Query Reranking As A Service

Abolfazl Asudeh[‡], Nan Zhang[††], Gautam Das[‡]
[‡]University of Texas at Arlington; [††]George Washington University
[‡]{ab.asudeh@mavs, gdas@cse}.uta.edu, [††]nzhang10@gwu.edu

ABSTRACT

The ranked retrieval model has rapidly become the de facto way for search query processing in client-server databases, especially those on the web. Despite of the extensive efforts in the database community on designing better ranking functions/mechanisms, many such databases in practice still fail to address the diverse and sometimes contradicting preferences of users on tuple ranking, perhaps (at least partially) due to the lack of expertise and/or motivation for the database owner to design truly effective ranking functions. This paper takes a different route on addressing the issue by defining a novel *query reranking problem*, i.e., we aim to design a third-party service that uses nothing but the public search interface of a client-server database to enable the on-the-fly processing of queries with any user-specified ranking functions (with or without selection conditions), no matter if the ranking function is supported by the database or not. We analyze the worst-case complexity of the problem and introduce a number of ideas, e.g., on-the-fly indexing, domination detection and virtual tuple pruning, to reduce the average-case cost of the query reranking algorithm. We also present extensive experimental results on real-world datasets, in both offline and live online systems, that demonstrate the effectiveness of our proposed techniques.

1. INTRODUCTION

Problem Motivation: The ranked retrieval model has rapidly replaced the traditional Boolean retrieval model as the de facto way for query processing in client-server (e.g., web) databases. Unlike the Boolean retrieval model which returns all tuples matching the search query selection condition, the ranked retrieval model orders the matching tuples according to an often proprietary ranking function, and returns the top-k tuples matching the selection condition (with possible page-turn support for retrieving additional tuples).

The ranked retrieval model naturally fits the usage patterns of client-server databases. For example, the short attention span of clients such as web users demands the most desirable tuples to be returned first. In addition, to achieve a short response time (e.g., for web databases), it is essential to limit the length of returned results to a small value such as k. Nonetheless, the ranked retrieval model also places more responsibilities on the web database de-

signer, as the *ranking function* design now becomes a critical feature that must properly capture the need of database users.

In a practical situation, different users often have diverse and sometimes contradicting preferences on numerous factors. Even more importantly, many database owners simply lack the expertise, resources, or even motivation (e.g., in the case of government web databases created for policy or legal compliance purposes) to properly study the requirements of their users and design the most effective ranking functions. For example, many flight-search websites, including Kayak, Google Flights, Sky Scanner, Expedia, and Priceline offer limited ranking options on a subset of the attributes, that, for example, does not help ranking based on cost per mileage. Similar limitations apply to the websites such as Yahoo! Autos (resp. Blue Nile), for ranking, for example, based on mileage per year (resp. summation of depth and table percent). As a result, there is often a significant gap, in terms of both design and diversity, between the ranking function(s) supported by the client-server database and the true preferences of the database users. The objective of this paper is to define and study the *query re-ranking* problem which bridges this gap for real-world client-server databases.

Query Re-Ranking: Given the challenge for a real-world database owner to provide a comprehensive coverage of user-preferred ranking functions, in this paper we develop a *third-party query re-ranking service* which uses nothing but the public search interface of a client-server database to enable the on-the-fly processing of queries with user-specified ranking functions (with or without selection conditions), no matter if the ranking function is supported by the database or not. This query re-ranking service can enable a wide range of interesting applications. For example, one may build a personalized ranking application using this service, offering users with the ability to remember their preferences across multiple web databases (e.g., multiple car dealers) and apply the same personalized ranking over all of them despite the lack of such support by these web databases. As another example, one may use the re-ranking service to build a dedicated application for users with disabilities, special needs, etc., to enjoy appropriate ranking over databases that do not specifically tailor to their needs.

There are two critical requirements for a solution to the query re-ranking service: First, the output query answer must precisely follow the user-specified ranking function. Second, the query re-ranking service must minimize the number of queries it issues to the client-server database in order to answer a user-specified query. This requirement is crucial for two reasons: First is to ensure a fast response time to the user query. Second is to reduce the burden on the client-server database, as many real-world ones, especially web databases, enforce stringent rate limits on queries from the same IP address or API user (e.g., Google Flight Search API allows only 50 free queries per user per day).

Problem Novelty: While extensive studies have focused on translating an unsupported query to multiple search queries supported by a database, there has not been research on the *translation of ranking* requirements of queries. Related work includes the studies on crawling client-server databases [15], as a baseline solution for query re-ranking is to first crawl all tuples from the client-server database, and then process the user query and ranking function locally. The problem, however, is the high query cost, since the number of queries to be issued to the client-server database for crawling is proven [15] to range from at least linear to the database size in the best-case scenario to quadratic and higher in worse cases.

Another seemingly simple solution is for the third-party service to retrieve more than k tuples matching the user query, say $h \cdot k$ tuples by using the "page-down" feature provided by a client-server database (or [16, 17] when such a feature is unavailable), then locally re-rank the $h \cdot k$ tuples and return the top-k ones. There are two problems with this solution. First, since many client-server databases choose not to publish the design of their proprietary ranking functions (e.g., simply naming it "rank by popularity" in web databases), results returned by this approach will have unknown error unless all tuples satisfying the user query are crawled. Second, when the database ranking function differs significantly from the user-specified one, this approach may have to issue many page-downs (i.e., a large h) in order to retrieve the real top-k answers according to the user-specified ranking function.

Finally, note that our problem stands in sharp contrast with existing studies on processing top-k queries over traditional databases using pre-built indices and/or materialized views (e.g., [4,10]). The key difference here is the underlying *data access model*: Unlike prior work which assume complete access to data, we are facing a restricted, top-k, search interface provided by the database.

Outline of Technical Results: We start by considering a simple instance of the problem, where the user-desired ranking function is on a single attribute, and developing Algorithm 1D-RERANK to solve it. Note that this special, 1D, case not only helps with explaining the key technical challenges of query reranking, but also can be surprisingly useful for real-world web databases. For example, while flight search websites like Kayak offer the ability to specify a range query on layover time, it does not support ranking according to the attribute. The 1D-RERANK algorithm handily addresses this need by enabling a "Get-Next" primitive - i.e., upon given a user query q, an attribute A_i, and the top-h tuples satisfying q according to A_i, it finds the "next", i.e., $(h+1)$-th ranked, tuple.

In the development of 1D-RERANK, we rigidly prove that, in the worst-case scenario, retrieving even just the top-1 tuple requires crawling of the entire database. Nonetheless, we also show that the practical query cost tends to be much smaller. Specifically, we found a key factor (negatively) affecting query cost to be what we refer to as "dense regions" - i.e., a large number of tuples clustering together within a small interval, that may be queried again and again for the processing of different queries. This prompts us to propose an *on-the-fly indexing* idea that detects such dense regions and proactively crawls top-ranked tuples in it to avoid the waste on processing future user queries.

To solve the general problem of query reranking for any arbitrary user-desired ranking function (rather than just 1D), a seemingly simple solution is to directly apply a classic top-k query processing algorithm that leverages sorted access to each attribute, e.g., Fagin's algorithm [9], by calling the "Get-Next" primitive provided by 1D-RERANK as a subroutine. The problem with this is that it incurs a significant waste of queries when applied to client-server databases, mainly because it fails to leverage the multi-predicate (conjunctive) queries supported by the underlying database.

To address the issue, we develop MD-RERANK (i.e., Multi-Dimensional Rerank), a query re-ranking algorithm that identifies a small number of multi-predicate queries to directly retrieve the top-k tuples according to a user query. We note a key difference between the 1D and MD cases: In the 1D case, a single query is enough to cover the subspace outranking a given tuple, while the MD case requires a much larger number of queries due to the more complex shape of the subspace. We develop two main ideas, namely *direct domination detection* and *virtual tuple pruning*, to significantly reduce the query cost for MD-RERANK. In addition, like in the 1D case, we observe the high query cost incurred by "dense regions", and include in MD-RERANK our on-the-fly indexing idea to reduce the amortized cost of query re-ranking.

Our contributions also include a comprehensive set of experiments on real-world web databases, both in an offline setting (for having the freedom to control the database settings) and through online *live* experiments over real-world web databases. Specifically, we constructed a Top-k web search interface in the offline experiment, and evaluated the performance of the algorithms in different situations, by varying the parameters such as database size, system-k, and system ranking function. In addition we also tested our algorithms live online over two popular websites, namely Yahoo! Autos and and Blue Nile, the largest diamond online retailer. The experiment results verify the effectiveness of our proposed techniques and their superiority over the baseline competitors.

2. PRELIMINARIES

2.1 Database Model

Database: Consider a client-server database D with n tuples over m ordinal attributes $A_1, \ldots, A_m$. Let the value domain of A_i be $V(A_i) = \{v_{i1}, \ldots, v_{i|V(A_i)|}\}$. The database may also have other categorical attributes $B_1, \ldots, B_{m'}$. But since they are usually not part of any ranking function, they are not the focus of our attention for the purpose of this paper. We assume each tuple t to have a none-NULL value on each (ordinal) attribute A_i, which we refer to as $t[A_i]$ ($t[A_i] \in V(A_i)$). Note that if NULL values do exist in the database, the ranking function usually substitutes it with another default value (e.g., the mean or extreme value of an attribute). In that case, we simply consider the occurrence of NULL as the substituted value. In most part of the paper, we make the general positioning assumption [20], i.e., each tuple has a unique value on each attribute, before introducing a simple post-processing step that removes this assumption in § 5.

