to solve the essential best-k query by enumerate all the subsets in the working set and check the optimality of each subset. It can be submitted as the new query below.

SELECT $a_1, ...a_n$
FROM Table T
SUBJECTTO $\Delta_1(t)$, ..., $\delta_m(t)$

4.2 Ordered Set Enumeration Trees

We address the efficiency problem by proposing a data structure – *ordered set enumeration trees* (OSE), which is an extension of set enumeration trees proposed in [16].

- OSE is a tree built upon n data tuples. For example, Figure 2 (a) is such a tree over 4 tuples: 1, 2, 3, and 4.
- The root node (Node A in Figure 2 (a)) has its array including all n data tuples.
- Each node in an OSE tree includes an array, and we call the array items "buckets". For example, Node B in Figure 2 (a) has an array {2, 3, 4}, each of which is a bucket.
- The array in each node is ordered. The order is consistent in the whole tree. For example, if item 2 is before item 3 in any node, it will be always before item 3 in all other nodes. Given the order, we term all other buckets after a certain bucket in the same node as its $f$ollowup buckets. For example, in Node B, 3 and 4 are 2's followup buckets.

- Each bucket has 0 or 1 children node. We therefore can define *p*arent bucket and *p*arent node, respectively. For example, B's parent node is A and its parent bucket is $\{1\}$ in Node A. Node that 2 in Node A is *n*ot B's parent bucket.

- The items in a children node is a subset of the followup buckets of its parent bucket. For example, Node B has item 2, 3, and 4, which are 1 in Node A's followup buckets.

Intuitively, this tree enumerates all subsets of items in the root node. Any path, starting from a root bucket, is one-to-one mapped to a subset. For example, a path, including 2 in Node A, 3 in Node C, and 4 in Node G, stands for the set {2,3,4}, which is a subset of {1,2,3,4}. On the other hand, given any subset, {1,3,4} for example, we then choose the first item from the root node (1 in Node A), and choose the second item from the first one's children node (3 in B). With the same selection principle, we can find an unique path in the tree, which stands for any arbitrary subset. Since our paths are always start from a bucket in a root node, we indeed can use the ending bucket of a path to stand for this whole path. For example, 4 in Node G stands for the path: 2 in A, 3 in C, and 4 in G. In the rest of this paper, we use this abbreviation for easy description. Thus, when we mention a bucket, we often mean the path and furthermore a subset corresponding to this path.

If we define all buckets before a bucket b in the same node as **preceding buckets** of b and replace "followup buckets" with "preceding buckets" in the above definition, we obtain a new data structure **reverse ordered set enumeration tree**. Figure 2 (b) is the reverse tree corresponding to Figure 2 (a). Unless specifically stated, we discuss our algorithm on an OSE tree, but all techniques can be used in a reverse one.

An ordered set enumeration search tree has two important properties: 1) Any set T represented by a bucket is a *proper superset* of the set K represented by its parent bucket, and $|K| + 1 = |T|$. 2) The sets K_1 and K_2 that are represented by two buckets in the same node are the same size $|K_1| = |K_2|$, and also $|K_1 \cup K_2| = |K_1| + 1$. For example, the two buckets in Node E represent sets $\{1, 2, 3\}$ and $\{1, 2, 4\}$ respectively, their parent bucket represents a set $\{1, 2\}$.

4.2.1 Preference/utility values

While the general rule of the content stored buckets is for the system to easily obtain a value $\psi(X)$ for each set X presented by its corresponding bucket. Thus, depending on preference function, the contents can be different. We discuss the following two cases:

- First degree measures
  According to the tree, for any two buckets in the same node (X and Y), $X \cup Y$ has only one tuple (assuming it is x) more than X itself. Thus, clearly, $\psi(X) = \pi(\psi(X), \psi(Y))$. Operator π depends on function ψ. If ψ is a sum, π is a sum as well. If ψ is $\max$ or $\min$, π is $\max$ or $\min$ correspondingly. In the sum case, the values that a bucket need to store are $\psi(X)$ and $\psi(x)$, where x is the single item in the current bucket.