Query Interface: Most client-server databases allow users to issue certain "simplistic" search queries. Often these queries are limited to conjunctive ones with predicates on one or a few attributes. Examples here include web databases, which usually allow such conjunctive queries to be specified through a form-like web search interface. Formally, we consider search queries of the form

q: SELECT * FROM D WHERE $A_{i_1} \in (v_{i_1}, v'_{i_1})$ AND $\cdots$ AND $A_{i_p} \in (v_{i_p}, v'_{i_p})$ AND *conjunctive predicates on* $B_1, \ldots, B_{m'}$,

where $\{A_{i_1}, \ldots, A_{i_p}\} \subseteq \{A_1, \ldots, A_m\}$ is a subset of ordinal attributes, and $(v_{i_j}, v'_{i_j}) \subseteq V(A_{i_j})$ is a range within the value domain of A_{i_j}.

A subtle issue here is that our definition of q only includes open ranges (x, y), i.e., $x < A_i < y$, while real-world client-server databases may offer close ranges $[x, y]$, i.e., $x \leq A_i \leq y$, or a combination of both (e.g., $(x, y]$). We note that these minor variations do not affect the studies in this paper, because it is easy to derive the answer to q even when only close ranges are allowed by

database: One simply needs to find a value arbitrarily close to the limits, say $x + \epsilon$ and $y - \epsilon$ with an arbitrarily small $\epsilon > 0$, and substitute (x, y) with $[x + \epsilon, y - \epsilon]$. In the case where the value domains are discrete, substitutions can be made to the closest discrete value in the domain.

As discussed in § 1, once a client-server database receives query q from a user, it often limits the number of returned tuples to a small value k. We use q to refer to the set of tuples *actually returned* by q, $R(q)$ to refer to the the set of tuples matching q (which can be a proper superset of the returned tuples q when there are more than k returning tuples, and $|R(q)|$ to refer to the number of tuples matching q. When $|R(q)| > k$, we say that q *overflows* because only k tuples can be returned. Otherwise, if $|R(q)| \in [1, k]$, we say that q returns a *valid* answer. At the other extreme, we say that q *underflows* when it returns empty, i.e., $|R(q)| = 0$.

System Ranking Function: In most parts of the paper, we make a conservative assumption that, when $|R(q)| > k$, the database selects the k returned tuples from $R(q)$ according to a proprietary *system ranking function* unbeknown to the query reranking service. That is, we make *no* assumption about the system ranking function whatsoever. In § 5, we also consider cases where the database offers more ranking options, e.g., ORDER BY according to a subset of ordinal attributes.

2.2 Problem Definition

The objective of this paper is to enable a *third-party query reranking service* which enables a user-specified ranking function for a user-specified query q, when the query q is supported by the underlying client-server database but the ranking function is *not*.

User-Specified Ranking Functions: We allow a user of the query reranking service to specify a *user-specified ranking function* $S(q, t)$ which takes as input the user query q and one or more ordinal attributes (i.e., $A_1, \ldots, A_m$) of a tuple t, and outputs the ranking score for t in processing q. The *smaller* the score $S(q, t)$ is, the *higher ranked* t will be in the query answer, i.e., the more likely t is included in the query answer when $R(q) > k$. Without causing ambiguity, we also represent $S(q, t)$ as $S(t)$ when the context (i.e., the user query being processed) is clear.

We support a wide variety of user-specified ranking functions with only one requirement: *monotonicity*. Given a user query q, a ranking function $S(t)$ is monotonic if and only if there exists an order of values for each attribute domain, which we represent as $\prec$ with $v_1 \prec v_2$ indicating v_1 being higher-ranked than v_2, such that there does not exist two possible tuple values t_1 and t_2 with $S(t_1) < S(t_2)$ yet $t_2[A_i] \prec t_1[A_i]$ for all $i \in [1, m]$.

Intuitively, the definition states that if t_1 outranks t_2 according to $S(\cdot)$, then t_1 has to outrank t_2 on at least one attribute according to the order $\prec$. In other words, t_1 cannot outrank t_2 if it is dominated [5] by t_2. Another interesting note here is that we do *not* require all user-specified ranking functions to follow the same attribute-value order $\prec$. For example, one ranking function may prefer higher prices while the other prefers lower prices.

Performance Measure: To enable query reranking, we have to issue a number of queries to the underlying client-server database. It is important to understand that the most important efficiency factor here is *the total number of queries* issued to the database, not the computational time. The rational behind it is that almost all client-server databases, enforce certain *query-rate limit* by allowing only a limited number of queries per day from each IP address, API account, etc.

Problem Definition: In this paper, we consider the problem of query reranking in a "Get-Next", i.e., incremental processing, fash-ion. That is, for a given user query q, a user-specified ranking function S, and the top-h tuples satisfying q according to S, we aim to find the No. $(h + 1)$ tuple. When $h = 0$, this means finding the top-1 for given q and S. One can see that finding the top-h tuples for q and S can be easily solved by repeatedly calling the Get-Next function. The reason why we define the problem in this fashion is to address the real-world scenario where a user first retrieves the top-h answers and, if still unsatisfied with the returned tuples, proceeds to ask for the No. $(h + 1)$. By supporting incremental processing, we can progressively return top answers while paying only the incremental cost.

> QUERY RERANKING PROBLEM: Consider a client-server database D with a top-k interface and an arbitrary, unknown, system ranking function. Given a user query q, a user-specified monotonic ranking function S, and the top-h ($h \geq 0$ can be greater than, equal to, or smaller than k) tuples satisfying q according to S, discover the No. $(h + 1)$ tuple for q while minimizing the number of queries issued to the client-server database D.

3. 1D-RERANK

We start by considering the simple 1D version of the query reranking problem which, as discussed in the introduction, can also be surprisingly useful in practice. Specifically, for a given attribute A_i, a user query q, and the h tuples having the minimum values of A_i among $R(q)$ (i.e., tuples satisfying q), our goal here is to find tuple $t(q, A_i, h + 1)$, which satisfies q and has the $(h + 1)$-th smallest value on A_i among $R(q)$, while minimizing the number of queries issued to the underlying database.

3.1 Baseline Solution and Its Problem

3.1.1 1D-BASELINE

Baseline Design: Since our focus here is to discover $t(q, A_i, h + 1)$ given q, A_i and h, without causing ambiguity, we use t_{h+1} as a short-hand representation of $t(q, A_i, h + 1)$. A baseline solution for finding t_{h+1} is to start with issuing to the underlying database query q_1: SELECT * FROM D WHERE $A_i > t_h[A_i]$ AND $Sel(q)$, where $Sel(q)$ represents all selection conditions specified in q. If $h = 0$, this query simply becomes SELECT * FROM D WHERE $Sel(q)$.

Note that the answer to q_1 must return non-empty, because otherwise it means there are only h tuples matching q. Let a_1 be the one having minimum A_i among all returned tuples. Given a_1, the next query we issue is q_2: WHERE $A_i \in (t_h[A_i], a_1[A_i])$ AND $Sel(q)$. In other words, we narrow the search region on A_i to "push the envelop" and discover any tuple with even "better" A_i than what we have seen so far.

If q_2 returns empty, then $t_{h+1} = a_1$. Otherwise, we can construct and issue q_3, q_4, $\ldots$, in a similar fashion. More generally, given a_j being the tuple with minimum A_i returned by q_j, the next query we issue is q_{j+1}: WHERE $A_i \in (t_h[A_i], a_j[A_i])$ AND $Sel(q)$. We stop when q_{j+1} returns empty, at which time we conclude $t_{h+1} = a_j$. Algorithm 1, 1D-BASELINE, depicts the pseudo-code of this baseline solution.

Leveraging History: An implementation issue worth noting for 1D-BASELINE is how to leverage the historic query answers we have already received from the underlying client-server database. This applies not only during the processing of a user query, but also across the processing of different user queries.

During the process of user query q, for example, we do not have to start with the range of $A_i \in (t_h[A_i], \infty)$ as stated in the basic algorithm design. Instead, if we have already "seen" tuples in

$R(q)$ that have $A_i > t_h[A_i]$ in the historic query answers, then we can first identify such a tuple with the minimum A_i, denoted by t', and then start the searching process with $A_i \in (t_h[A_i], t')$, a much smaller region that can yield significant query savings, as shown in the query cost analysis below. More generally, this exact idea applies across the processing of different user queries, by identifying tuples in historic query answers that match the user query being processed. By doing so, the more queries we have processed, the more likely we can prune the search space for t_{h+1} based on historic query answers.

Algorithm 1 1D-BASELINE

1: $t_{h+1} = \text{argmin}_{t[A_i]}\{t \in \text{history} \mid t[A_i] > t_h[A_i]\}$
2: $T = \text{Top-}k(\text{WHERE } t_{h+1}[A_i] > A_i > t_h[A_i] \text{ AND } Sel(q))$
3: **while** T is overflow
4:     $t_{h+1} = \text{argmin}_{t[A_i]}\{t \in T\}$
5:     $T = \text{Top-}k(\text{WHERE } t_{h+1}[A_i] > A_i > t_h[A_i] \text{ AND } Sel(q))$
6: **return** t_{h+1}

3.1.2 Negative Result: Lower Bound on Worst-Case Query Cost

While simple, 1D-BASELINE has a major problem on query cost, as it depends on the correlation between A_i and the system ranking function which we know nothing about and has no control over. For example, if the system ranking function is exactly according to A_i, then the query cost of finding t_{h+1} is 2: q_1 returns t_{h+1} and q_2 returns empty to confirm that t_{h+1} is indeed the "next" tuple. On the other hand, if the system ranking function is the exact opposite to A_i, then the query cost for the baseline solution is exactly $|R(q)| + 1$ in the worst-case scenario (when $k = 1$), because every tuple satisfying q will be returned before t_{h+1} is revealed at the end. While it might be tempting to try to "adapt to" such ill-conditioned system ranking functions, the following theorem actually shows that the problem is not fixable in the worst-case sense. Specifically, there is a lower bound of n/k on the query cost required for query reranking given the worst-case data distribution and worst-case system ranking function.