- Second degree measures
  Let us define $T = X \cap Y$. According to the definition of the OSE tree, T has precisely one tuple (assuming it is x) less than X, and X is a tuple (assuming it is y) less than $X \cup Y$. Thus, $\psi(X \cup Y) = \pi(\psi(X), \psi(Y), \psi(T), \psi(x, y))$. π again depends on ψ. Take the sum for example, $\psi(X \cup Y) = \psi(X) + \psi(Y) - \psi(T) + \psi(x, y)$. In an OSE tree , T is actually the parent bucket of X and Y. Hence, the function value $\psi(T)$ can be easily obtained from this parent bucket (by storing $\psi(T)$ into bucket T).

4.2.2 Depth-first search v.s. Breadth-first search

While all buckets represent all subsets of tuples in the root node, a tree traveling process indeed performs an exhaustive enumeration of all these subsets. Presumably, we can use both a depth-first search strategy and a breadth-first search strategy to do the traveling. However, depth first search does not need to keep all the internal nodes. Due to the large size of a OSE tree, we therefore favor the depth-first search. Moreover, when doing depth-first search, because of the existence of the primary preference function, we can almost always stop a traveling process in the middle, a property that breadth-first search does not have. This will be further discussed in the experiment part.

4.3 Child node construction and branch cutting

Pure set enumeration is probably suitable when a data size is very small. While the data size is relatively large, we have to prune the search space. We observe that the search space can be pruned when the preference functions satisfy the following property:

Definition 4: Monotonicity [1]
A preference function $\psi \in \Psi$ is said to satisfy the monotonicity property if $\forall X, Y \subseteq D$, $\psi(X \cup Y) \geq \psi(Y)$.

Monotonicity property states a function value monotonically increases with more tuples added in. Many real functions do have this property. For example, buying more will almost always increase the total spending. In this paper, we only address the problem with monotonicity property and leave others in the future work.

We start the whole algorithm from a single root node, and recursively create the different branches under each bucket of each node. Given the current node cNode and one of its bucket cBucket, we use an algorithm called Construct(cNode, cBucket) (Algorithm 1) to construct a child node nNode under cBucket.

It is easy to see that not every data tuple in cBucket's followup buckets will be put into nNode because of the monotonicity property. Considering a constraint $\delta(\psi(T) \leq c)$, if a set T violates the constraint, *i.e.* $\psi(T) > c$, we immediately conclude that all its super sets violate the constraint as well, and hence can be safely ignored. In a OSE tree, we verify the constraint satisfaction for each preference function. Any violation will prevent the corresponding bucket from being added into the nNode.

The methods of deriving $\Delta_i(B \cup K)$ from $\Delta_i(B)$ and $\Delta_i(K)$ are dependent on the specific forms of preference functions. We take a complex 2-degree measure function for example: $\psi(T) = \sum_{t_1,t_2 \in T} \psi(t_1, t_2)$. Let b_1 and b_2 be two buckets in the same node, and they represent two equal-size sets $B_1 = B \cup \{b_1\}$ and $B_2 = B \cup \{b_2\}$. To append a bucket with ID b_2 under bucket b_1, we have to compute $\psi(B_1 \cup \{b_2\})$, which is equal to $\psi(B_1) + \psi(B_2) - \psi(B) + \psi(b_1, b_2)$. This means that we store two values in each bucket t: $\psi(T)$ and $\psi(T - \{t\})$ for deriving Δ function values of its supper set easily.

```
1:  construct an empty node nNode under cBucket
2:  for each bucket b in cBucket's followup buckets do
3:      isOk = true
4:      let B be the set b stands for
5:      let K be the set cBucket stands for
6:      for each constraints $\Delta_i$ do
7:          derive $\Delta_i(B \cup K)$ from $\Delta_i(B)$ and $\Delta_i(K)$
8:          if $\Delta_i(B \cup K) == 1$ then
9:              isOk = false
10:         end if
11:     end for
12:     if isOk==true then
13:         add a new bucket $n$ with ID same to $b$ into nNode
14:     end if
15: end for
```

Algorithm 1: A child node construction algorithm. Construct(cNode, cBucket) constructs a children node nNode under the current bucket cBucket of the current node cNode.

4.3.1 Early stop conditions

Besides the branch cutting techniques above, we often have other heuristics for pruning the branches of a OSE tree. These heuristics are function specific, we thus discuss them one by one. We again use $\max u(T)$ as our objective function. For easy elaboration, we assume the returned result size k is predefined, though this condition can sometimes be relaxed in a more sophisticated way. The propositions in this section are all relatively straightforward so that we sketch the basic ideas without doing strict proofs.

Proposition 4: $u(T) = \max_{t \in T}(u(t))$
If data tuples in each node are sorted in a descending order of $u(t)$, the first set satisfying all constraints that a depth first search encounters is an optimal solution.

Note that this makes the objective function max $\max_{t \in T}(u(t))$. Because of the descending order on the utility function, the biggest item in each set is the one in the root bucket, which is clearly the biggest item in all subsets rooted in this bucket.

Proposition 5: $u(T) = \min_{t \in T}(u(t))$
If data tuples sorted in a descending order of $u(t)$, the first set satisfying all constraints that a depth first search encounters in a **reverse ordered set enumeration tree** is an optimal solution.

This proposition is applied when users ask for the smallest item in a set is largest. Clearly, it is the corresponding example of the previous proposition.

Proposition 6: $u(T) = \sum_t u(t)$
Assume that the current best-k tuple set K, satisfying all constraints, has objective value $u(K)$. The traveling is now at bucket x in node X. We do not need to generate b's child node if either of the following two conditions holds: 1) b has less than $k - |X|$ followup buckets; 2) $[u(B) + \sum_{t \in \Omega} u(t)] \leq u(K)$, where Ω stands for the $k - |B|$ best tuples in b's followup buckets.

The first condition is saying that one does not have to continue a search if no enough tuple left. For example, if one needs $k = 3$ items in Figure 2, one does not need to generate Node D at all from $\{3\}$ in Node A, because no enough followup buckets left. The second one shows that even if there are enough tuples left, searching can stop if the maximum possible set is less than the best set that has already been obtained. Again, if one needs 2 item in Figure 2, but the largest two item from $\{2, 3, 4\}$ even smaller than the current best solution, one does not have to construct Node C, Node D and Node G.

Compared with algorithms in some related work, our algorithm is novel in that we address the new best-k problem in a general framework – working set construction and item selections. Our proposed "ordered enumeration tree" is also different from the original enumeration tree at that it can support several pruning heuristics. Moreover, compared with [12, 12], our algorithm is targeting at the exact solutions and also the best-k selection is also different from simple aggregation operators.

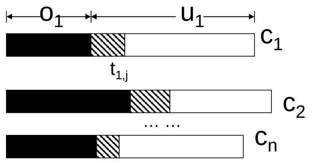

Figure 3. An illustration of the greedy algorithm.

4.4 Approximate Answers

Although the results in the previous subsection show that the depth-first search algorithm is relatively efficient, it may still not be feasible to run it on a very large data set due to the intrinsic NP-hard property. We have two different ways to make the approximation solution: First, we stop our previous algorithm in the middle to return the current best solution. We call it any-time stopping. Second, we develop the greedy algorithm. Below, we introduce the greedy strategy, and we will compare the two strategies in the experiment section.

A greedy algorithm first selects a single best tuple, and gradually adds new tuples by choosing the next best one given the current selected ones. The key decision in this greedy algorithm is to decide what the next best tuple is. In our problem, this has to be decided by multiple constraint utility functions. When we add a new tuple to a set, the cost value of every utility function may potentially increase. To avoid having a cost value above a desired bound, we should avoid "consuming" the critical utilities. A critical utility is one that is closest to be run out. We use Figure 3 to illustrate this idea. Assume we have n different utility functions, and their constraint bounds are c_i, $i = 1, ..., n$, respectively. Black area (o_i) stands for the utilities used/consumed by the previously selected tuples , and the other areas (u_i) are the utilities left. $o_i + u_i = c_i$. When a new tuple j is selected, it occupies the tilt areas($t_{i,j}$). Assume its objective gain is b_j. We want to select a tuple, which will occupy minimum percentage of the remaining area (thus consuming least utilities), while having maximum objective gain. We formalize this heuristic criterion with the following measure:

$$\arg\max_j \{ \frac{b_j}{\max_i \frac{t_{i,j}}{u_i}} \times \prod_i \delta(\frac{t_{i,j}}{u_i} \leq 1) \},$$

where δ is an indicator function($\delta(\text{true}) = 1$, $\delta(\text{false}) = 0$). The tuple that can maximize the measure above is selected as the next best one.

bound1	bound2	cons. pref. 1	cons. pref. 2	objective
$+\infty$	$+\infty$	498.0	221.4	754.8
300	450	419.7	157.4	441.9
250	400	411.8	147.3	391.7
200	350	397.4	143.0	344.4

Table 3. Trade-off between the primary objective and the constrained preferences

5 Experiments

In this section, we evaluate the proposed algorithms using simulated data sets. Since the construction of the working set is based on well studied database techniques, such as top-k queries, we focus our evaluation on the essential problem of best-k query, i.e. select items from a given working set. As discussed before, this problem is in general an NP-hard problem. Therefore, the evaluation focuses on the running speed as well as the closeness to the optimal solution if an approximation is applied.