THEOREM 1. $\forall n > 1$, *there exists a database of n tuples such that finding the top-ranked tuple on an attribute through a top-k search interface requires at least n/k queries that retrieve all the n tuples.*

PROOF. Without loss of generality, consider a database with only one attribute A and an unknown ranking function. Let (v_0, v_∞) be the domain of A. Note that this means (1) the query re-ranking algorithm can only issue queries of the form SELECT * FROM D WHERE $A \in (v_1, v_2)$, where $v_0 \leq v_1 < v_2 \leq v_\infty$, (2) the returned tuples will be ranked in an arbitrary order, and (3) the objective of the query re-ranking algorithm is to find the tuple with the smallest A.

For any given query re-ranking algorithm $\mathcal{R}$, consider the following query processing mechanism $\mathcal{Q}$ for the database: During the processing of all queries, we maintain a min-query-threshold v_q with initial value v_∞. If a query q issued by $\mathcal{R}$ has lower bound *not* equal to v_0, i.e., q: WHERE $A \in (v_1, v_2)$ with $v_1 > v_0$, $\mathcal{Q}$ returns whatever tuples already returned in historic query answers that fall into range (v_1, v_2). It also sets $v_q = \min(v_q, v_1)$.

Otherwise, if q is of the form WHERE $A \in (v_0, v_2)$ with $v_2 > v_0$, then $\mathcal{Q}$ returns an overflowing answer with k tuples. These k tuples include those in the historic query answers that fall into (v_0, v_2). If more than k such tuples exist in the history, we choose an arbitrary size-k subset. If fewer than k such tuples exist, we

fill up the remaining slots with arbitrary values in range $((v_0 + v_q)/2, v_q)$[1]. We also set v_q to be $(v_0 + v_q)/2$.

There are two critical observations here. First is that for any query sequence $q_1, \ldots, q_h$ with $h \leq n/k$, we can always construct a database D of at most n tuples, such that the query answers generated by $\mathcal{Q}$ are consistent with what D produces. Specifically, D would simply be the union of all tuples returned. Note that our maintenance of v_q ensures the consistency.

The second critical observation is that no query re-ranking algorithm $\mathcal{R}$ can find the tuple with the smallest A without issuing at least n/k queries. The reason is simple: since $n/k - 1$ queries cannot reveal all n tuples, we can add a tuple t with $A = (v_0 + v_q)/2$ to the database, where v_q is its value after processing all $n/k - 1$ queries. One can see that the answers to all $n/k - 1$ queries can remain the same. As such, for any $n > 1$, there exists a database containing n tuples such that finding the top-ranked one for an attribute requires at least n/k queries, which according to [15] is sufficient for crawling the entire database in a 1D space. $\square$

3.2 1D-RERANK

Given the above result, we have to shift our attention to reducing the cost of finding t_{h+1} in an average-case scenario, e.g., when the tuples are more or less uniformly distributed on A_i (instead of forming a highly skewed distribution as constructed in the proof of Theorem 1). To this end, we start this subsection by considering a binary-search algorithm. After pointing out the deficiency of this algorithm when facing certain system ranking functions, we introduce our idea of on-the-fly indexing for the design of 1D-RERANK, our final algorithm for query reranking with a single-attribute user-specified ranking function.

3.2.1 1D-BINARY and its Problem

The binary search algorithm departs from 1D-BASELINE on the construction of q_2: Given a_1, instead of issuing q_2: WHERE $A_i \in (t_h[A_i], a_1[A_i])$ AND $Sel(q)$, we issue here

q_2' : WHERE $A_i \in (t_h[A_i], (a_1[A_i] + t_h[A_i])/2)$ AND $Sel(q)$.

This query has two possible outcomes: If it returns non-empty, we consider the returned tuple with minimum A_i, say a_2, and construct q_3' according to a_2. The other possible outcome is for q_2' to return empty. In this case, we issue q_2'': WHERE $A_i \in [(a_1[A_i] + t_h[A_i])/2, a_1[A_i])$ AND $Sel(q)$, which has to return non-empty as otherwise $t_{h+1} = a_1$. In either case, the *search space* (i.e., the range in which t_{h+1} must reside) is reduced by at least half. Algorithm 2, 1D-BINARY, depicts the pseudocode.

Algorithm 2 1D-BINARY

1: $t_{h+1} = \text{argmin}_{t[A_i]}\{t \in \text{History} \mid t[A_i] > t_h[A_i]\}$
2: **do**
3:     $q' = \text{WHERE } A_i \in (t_h[A_i], (t_{h+1}[A_i] + t_h[A_i])/2) \text{ AND } Sel(q)$
4:     $T = \text{Top-}k(q')$
5:     **if** T is underflow, **then**
6:         $q' = \text{WHERE } A_i \in [(t_{h+1}[A_i] + t_h[A_i])/2, t_{h+1}[A_i])$ AND $Sel(q)$
7:         $T = \text{Top-}k(q')$
8:     **if** T is not underflow, **then** $t_{h+1} = \text{argmin}_{t[A_i]}\{t \in T\}$
9: **while** T is overflow
10: **return** t_{h+1}

---

[1]Note that any factor here (besides 2) works too. So in general the range can be $((v_0 + v_q) \cdot \alpha, v_q)$ so long as $\alpha > 0$.

Query Cost Analysis: While the design of 1D-BINARY is simple, the query-cost analysis of it yields an interesting observation which motivates the indexing-based design of our final 1D-RERANK algorithm. Let

$$\epsilon_k = t_{h+k+1}[A_i] - t_{h+1}[A_i]. \qquad (1)$$

An important observation here is that the execution of 1D-BINARY must conclude when the search space is reduced to width smaller than ϵ_k, because no such range can cover $t_{h+1}[A_i]$ while matching more than k tuples. Thus, the worst-case query cost of 1D-BINARY is

$$O(\min(\log_2(|V(q, A_i)|/\epsilon_k), |R(q)|/k)), \qquad (2)$$

where $|V(q, A_i)|$ is the range of A_i among tuples satisfying q - i.e., $\max_{t \in R(q)} t[A_i] - \min_{t \in R(q)} t[A_i]$. Note that the second input to the $\min$ function in (2) is because every pair of queries issued by 1D-BINARY, i.e., q'_j and q''_j, must return at least k tuples never seen before that satisfies q.

The query-cost bound in (2) illustrates both the effectiveness and the potential problem of Algorithm 1D-BINARY. On one hand, one can see that 1D-BINARY performs well when the tuples matching q are uniformly distributed on A_i, because in this case the expected value of ϵ_k becomes $k \cdot |V(q, A_i)|/|R(q)|$, leading to a query cost of $O(\log_2(|R(q)|/k))$. On the other hand, 1D-BINARY still incurs a high query cost (as bad as $\Omega(|R(q)|/k)$, just as indicated by Theorem 1 when (1) the system ranking function is ill-conditioned, i.e., negatively correlated with A_i, and (2) Within $R(q)$ there are *densely clustered* tuples with extremely close values on A_i, leading to a small ϵ_k. Unfortunately, once the two conditions are met, the high query cost 1D-BINARY is likely to be incurred again and again for different user queries q, leading to an expensive reranking service. It is this observation which motivates our index-based reranking idea discussed next.

3.2.2 Algorithm 1D-RERANK: On-The-Fly Indexing

Oracle-based Design: According to the above observation, densely clustered tuples cause a high query cost of 1D-BINARY. To address the issue, we start by considering an ideal scenario where there exists an oracle which identifies these "dense regions" and reveals the tuple with minimum A_i in these regions without costing us any query. Of course, no such oracle exists in practice. Nevertheless, what we shall do here is to analyze the query cost of 1D-BINARY given such an oracle, and then show how this oracle can be "simulated" with a low-cost on-the-fly indexing technique.

Specifically, for any given region $[x, y] \in V(A_i)$, we call it a *dense region* if and only if it covers at least s tuples *and* $y - x < |V(A_i)| \cdot (s/n)/c$, where c and s are parameters. In other words, the density of tuples in $[x, y]$ is more than c times higher than the uniform distribution (which yields an expected value of $E(y-x) = |V(A_i)| \cdot (s/n)$). The setting of c and s is a subtle issue which we specifically address at the end of this subsection. Given the definition of dense region, the oracle functions as follows: Upon given a user query q, an attribute A_i, and a range $[x, y] \subseteq V(A_i)$ as input, the oracle either returns empty if $[x, y]$ is not dense, or a tuple t which (1) satisfies q, (2) has $A_i \in [x, y]$, and (3) features the smallest A_i among all tuples satisfying (1) and (2).