SELECT *
FROM Table T
MAXIMIZW u(T)
SUBJECTTO $\Delta_1(t)$, ..., $\Delta_m(t)$
LIMIT k

Specifically, we study the impact of following parameters:
1) n: the number of data tuples in T
2) k: the number of returned tuples
3) m: the number of constraints

To avoid any bias of random sampling, every number that we report is the arithmetic mean of 10 independent simulation with the identical simulation parameters. All experiments are done on a Linux platform with a processor of 2.4GHZ cpu speed and 1G memory.

5.1 Trade-off between the primary preference and the constrained preferences

We first examine the difference with or without the constraint functions. We use an objective function and two constraints: one is 1-degree interaction and the other is 2-degree interaction. We assume $n = 200$ and $k = 8$. The results are shown in Table 3. As expected, when setting different bounds, we observe the output results with different constraint preference values(Column 3 & 4) and objective values(Column 5). The trade-off is very clear. When applying a tight constraint, a certain amount of the objective value is sacrificed.

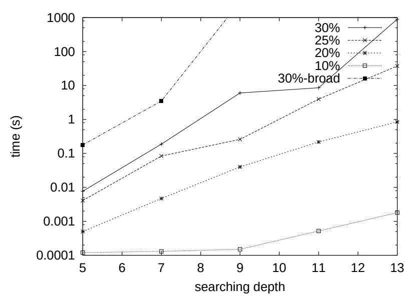

Figure 4. Scalability on the searching depth.

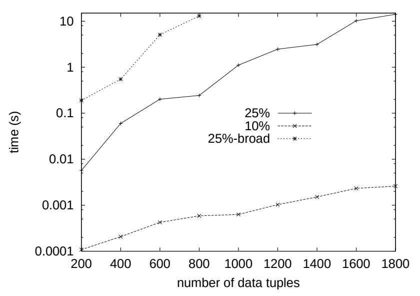

Figure 5. Scalability on the number of tuples.

5.2 Depth-first search and breadth-first search

Since executing a best-k query is computationally expensive, it is necessary to study the efficiency of the proposed algorithms. A comparison between our algorithms with brute-force enumeration(*i.e* without employing the the branches cutting with the monotonicity property) would not be very interesting, because the cutting can definitely prune the searching branches. Indeed, with reasonable tight bounds, we find that more than 99% nodes in a depth-first tree can be pruned by monotonicity property. This will clearly largely accelerate the searching process. We thus compare the depth-first search versus breadth-first search both with branch-cutting.

We first evaluate the scalability on the number of selected tuples, k, by fixing the total number of tuples to $n = 200$. Since all preference values are with a uniform distribution $[1 : 100]$, and thus the expected value of each tuple is 50, we use the percentage of the expected values as the constraint bounds. For example, if the search depth is 5, the expected value of 5 tuples is $50 \times 5 = 250$. 30% bounds will be $250 * 0.3 = 75$. Figure 4 shows the scalability of the algorithm over k. The x-axis is the search depth while the y-axis is the running time in log scale. Given the inherent complexity of the problem, it is not surprising that the figure still shows an exponential curve. However, as shown in the figure, the depth-first search algorithm is several orders of magnitude faster than the breadth-first search, which cannot finish in hours after $k > 9$.

We then evaluate the algorithm's scalability over the total number of tuples. By fixing k to 5, and varying n from 200 to 2000, Figure 5 shows two curves representing 25% and 10% bounds, respectively. Clearly, we observe a sub-exponential property in this figure, which demonstrates the good scalability of our algorithm. Although the worst case is always exponential, our algorithm can often stop much earlier to return an optimal answer without actually going through all the branches. We also observe the depth-first algorithm outperforms the breadth-first one, which again cannot finish experiments when n is larger than 1000.

the number of constraints	3	4	5	6
running time(s)	49.91	4.68	1.00	0.66

Table 4. The impacts of the number of constraints

5.3 The impacts on the number of constraints

Presumably, when we have more constraints(i.e m is larger), the search space can be more easily pruned. In this part, we study the impact of the value m. We fix $n = 1000$, $k = 8$ and all bounds at 50% level. When varying the number of the constrained preferences, we test the running time. The results are shown in Table 4. We see clearly the descending trend of the running time with respect to the number of constraints. When there are three constraints, it needs around 50 seconds, but six constraints reduces the process within less 1 second.

5.4 Experiments on early stop conditions

We examine the effectiveness of those early stop conditions. By the same process, we fix $k = 8$ and vary the value of n, and generate two curves in Figure 6. From the figure, we can see that the algorithm performance is improved with applying all these early stop conditions.

5.5 Any-time stopping

In this section, we evaluate the any-time stopping property of the depth first search algorithm. The test is done with $n = 500$ tuples, $k = 8$, and 25% constraint bounds. We use 6 sets of data for simulation. (Note that we do not do 10 set averaging process for this experiments, because it is for studying the individual set's property.) We compare the time when the program achieves optimal solutions(2^{nd} row) with the time when the program (with early stopping

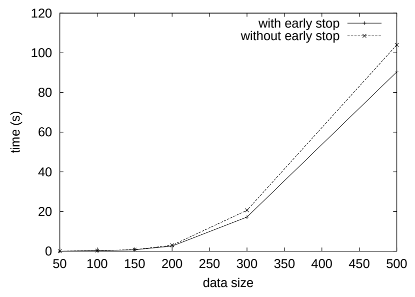

Figure 6. Early stop conditions.

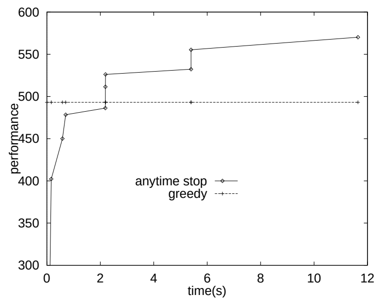

Figure 7. Performance comparison between any time stopping and greedy algorithms.

applied) actually stops(3^{rd} row) in Table 5. The last row shows the percentage of second row over third row. It is clear that the program obtains optimal solution much earlier than the time it actually stops, when the optimum is guaranteed. The percentage is usually no larger than 15%.

Figure 7 shows how the performance increases as we allow the program to run longer. We only plotted the x-axis up to the time of finding the optimal solution. We also observe that the performance is improved quickly in the beginning, and becomes flat soon after, though it still increases slightly when more time is spent. Thus, empirically, stopping in the middle usually does not incur too much quality loss.

We also compare the two approximation strategies – any time stopping and greedy algorithm in Figure 7. The curve for the greedy algorithm is all flat because it finishes in a very short period. Clearly, the greedy algorithm is much faster than the any-time stopping strategy, but when running time is reasonably long, the any-time stopping method can outperform the greedy algorithm.

6 Related Works

Our work is probably most related to the study of top-k queries. The top-k query problem was first proposed in [8], and has been studied in several later works [9, 4, 13, 6, 2, 15, 5, 14], in which many different ways of efficiently executing a top-k query have been proposed. Our work differs from these works in that we address a more general type of queries that can cover top-k queries as special cases.

The need for relaxing/extending the traditional Boolean queries has also been recognized in many other works. For example, probabilistic relational algebra was studied in [7, 11] to incorporate fuzzy concepts in a relational database model.

The study of aggregation operations in database systems [20, 12] is related to our work in that the computation of complex preference functions often involves the computation of aggregation operations, though the preference functions are often more complex than the single aggregation operators, which are the main focus of the existing work on aggregation operations.

Our work is also related to our previous work on a risk minimization framework for information retrieval [18, 19], in which the retrieval problem has been modeled as a decision problem involving uncertainties and the Bayesian decision theory is applied to formalize the framework. In current work, we are concerned with deterministic decision problems and focus on how to efficiently compute an approximately optimal decision.

7 Conclusions and Future Work

As exploratory queries become more and more popular, the study of complex database queries attracts much attention recently. In this paper, we propose and study a new database query problem, called best-k query, which generalizes the existing top-k query by relaxing its Independence assumption on scoring each selected tuple. In order to unify different types of database queries, we model the database selection problem generally as a decision problem, in which a database system would respond to a database query by selecting a subset of objects that optimize a certain utility function defined on the objects. We show that such an optimization framework covers the boolean query search, the top-k query search, and the best-k query search all as special cases, corresponding to different utility functions. The proposed framework provides a roadmap for exploring complex database queries with different levels of complexity.