With the existence of this oracle, we introduce a small yet critical revision to 1D-BINARY, by *terminating* binary search whenever the width of the search space becomes narrower than the threshold for dense region, i.e., $\epsilon_k < |V(A_i)| \cdot (s/n)/c$. Then, we call the oracle with the remaining search space as input. Note that doing so may lead to two possible returns from the oracle:

One is when the region is indeed dense. In this case, the oracle will directly return us t_{h+1} with zero cost. The other possible

outcome is an empty return, indicating that the region is not really dense, instead containing more than k (otherwise 1D-BINARY would have already terminated) but fewer than s tuples. Note that this is not a bad outcome either, because it means that by following the baseline technique (1D-BASELINE) on the remaining search space, we can always find t_{h+1} within $O(s/k)$ tuples.

Algorithm 3 depicts the pseudocode of 1D-RERANK, the revised algorithm. The following theorem shows its query cost, which follows directly from the above discussions.

Algorithm 3 1D-RERANK

1: $t_{h+1} = \operatorname{argmin}_{t[A_i]}\{t \in \text{History} \mid t[A_i] > t_h[A_i]\}$
2: **while** $(t_{h+1}[A_i] - t_h[A_i]) < |V(A_i)| \cdot (s/n)/c$
3:     $q\prime = \text{WHERE } A_i \in (t_h[A_i], (t_{h+1}[A_i] + t_h[A_i])/2) \text{ AND } Sel(q)$
4:     $T = \text{Top-}k(q\prime)$
5:     **if** T is underflow
6:       $q\prime = \text{WHERE } A_i \in [(t_{h+1}[A_i] + t_h[A_i])/2, t_{h+1}[A_i]) \text{ AND } Sel(q)$
7:       $T = \text{Top-}k(q\prime)$
8:     **if** T is not underflow
9:       $t_{h+1} = \operatorname{argmin}_{t[A_i]}\{t \in T\}$
10:     **if** T is valid **break**
11: **if** T is valid
12:     look up t_{h+1} at ORACLE(A_i,$(t_h[A_i], t_{h+1}[A_i])$,q)
13: **return** t_{h+1}

THEOREM 2. *The query cost of 1D-RERANK, with the presence of the oracle, is $O(\log(c \cdot n/s) + s/k)$.*

PROOF. The query cost of of 1D-RERANK, with the presence of the oracle, is the summation of (i) the query cost of following 1D-BINARY until the search space becomes narrower than the dense region threshold, which is $O(\log_2(|V(q, A_i)|/\epsilon_k))$, and (ii) the query cost of discovering t_{h+1} in the remaining region, using the oracle. Since $\epsilon_k < |V(A_i)| \cdot (s/n)/c$, the first term is $O(\log(c \cdot n/s))$. As discussed previously, if the oracle does not include the remaining region, the region is not dense and contains fewer than s tuples. Thus, following 1D-BASELNE in the remaining region requires at most s/k queries. □

Note that the query cost indicated by the theorem is very small. For example, when $c = n$ and $s = k \cdot \log n$, the query cost is $O(\log n)$, substantially smaller than that of 1D-BINARY. Of course, the oracle does not exist in any real system. Thus, our goal next is to simulate this oracle with an efficient on-the-fly indexing technique.

On-The-Fly Indexing: Our idea for simulating the oracle is simple: once 1D-RERANK decides to call the oracle with a range (x, y), we invoke the 1D-BASELINE algorithm on SELECT * FROM D WHERE $A_i \in (x, y)$ to find the tuple t with smallest A_i in the range. If t satisfies the user query q being processed, then we can stop and output t. Otherwise, we call 1D-BASELINE on WHERE $A_i \in (t[A_i], y)$ to find the No. 2 tuple, and repeat this process until finding one that satisfies q. All tuples discovered during the process are then added into the "dense index" that is maintained throughout the processing of all user queries.

Algorithm 4 depicts the on-the-fly index building process. Note that the index we maintain is essentially a set of 3-tuples

$$\langle A_i, (x, y), D(A_i, x, y)) \rangle, \qquad (3)$$

where A_i is an attribute, (x, y) is a range in $V(A_i)$ (non-overlapping with other indexed ranges of A_i), and $D(A_i, x, y)$ contains all (top-ranked) tuples we have discovered that have $A_i \in (x, y)$.

Algorithm 4 ORACLE

1: **if** ORACLE(A_i,x,y) exists
2:     **return** $\arg\min_{t[A_i]}\{t \in D(A_i,x,y))| \ t$ matches $Sel(q)\}$
3:   t=1D-BASELINE(WHERE $A_i \in (x,y)$)
4:   add t to $D(A_i,x,y)$
5: **while** t does not satisfy $Sel(q)$
6:     t=1D-BASELINE(WHERE $A_i \in (t[A_i],y)$)
7:     add t to $D(A_i,x,y)$
8: **return** t

Note that this simulation does differ a bit from the ideal oracle. Specifically, it does not really determine if the region is dense or not. Even if the region is not dense, this simulated oracle still outputs the correct tuple. What we would like to note, however, is that this difference has no implication whatsoever on the query cost of 1D-RERANK. Specifically, what happens here is simply that the on-the-fly indexing process pre-issues the queries 1D-RERANK is supposed to issue when the oracle returns empty. The overall query cost remains exactly the same.

Another noteworthy design in on-the-fly indexing is the call of 1D-BASELINE on SELECT * FROM D WHERE $A_i \in (x,y)$, a query that does not "inherit" the selection conditions in the user query q being processed. This might appear like a waste as 1D-BASELINE could issue fewer queries with a narrower input query. Nonetheless, we note that rationale here is that a dense region might be covered by multiple user queries repeatedly. By keeping the index construction generic to all user queries, we reduce the amortized cost of indexing as the dense index can make future reranking processes more efficient.

Parameter Settings: To properly set the two parameters for dense index, c and s, we need to consider not only the query cost derived in Theorem 2, but also the cost for building the index, which is considered in the following theorem:

THEOREM 3. *The total query cost incurred by on-the-fly indexing (for processing all user queries) is at most*

$$\sum_{h=1}^{n-s-1} c(h) \qquad (4)$$

where $c(h) = 1$ if there exists $j \in [h-s, h]$, such that

$$t(*, A_i, j + s + 1)[A_i] - t(*, A_i, j)[A_i] < \frac{s \cdot |V(A_i)|}{c \cdot n}, \quad (5)$$

and 0 otherwise. Here $t(*, A_i, j)$ refers to the j-th ranked tuple according to A_i in the entire database.

PROOF. The discovery of every tuple in the dense region takes at most the amortized cost of one query. That is because 1D-BASELINE assures the discovery on k unseen tuples by every non-underflowing query, i.e. every tuple in the dense region is discovered by one and only one query. Thus the query cost is at most equal to the number of tuples in the dense regions. Each tuple t is in the dense region with regard to the dimension A_i, if, sorting the tuples on A_i, we can construct a window containing t, with size less than the dense region threshold, that has at least s tuples. Suppose t is ranked h-th based on A_i. Equation 5 checks the existence of such a window around it. The total cost thus, is at most the number of the tuples for which this equation is true. This is reflected in Equation 4. □

One can see from the above theorem and Theorem 2 how c and s impacts the query cost: the larger c is, the fewer dense regions there will be, leading to a lower indexing cost. On the other hand, the per-query reranking cost increases at the log scale with c. Similarly,

the larger s is, the fewer dense regions there will be (because a larger s reduces the variance of tuple density), while the per-query reranking cost increases linearly with s. Given the different rate of increase for the per-query reranking cost with c and s, we should set c to be a larger value to leverage its log-scale effect, while keep s small to maintain an efficient reranking process.

Specifically, we set $c = n$ and $s = k \cdot \log n$. One can see that the per-query reranking cost of 1D-RERANK in this case is $O(\log n)$. While the indexing cost depends on the specific data distribution (after all, we are bounded by Theorem 1 in terms of worst-case performance), the large value of $c = n$ makes it extremely unlikely for the indexing cost to be high. In particular, note that even if the density surrounding each tuple follows a heavy-tailed scale-free distribution, the setting of $c = n$ still makes the number of dense regions, therefore the query cost for indexing, a constant.We shall verify this intuition and perform a comprehensive test of different parameter settings in the experimental evaluations.

Now let us use 1D-RERANK on a mock example specified in Example 1, in order to find the Top-1 tuple with minimum delay.

EXAMPLE 1. *Consider a flight-search website, with two attributes: Price (A_1) and Delay (A_2). Suppose $D = \{t_1[50, 1000], t_2[-51, 990], \ldots, t_{50}[99, 510], t_{51}[200, 200], t_{52}[510, 99], t_{53}[520, 98], \ldots, t_{100}[990, 51]\}$, while the Top-1 interface of the database ranks the flights based on price ascendingly.*

Let $s = 6$ and $c = 20$. Starting from SELECT *, 1D-RERANK issues the following set of queries (in q_i(condition):result format): $\{q_0() : t_1, q_1(A_2 < 500) : t_{51}, q_2(A_2 < 100) : t_{52}, q_3(A_2 < 50) : \emptyset, q_4(50 \leq A_2 < 99) : t_{53}, q_5(50 \leq A_2 < 74) : t_{78}, q_6(50 \leq A_2 < 61.5) : t_{90}, q_7(50 \leq A_2 < 55.5) : t_{96}$, (reaching the dense region threshold) $q_8(50 \leq A_2 < 52.5) : t_{99}, q_9(50 \leq A_2 < 52) : t_{100}, q_{10}(50 \leq A_2 < 51) : \emptyset\}$, and returns t_{100}.

4. MD-RERANK

In this section, we consider the generic query reranking problem, i.e., over any monotonic user-specified ranking function. We start by pointing out the problem of implementing a classic top-k query processing algorithm such as TA [9] by calling 1D-RERANK as a subroutine. The problem illustrates the necessity of properly leveraging the conjunctive queries supported by the search interface of the underlying database. To do so, we start with the design of MD-BASELINE, a baseline technique similar to 1D-BASELINE. Despite of the similarity, we shall point out a key difference between two cases: MD-BASELINE requires many more queries because of the more complex shape of what we refer to as a tuple's "rank-contour" - i.e., the subspace (e.g., a line in 2D space) containing all possible tuples that have the same user-defined ranking score as a given tuple t. To reduce this high query cost, we propose Algorithm MD-BINARY which features two main ideas, *direct domination detection* and *virtual tuple pruning*. Finally, we integrate the dense-region indexing idea with MD-BINARY to produce our final MD-RERANK algorithm.

4.1 Problem with TA over 1D-RERANK

A seemingly simple solution to solve the generic query reranking problem is to directly apply a classic top-k query processing algorithm, e.g., the threshold (TA) algorithm [9], over the Get-Next primitive offered by 1D-RERANK. While we refer readers to [9] for the detailed design of TA, it is easy to see that 1D-RERANK offers all the data structure required by TA, i.e., a sorted access to each attribute. Note that the random access requirement does not apply here because, as discussed in the preliminary section, the search interface returns all attribute values of a tuple without the
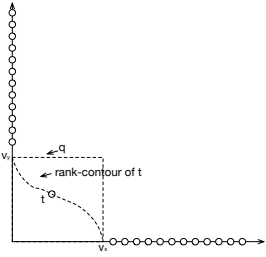
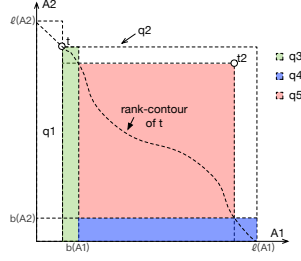**Figure 1: Illustration of problem with TA over 1D-RERANK**

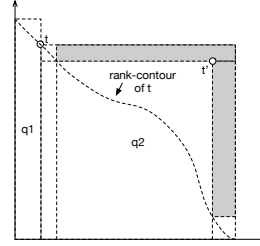**Figure 2: Example of MD-BASELINE**

Figure 3: Illustration of problem with MD-Baseline
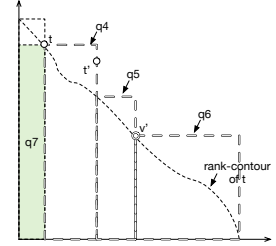

Figure 4: Design of MD-Binary: Example 2

need for accessing each attribute separately. Since TA supports all monotonic ranking functions, this simple combination solves the generic query reranking problem defined in § 2.

While simple, this solution suffers from a major efficiency problem, mainly because of not leveraging the full power provided by client-server databases. By exclusively calling 1D-RERANK as a subroutine, this solution focuses on just one attribute at a time and does not issue any multi-predicate (conjunctive) queries supported by the underlying database (unless such predicates are copied from the user query). The example in Figure 1 illustrates the problem: In the example, there is a large number of tuples with extreme values on both attributes (i.e., tuples on the x- and y-axis). Since this TA-based solution focuses on one attribute at a time, these extreme-value tuples have to be enumerated first *even when* the system ranking function completely aligns (e.g., equals) the user-desired ranking function.

On the other hand, one can observe from the figure the power bestowed by the ability to issue multi-predicate conjunctive queries. As an example, consider the case where the system ranking function is well-conditioned and returns t as the result for SELECT * FROM D. Given t, we can compute its rank-contour, i.e., the line/curve that passes through all 2D points with user-defined score equal to $\mathcal{S}(t)$, i.e., the score of t. The curve in the figure depicts an example. Given the rank-contour, we can issue the smallest 2D query encompassing the contour, e.g., q in Figure 1, and immediately conclude that t is the No. 1 tuple when q returns t and nothing else (assuming $k > 1$). This represents a significant saving from the query cost of implementing TA over 1D-RERANK.

4.2 MD-Baseline

4.2.1 Discovery of Top-1

To leverage the power of multi-predicate queries, we start with developing a baseline algorithm similar to 1D-BASELINE. The algorithm starts with discovering the top-1 tuple t according to an arbitrary attribute, say A_1. Then, we compute the rank-contour of t (according to the user ranking function, of course), specifically the values where t's rank-contour intersects with each dimension, i.e.,

$$\ell(A_i) = \max\{v \in V(A_i)|\mathcal{S}(t) \leq \mathcal{S}(0, \ldots, 0, v, 0, \ldots, 0)\}. \quad (6)$$

Figure 2 depicts an example of $\ell(A_i)$ for the two dimensions, computed according to t.

We now issue m queries of the form

$q_1 : A_1 < t[A_1] \ \& \ A_2 < \ell(A_2) \ \& \ \cdots \ \& \ A_m < \ell(A_m)$

$q_2 : A_1 \in [t[A_1], \ell(A_1)) \ \& \ A_2 < t[A_2] \ \& \ A_3 < \ell(A_3) \ \& \ \cdots$
$\quad \& \ A_m < \ell(A_m)$

$q_m : A_1 \in [t[A_1], \ell(A_1)) \ \& \ \cdots \ \& \ A_{m-1} \in [t[A_{m-1}],$
$\quad \ell(A_{m-1})) \ \& \ A_m < t[A_m] \quad (7)$

Again, Figure 2 shows an example of q_1 and q_2 for the 2D space.

One can see that the union of these m (mutually exclusive) queries covers in its entirety the region "underneath" the rank-contour of t. Thus, if none of them overflows, we can safely conclude that the No. 1 tuple must be either t or one of the tuples returned by one of the m queries. If at least one query overflows and returns t' with score $\mathcal{S}(t') < \mathcal{S}(t)$, i.e., t' that ranks higher than t, we restart the entire process with $t = t'$.

Otherwise, for each query q_i that overflows, we "partition" it further into $m + 1$ queries. Let t_i be the tuple returned by q_i. We compute for each attribute A_j a value $b(A_j)$ such that

$$b(A_j) = \min\{v \in V(A_j)|\mathcal{S}(t) \leq$$
$$\mathcal{S}(t_i[A_1], \ldots, t_i[A_{j-1}], b(A_j), t_i[A_{j+1}], \ldots, t_i[A_m])\}. \quad (8)$$

Intuitively, $b(A_j)$ can be understood as follows: In order for a tuple t' to outrank t, the highest-ranked tuple discovered so far, it must either "outperform" $b(A_j)$ on at least one attribute, i.e., $\exists A_j$ with $t'[A_j] < b(A_j)$, or it must dominate t_i. Examples of $b(A_1)$ and $b(A_2)$ are shown in Figure 2.

Note that, while any monotonic (user-defined) ranking function yields a unique solution for $b(A_j)$, the complexity of computing it can vary significantly depending upon the design of the ranking function. Nonetheless, recall from § 2 that our main efficiency concern is on the *query cost* of the reranking process rather than the computational cost for solving $b(A_j)$ locally (which does not incur any additional query to the underlying database). Furthermore, the most extensively studied ranking function in the literature, a linear combination of multiple attributes, features a constant-time solver for $b(A_j)$.

Given $b(A_j)$, we are now ready to construct the $m + 1$ queries we issue. The first m queries $q_{i1}, \ldots, q_{im}$ cover those tuples outperforming $b(A_1), \ldots, b(A_m)$ on $A_1, \ldots, A_m$, respectively; while the last one covers those tuples dominating t_i. Specifically, q_{ij} $(j \in [1, m])$ is the AND of q_i and

$$(A_1 \geq b(A_1)) \text{ AND } \cdots \text{ AND } (A_{j-1} \geq b(A_{j-1}))$$
$$\text{AND } (A_j < b(A_j)) \quad (9)$$

The last query is the AND of q_i and $A_1 \leq t_i[A_1]$ AND $\cdots$ AND $A_m \leq t_i[A_m]$, i.e., covering the space dominating t_i.

Once again, at anytime during the process if a query returns t' with $\mathcal{S}(t') < \mathcal{S}(t)$, we restart the entire process with $t = t'$. Otherwise, for each query that overflows, we "partition" it into $m + 1$ queries as described above.

In terms of query cost, recall from § 2 our idea of leveraging the query history by checking if any previously discovered tuples match the query we are about to issue. Given the idea, each tuple will be retrieved at most once by MD-BASELINE. Since each tuple we discover triggers at most $m + 1$ queries which are mutually exclusive with each other, one can see that the worst-case query cost of MD-BASELINE for discovering the top-1 tuple is $O(m \cdot n)$.

4.2.2   Discovery of Top-k

We now discuss how to discover the top-k ($k > 1$) tuples satisfying a given query. To start, consider the discovery of No. 2 tuple after finding the top-1 tuple t_1. What we can do is to pick an arbitrary attribute, say A_1, and partition the search space into two parts: $A_1 < t_1[A_1]$ and $A_1 > t_1[A_1]$. Then, we launch the top-1 discovery algorithm on each subspace. Note that during the discovery, we can reuse the historic query answers - e.g., by starting from the tuple(s) we have also retrieved in each subspace that have the smallest $\mathcal{S}(\cdot)$. One can see that one of the two discovered top-1s must be the actual No. 2 tuple t_2 of the entire space.