We further study how to efficiently evaluate a best-k query. We prove that finding answers to a best-k query is an NP-hard problem and study efficient algorithms to find answers to a best-k query. We note that many practically interesting utility functions satisfy a monotonicity property, which can be exploited to dramatically prune the search space. Accordingly, we propose to use a branch-cutting strategy to prune the search space in both breadth-first and depth-first search. We evaluate these algorithms on simu-

dataset	1	2	3	4	5	6
optimal time	2.53	5.28	0.83	0.48	1.27	0.118
total time	116.98	33.27	175.48	13.52	10.16	11.02
opt/total	2.2%	15.9%	0.47%	3.6%	12.5%	1.1%

Table 5. Any time stop.

lated data sets. The results show that, for any non-trivial best-k problems, the depth-first search strategy is more feasible and much faster than the breadth-first search strategy because the latter easily runs out of memory. Moreover, the depth-first branch-cutting algorithm has the additional benefit of being an anytime algorithm, which allows a user to make flexible tradeoff between the optimality of results and the running time of the query. Experiment results show that, using anytime stopping, the depth-first branch-cutting algorithm achieves better approximation than a greedy algorithm for finding approximate answers. Thus among the algorithms we explored, the depth-first branch-cutting algorithm is the best.

There are several issues for further exploring and extending the work presented here: (1) More efficient algorithms: The depth-first search algorithm can work on any monotonic preference functions. It is possible to develop even more efficient algorithms by making additional assumptions about the preference functions. For example, if the constraint functions only involve first-degree measure functions, there may be additional heuristics that can be exploited to further speed up the algorithm. (2) Nonmonotonic preference functions: Some preference functions (e.g., aggregation by average) do not satisfy the monotonicity property, so we cannot apply the proposed depth-first algorithm. It would be very interesting to study efficient algorithms that can answer best-k queries involving such preference functions.

References

[1] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *VLDB'94*, pages 487–499, 1994.

[2] R. Agrawal and E. L. Wimmers. A framework for expressing and combining preferences. In *SIGMOD'00*, pages 297–306, 2000.

[3] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE'02*, pages 369–380, 2002.

[4] K. C.-C. Chang and S. won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD'02*, pages 346–357, 2002.

[5] S. Chaudhuri, G. Das, V. Hristidis, and G. Weikum. Probabilistic ranking of database query results. In *VLDB*, pages 888–899, 2004.

[6] S. Chaudhuri and L. Gravano. Evaluating top-k selection queries. In *VLDB'99*, pages 397–410, 1999.

[7] T. Eiter, T. Lukasiewicz, and M. Walter. A data model and algebra for probabilistic complex values. *Annals of Mathematics and Artificial Intelligence*, pages 205–252, 2001.

[8] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, 1996.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS'01*, 1994.

[10] P. C. Fishburn. *Nonlinear Preference and Utility Theory*. The John Hopkins Press, 1988.

[11] N. Fuhr and T. Rölleke. A probabilistic relational algebra for the integration of information retrieval and database systems. *ACM Transactions on Information Systems*, pages 32–66, 1997.

[12] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB'03*, pages 778–789, 2003.

[13] U. Güntzer, W.-T. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *VLDB'00*, pages 419–428, 2000.

[14] L. Guo, J. Shanmugasundaram, K. Beyer, and E. Shekita. Efficient inverted lists and query algorithms for structured value ranking in update-intensive relational databases. In *ICDE '05: Proceedings of the 21st International Conference on Data Engineering (ICDE'05)*, pages 298–309, 2005.

[15] V. Hristidis, N. Koudas, and Y. Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. In *SIGMOD '01: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 259–270, 2001.

[16] R. Rymon. Search through systematic set enumeration. In *In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 539–550, 1992.

[17] D. J. White. *Decision Theory*. Aldine Publishing Company, 1969.

[18] C. Zhai. *Risk Minimization and Language Modeling in Text Retrieval*. PhD thesis, Carnegie Mellon University, 2002.

[19] C. Zhai, W. W. Cohen, and J. Lafferty. Beyond independent relevance: Methods and evaluation metrics for subtopic retrieval. In *SIGIR'03*, pages 10–17, 2003.

[20] D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *PODS*, 2002.