Once t_2 is discovered, in order to discover the No. 3 tuple, we only need to further split the subspace from which we just discovered t_2 (into two parts). For example, if we discovered t_2 from $A_1 > t_1[A_1]$, then we can split it again into $A_1 \in (t_1[A_1], t_2[A_1])$ and $A_2 > t_2[A_1]$. One can see that the No. 3 tuple must be either the top-1 of one of the two parts or the top-1 of $A_1 < t_1[A_1]$, which we have already discovered. As such, the discovery of each tuple in top-k, say No. h, requires launching the top-1 discovery algorithm *exactly twice*, over the two newly split subspaces of the subspace from which the No. $h-1$ tuple was discovered. Thus, the worst-case query cost for MD-BASELINE to discover all top-k tuples is $O(m \cdot n \cdot k)$.

4.3   MD-Binary

4.3.1   Problem of MD-Baseline

A main problem of MD-Baseline is its poor performance when the system ranking function is *negatively correlated* to the user-desired ranking function. To understand why, consider how MD-Baseline compared with the 1D-Baseline algorithm discussed in § 3. Both algorithms are iterative in nature; and the objectives for each iteration are almost identical in both algorithms: once a tuple t is discovered, find another tuple t' that outranks it according to the input ranking function. The difference, however, is that while it is easy to construct in 1D-Baseline a query that covers only those tuples which outranks t (for the attribute under consideration), doing so in the MD case is impossible.

The reason for this difference is straightforward: observe from Figure 3 that, when there are more than one, say two, attributes, the subspace of tuples outranking t is roughly "triangular" in shape. On the other hand, only "rectangular" queries are supported by the database. This forces us to issue at least m queries to "cover" the subspace outranking t (without covering, and returning, t itself).

The problem for this "coverage" strategy in MD-Baseline, however, is that the rectangular queries it issues may match many tuples that indeed rank lower (i.e., have larger $\mathcal{S}(\cdot)$) than t according to the desired ranking function. For example, half of the space covered by q_2 in Figure 3 is occupied by tuples that rank lower than t. This means that, when the system ranking function is negatively correlated with our desired one, queries like q_2 in Figure 3 are most likely going to return tuples that rank lower than t. This outcome has two important ramifications on the efficiency of MD-Baseline: First, it significantly slows down the process of iteratively finding a tuple that outranks the previous one. Second, within each iteration, it slows down the pruning of the search space. For example, observe from Figure 3 that, after q_2 returns t', the pruning effect on the space covered by q_2 is minimal, i.e., only the dark subspace on the top-right corner of q_2.

4.3.2   Design of MD-Binary

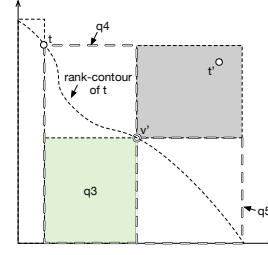We propose two ideas in MD-Binary to address the two ramifications of MD-Baseline, respectively:

Figure 5: Design of MD-Binary: Example 1

Direct Domination Detection: The intuition of this idea can be stated as follows: When a query such as q_2 returns a tuple t' that ranks lower than t, we attempt to "test" whether this is indeed caused by the absence of higher-ranked tuples in q_2, or by the ill-conditioned nature of the system ranking function. As discussed above, there is no way to efficiently cover the subspace of tuples outranking t. Thus, what we do here is to find the single query which (1) is a subquery of q_2, (2) only covers the subspace outranking t, and (3) has the maximum volume among all queries that satisfy (1) and (2).

For example, when q_2 in Figure 3 returns t', we issue q_3 (marked in green) in Figure 5 which covers roughly half of the "triangular" subspace underneath the rank-contour of t in q_2. As another example, if q_1 in Figure 2 returns a tuple with lower rank than t, then we the max-volume tuple would be q_7 in Figure 4, which covers almost all of the subspace outranking t in q_1. One can see from these examples that, if the returning of t' is caused by the ill-conditioned system ranking function while there are abundant tuples outranking t, then q_3 and/or q_7 are likely to return such a tuple and successfully push MD-Binary to the next iteration. If, on the other hand, q_3 returns empty, we use the next idea to further partition q_2, in order to determine whether there is any tuple in it that outranks t.

Virtual Tuple Pruning: We now address the second problem of MD-Baseline, i.e., the lack of pruning power when the system ranking function is negatively correlated with the desired one. To this end, our idea is to prune the search space according to *not* the returned tuple, but a *virtual tuple* created for the purpose of minimizing the pruned subspace. Figure 5 illustrates an example: Instead of partitioning q_2 with t' like in Figure 3 which results in minimal pruning, we "create" a virtual tuple v' which maximizes the reduction of search space as marked in gray in Figure 5.

Figure 5 represents one possible outcome of virtual tuple pruning, when v' happens to dominate the tuple t' returned by q_2. The other possible outcome is depicted in Figure 4, where v' does not dominate t'. In this case, if we still split q_2 as in Figure 5, then one of the subspace (i.e., $x \in (t[x], v'[x])$ AND $y < t[y]$) would return t', making the query answer useless. As such, we split q_2 into three pieces in this scenario, as shown in Figure 4.

The more general design of virtual tuple pruning for an m-D database is shown in Algorithm 5. The algorithm also depicts the direct domination detection idea. Note from the algorithm that, depending on the values of t' and v' on the m attributes, the number of split subspaces can range from m, when v' dominates t', to $2m-1$, when t' dominates v' on all but one attribute.

One can see from the design that virtual tuple pruning does not affect the correctness of the algorithm: so long as $\mathcal{S}(v') \geq \mathcal{S}(t)$, the union of the split subspaces still cover q_2. On the other hand, the benefit of the idea can be readily observed from Figure 5: instead of having only a small reduction of the search space like in Figure 3, now we can prune half of the space in q_2 that rank below t (in this 2D case, of course). The experimental results in § 6 demonstrate the effectiveness of virtual tuple pruning.

Algorithm 5 MD-BINARY

1: apply 1D-RERANK on A_1 to t and set threshold=$s(t)$
2: add the queries in Equation 7 to the empty queue
3: **while** queue is not empty
4:    $q\prime$=queue.delete
5:    $T = $ Top-$k(q\prime)$;$t = \text{argmin}_{s(t)}\{t \in T\}$
6:    **if** $s(t) <$threshold
7:      threshold=$s(t)$; **goto** Line 2
8:    **if** T is valid: **continue**
9:    $v' = \text{argmax}_{vol(v)}\{v \in \text{contour}(t)\}$
10:   $T = $ Top-$k(\forall A \in \mathcal{A}, A \leq v'[A])$
11:   **if** T is not underflow
12:     t=argmin$_{s(t)}\{t \in T\}$; threshold=$s(t)$; **goto** Line 2
13:   **for each** $A_i \in \mathcal{A}$
14:     **if** $t[A_i] \geq v'[A_i]$ add the following query to the queue

$$q_1: \ q\prime \text{ AND } A_i < v'[A_i] \text{ AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$

15:     **else** add the following queries to the queue

$$q_1: \ q\prime \text{ AND } A_i < t[A_i] \text{ AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$
$$q_2: \ q\prime \text{ AND } A_i < v'[A_i] \text{ AND } A_{i+1} < b_t(A_{i+1})$$
$$\text{AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$

16: **return** t

4.4 MD-RERANK

Just like the 1D case, the query cost of MD-Binary may increase significantly when there is a dense cluster of tuples right above the rank-contour of the top-1 tuple. In this case, the split in MD-Binary may have to continue for a large number of times before all tuples in the cluster are excluded from the search space. Once again, our solution to this problem is index-based reranking. Like in the 1D case, we proactively *record as an index* densely located tuples once we encounter them, so that we do not need to incur a high query cost every time a query q triggers visits to the same dense region.

More specifically, MD-RERANK follows MD-Binary until a remaining search space (1) is covered by an already crawled region in the index; or (2) has volume smaller than $|V| \cdot (s/n)/c$, where $|V|$ is the volume of the entire data space, and s and c are the same as in 1D. In the earlier case, since the search space has been crawled already, we can directly reuse the crawled tuples. In the latter case, we follow the same procedure as in 1D-RERANK, i.e., we crawl the space and, if it indeed turns out to be dense (by containing at least s tuples), we include the crawled tuples into the index. Algorithm 6 depicts the pseduocode of MD-RERANK.

Algorithm 6 MD-RERANK

1: follow MD-BINARY
2: during the process for each query $q\prime$:
3:   **if** $V(q\prime) < |V| \cdot (s/n)/c$
4:     $q\prime = $ remove $Sel(q)\}$ from $q\prime$
5:     **if** ORACLE($q\prime$) exists
6:       **return** argmin$_{s(t)}\{t \in D(q\prime))|\ t \text{ matches } Sel(q)\}$
7:     t=MD-BASELINE($q\prime$); add t to temp
8:     **while** t does not satisfy $Sel(q)$
9:       $t_1 = $ MD-BASELINE($q\prime$ AND $A_1 < t[A_1]$)
10:      $t_2 = $ MD-BASELINE($q\prime$ AND $A_1 > t[A_1]$)
11:      t=min(t_1 , t_2); add t_1 and t_2 to temp
12:     add temp to $D(q\prime)$

Now, considering Example 1, let us apply MD-RERANK in order to find the Top-1 tuple with minimum *Delay+Price*. The algorithm applies 1D-RERANK to find the Top-1 on A_1 ($q_0()$: $t_1, q_1(A_1 < 50) : \emptyset$), then issues the query $q_2(A_1 \in [50, 1050)$

AND $A_2 < 1000) : t_2$. Next, constructing the virtual point $v'[525, 525]$, it issues the query $q_3(A_1 \leq 525 \text{ AND } A_2 \leq 525)$: t_{51}. Having found a better result, it restarts the process with queries $q_4(A_1 < 200 \text{ AND } A_2 < 400) : \emptyset$ and $q_5(A_1 \in [200, 400) \text{ AND } A_2 < 400) : \emptyset$. It finally returns t_{51} as Top-1.

5. DISCUSSIONS

General Positioning Assumption: In previous discussions, we made the general positioning assumption, for the simplicity of discussions. We now consider the removal of this assumption. Note that the removal of this assumption for MD-RERANK is extremely simple: the only tuple(s) that can be missed by MD-RERANK are those that have the exact same value on *every single attribute*. Thus, the only post-processing step required for removing the assumption is to form a fully specified query according to No. h tuple just discovered. If more than one, say i, tuples are returned, they become the No. h to No. $(h + i - 1)$ top-ranked tuples. Removing the assumption for 1D-RERANK is slightly more complex. For example, if we are running it over attribute A_1, the removal of the general positioning assumption means query SELECT * FROM D WHERE $A_1 = t[A_1]$ might overflow. In this case, our solution is to call the crawling algorithm [15] to discover, one at a time, tuples satisfying $A_1 = t[A_1]$, as all of these tuples have the same rank for the purpose of 1D-RERANK.

Multiple/Known System Ranking Functions: Another interesting issue arising in practice is when the client-server database offers more than one ranking functions, often times allowing ranking over a specific attribute. For example, Amazon.com offers not only a proprietary rank by "popularity", the design of which is unknown, but also ranking by price, which is an attribute usually involved in the user-specified ranking function. An interesting implication of such a "public" ranking function is that it might boost the performance of the TA-1D algorithm discussed in the beginning of § 4. Specifically, since now TA can simply use the public ranking function on the attribute instead of calling 1D-RERANK, it may have a even lower query cost than MD-RERANK when the user-desired ranking function aligns well with the system one.

Point Predicates: In this paper, we focused on cases where attributes involved in the ranking function are numeric attributes that support range queries. While this is often the case in practice (as evidenced in real-world websites such as the aforementioned Blue Nile where all attributes such as price, carat, clarity, etc., are available as range predicates), there are also cases where a ranking attribute with only a small number of domain values can only be specified as a point predicate (i.e., of the form $A_i = v$) in the database search interface. For 1D-RERANK, this is often a blessing because it simplifies the task to querying the attribute values in the preference order (plus the crawling-based provision as in the discussion for the general positioning assumption). On the other hand, it makes MD-RERANK much more costly, because now a conjunctive query covers a much smaller space than the range case. Thus, an intuition here is to prefer the TA-1D algorithm over MD-RERANK when a large number of attributes are searchable as point predicates only. Due to space limitations, we leave a comprehensive study of this issue to future work.

6. EXPERIMENTAL EVALUATION

6.1 Experimental Setup

In this section, we present our experimental results over a number of several real-world datasets, offline and online. We started

with the offline case by testing over a real-world dataset we have already collected. Specifically, we constructed a top-k web search interface over it, and then executed our algorithms through the interface. This offline setting enabled us to not only verify the correctness of our algorithms, but also investigate how the performance of query reranking changes with various factors such as the database size, the system ranking function, settings of the system search interface, etc. We followed the offline tests with online, live, experiments over two real-world web databases, including the largest online diamond retailer and a popular auto search website. In all these experiments, we applied the extensions described in § 5 to resolve the general positioning assumption which may not hold in practice.

Offline Dataset: We used the flight on-time dataset published by the US Department of Transportation (DOT)[2]. A wide range of third-party websites use this dataset to identify on-time performance of flights, routes, airports, airlines, etc. It consists of 457,013 flight records of 14 US carriers during the month of May 2015. It has 28 attributes, out of which we selected the following 8 attributes for ranking: *Dep-Delay, Taxi-Out, Taxi-In, Arr-Delay-New, CRS-Elapsed-Time, Actual-Elapsed-Time, Air-Time,* and *Distance*. The domain sizes are 1988, 180, 180, 1971, 718, 724, 676, and 5000, respectively. For the purpose of the experiments, we considered two system ranking functions: *0.3 AIR-TIME + TAXI-IN* (SR1) and *-0.1 DISTANCE - DEP-DELAY* (SR2). In general, SR1 has a positive correlation with the user-specified ranking functions we tested, while SR2 has a negative one. We set SR1 as the default ranking function in the experiments. The value of k offered by the database is set to 10 by default.

Online Experiments: We conducted live experiments over two real-world web-sites: *Blue Nile* (BN) and *Yahoo! Autos* (YA).
Blue Nile[3] is the largest diamonds online retailer in the world. At the time of our experiments, its catalog had 117,641 diamonds. We considered *Carat, Depth, LengthWidthRatio, Price,* and *Table* as the ranking attributes, and *Clarity, Color, Cut, Fluorescence, Polish, Shape,* and *Symmetry* for filtering. The domains for the ranking attributes are [0.23,22.74], [0.45,0.86], [0.49,0.89], [\$220,\$4506938] and [0.75,2.75], respectively. BN allows multiple ranking functions - ordering based on each attributes individually as well as by the derived attribute *price-per-carat*.
Yahoo! Autos is a popular website for buying used cars[4]. We considered the 13,169 cars listed for sale within 30 miles of New York city. We treated *Price, Milage* and *Year* as the ranking attributes, and *BodyStyle, DriveType, Transmission, Name* and *Model* as the filtering attributes. The cars had a price range between \$0 and \$50,000, mileage between 0 and 300,000, and were manufactured between 1993 and 2016. The default ranking function is "distance from a predefined location" (which is not monotonic). Additionally, it supports ranking by each of the numerical attributes individually.

Performance Measures: As explained in § 2, our algorithms always return the precise query answer. After verifying the correctness in all offline experiments, we turn our attention to the key performance measure, efficiency, which is measured by the number of queries issued to the web database.

6.2 1D Experiments

Constructing Workload of User Preference Queries: We tested a diverse set of user-specified queries of the form SELECT * FROM

---

[2] http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time
[3] http://www.bluenile.com/diamond-search
[4] https://autos.yahoo.com/used-cars/

D WHERE $Sel(q)$ ORDER BY A_i. Specifically, we randomly selected different subsets of filtering attributes for the WHERE clause, while choosing the (1D) ranking attribute uniformly at random. This approach has a number of appealing properties. First, it covers diverse cases that include ideal, worst-case and typical scenarios. Second, since 1D-RERANK uses on-the-fly indexing to amortize the cost between different user-issued queries, our diverse query workload simulates a real-world scenario where the service is used by multiple users. For each experimental configuration, we execute each of the queries and report the average query cost. Specifically, for the DOT dataset, we constructed 32 queries of which 25% do not have any filtering condition. For BN, we constructed a set of 20 queries, of which 4 have no filtering conditions, while these values are 15 and 2 for YA, respectively.

6.2.1 Experiments over the Real-world Dataset

Impact of Database Size and System Ranking Function: We started by testing the impact of database size on our algorithms for the two system ranking functions SR1 and SR2. To test databases of varying sizes, we drew 10 simple random samples of a given size from the DOT dataset, and measured the average query cost for the entire workload over these 10 small databases. Figures 6 and 7 show the average query cost for retrieving the top-1 tuple over SR1 and SR2, respectively. The database size, as expected, has negligible impact on the query cost. As discussed in § 3.2.2, by applying the on the fly indexing in order to simulate the dense regions oracle (while considering proper parameter settings), 1D-RERANK resolves the problem of 1D-BINARY and can work as good as $O(\log n)$. This is reflected in the figures where 1D-RERANK outperformed both 1D-BASELINE and 1D-BINARY significantly. One can also note that the change in system ranking function has a major impact on the performance comparison between 1D-BASELINE and 1D-BINARY. This is, as pointed in §3.1.2, due to the fact that when the system ranking function is positively correlated with the user-specified queries, the top result based on system function is close to the top user-queried tuple, and therefore, 1D-BASELINE only needs to explore a small area with few number of queries, while 1D-BINARY issues more queries on very narrow empty regions. On the other hand, when the system ranking function is negatively correlated with the user-specified queries, 1D-BASELINE need to explore a large region and slowly move toward the top user-queried tuple, while 1D-BINARY rapidly shrinks the search region by cutting the search space by half.

Impact of Value of k: Figure 8 shows the average (accumulative) query cost for retrieving top-1 to top-10 tuples when the system k varies from 1 to 10. There are two key observations from the figure: First, our query cost increases (about) linearly with the number of desired top answers, demonstrating its scalability to a large desired answer size. Second, the query cost, as expected, decreases when the system offers a larger k.

Impact of 1D-RERANK parameters s and c: Recall from § 2 that the performance of 1D-RERANK can be parameterized by s and c. We conducted two experiments to empirically verify the impact. In the first experiment, we fixed the value of s to n and varied c between 10 and n^2. In the other one, we fixed the value of c to $k \log_2 n$ and varied the value of s from 10 to n^2. Figure 9 shows the average query cost for both settings. As our theoretical results suggest, setting $c = k \log_2 n$ and $s = n$ resulted in the (almost) optimal performance. One can see that further reducing c or increasing s does not have much affect on query cost, yet significantly increases the index size.

Impact of Query Order on 1D-RERANK: Recall that 1D-RERANK constructs the index on the fly. As such, when queries are issued in different order, the index being maintained may differ. To
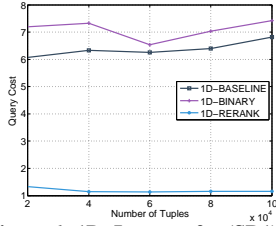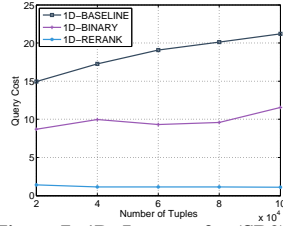
Figure 6: 1D: Impact of n (SR1)

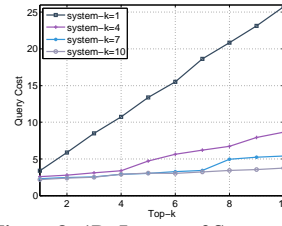**Figure 7: 1D: Impact of n (SR2)**

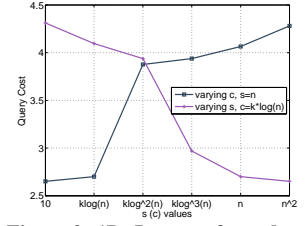**Figure 8: 1D: Impact of System-k**
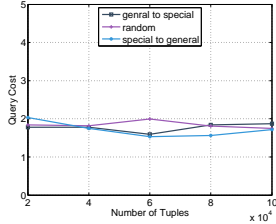

Figure 9: 1D: Impact of s and c

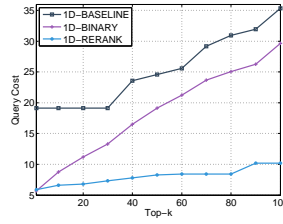**Figure 10: 1D: Impact of Query order in 1D-RERANK**

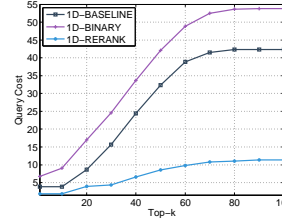**Figure 11: 1D: Top$k$ Query Cost (BN)**
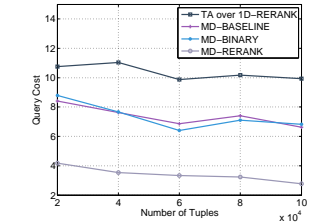

Figure 12: 1D: Top$k$ Query Cost (YA)

Figure 13: MD: Impact of n (SR1)

test whether the order of user queries have a major effect on the performance of 1D-RERANK, we ran an experiment using SR1 with three query-issuing orders: (1) from low to high selectivity (i.e., from more general to narrower queries), (2) from high to low selectivity, and (3) in a random order. Figure 10 shows that the query issuance order has a negligible effect on the query cost of 1D-RERANK.

6.2.2 Online Experiments

We also conducted two live experiments over Blue Nile and Yahoo! Autos, aiming to retrieve the top-100 tuples for each of the user query in the workload. The default system-k for BN and YA are 30 and 15, respectively, with the system ranking function being the default for each website, i.e., descending value of *price per carat* for BN and *distance from the pre-defined location* for YA.

Figures 11 and 12 show the average query cost for retrieving top-h tuples. As expected, 1D-RERANK significantly outperforms the other algorithms for both websites. For BN, even though 1D-BINARY performed well in the beginning, it required higher query cost for large values of h. That is because it keeps dividing the search area in half until the issued query becomes valid, thus it will probably end up with a valid query that contains fewer number of tuples. Considering the fact that here if the last (valid) query return k' tuples, we do not need to issue any queries for next $k' - 1$ Get-Next calls, having less number of tuples in the valid queries means less saving in the query cost. 1D-BASELINE, on the other hand, uses the current top tuple as the upper-bound to query the whole remaining region and has a higher chance to end up with a valid query with more tuples, which means more saving in the query cost for the next Get-Next operations. That is why in this experiment (Figure 11), after discovering the Top-1 tuple, it did not require more queries for the next 29 Get-Next operations.

6.3 MD Experiments

We compare the performance of MD-RERANK against "TA over 1D-RERANK", MD-BASELINE and MD-BINARY.
Constructing Workload of User Preference Queries: The workload is constructed using a process similar to one described in § 6.2, by selecting a subset from the set of all ranking attributes and choosing different weights between 0 and 1 for each of them. The workload consists of 32, 12 and 10 queries for DOT, BN and YA, respectively, of which 8, 3 and 2 do not have any filtering conditions.

6.3.1 Experiments over the Real-world Dataset

Impact of Database Size and System Ranking Function: The experimental setup was similar to the 1D experiments in § 6.2. We evaluated our algorithms for different database sizes and system ranking functions SR1 and SR2. Figures 13 and 14 shows the results for SR1 and SR2 respectively. In both cases, the algorithm MD-RERANK significantly outperformed all three competing baselines. One may notice an increase in the query cost of the algorithms when n increases in Figures 14, and a decrease in Figures 13. That is because when system and user-specified ranking function are anti-correlated, the more tuples database has, the more queries are required to find top tuples for the user-specified ranking function (since more tuples are ranked higher than them based on SR2). The case is vice-verse for SR1.
Impact of System-k: We then varied k, the number of tuples returned by the web database and measured the average query cost to retrieve top-10 tuples for the query workload. Figure 15 shows the results. As expected, higher values of system-k required lesser query cost to obtain the top-10 tuples. When $k = 1$, our algorithms were not able to use the savings by valid queries resulting in a substantial query cost.

6.3.2 Online Experiments

We applied MD-RERANK, as well as TA over 1D-RERANK, to retrieve the top-100 tuples for each query in the workload. Figure 16 shows the average query cost for the BN experiment. As shown in the figure, MD-RERANK outperformed TA significantly. The results for YA experiment is reflected in Figure 17. The substantial difference in query cost of the algorithms can be explained by the observation by the negative correlation between the ranking tuples in YA queries (for example the cars with higher mileage are probably cheaper). Hence TA algorithm had to issue many Get-Next operations before it finds the top tuples.

7. RELATED WORK

Top-k discovery methods can get divided in three main categories: *(sorted/random) access-based* methods, *layering-based* approaches, and *view-based* techniques. The first series of algorithms take the advantage of the data access methods. For example, *NRA* [9] assumes the existence of one sorted list of tuples for each attribute, and finds the Top-k only by exploring the lists, while *TA* [9] applies both random and sorted access. The more advanced algorithms in
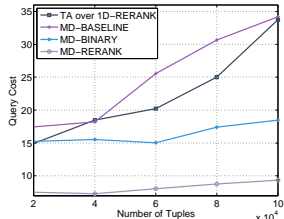
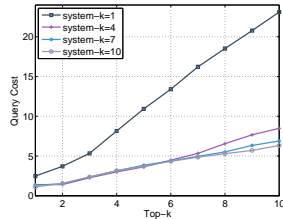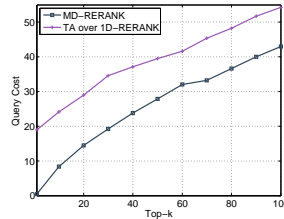**Figure 14: MD: Impact of n (SR2)**   **Figure 15: MD: Impact of System-k**   **Figure 16: MD: Top$k$ Query Cost (BN)**   **Figure 17: MD: Top$k$ Query Cost (YA)**
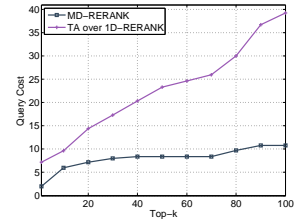
this category are *CA* [9], *Upper/Pick* [2], and [13]. The next category is the set of algorithms, such as *ONION* [4] and [19], that pre-process the data and index the layers of extremum tuples that gaurantee including the Top-k. View-based methods such as *PREFER* [10] and *LPTA* [6], employ the materialized views to increase the efficiency of Top-k discovery process. While prior work focused on minimizing the storage overhead of indices/materialized views and the computational overhead of processing top-k queries, we have to focus on minimizing the number of queries issued to the underlying database. This fundamentally different data access model also leads to a different cost model. For example, many prior work, such as [9] and [3], assume a separate cost for accessing each attribute and/or evaluating each predicate in the top-k query, while in our problem *all* attributes of a tuple are returned at once.

Hidden Databases Most of the prior works on the hidden databases relate to sampling, crawling the database, and aggregate estimation. Prior works such as [7, 18] propose efficient algorithms for collecting unbiased low-variance random samples of a given hidden databasee and [8, 11] provide unbiased aggregate estimators. While [12, 14, 15] aim toward crawling the whole hidden database, [1] only crawls the maxima index.

Top-k queries over Hidden Databases As the best of our knowledge, this is the first paper on reranking the query results of a hidden database. The only prior work about Top-k in hidden databases is [16]. Assuming the full knowledge of the system ranking function and attribute domains, its goal is to go beyond the Top-k limitation of the database interface, by partitioning the query space.

8. FINAL REMARKS

In this paper, we introduced a novel problem of query reranking, a third-party service that takes a client-server database with a proprietary ranking function and enables query processing according to any user-specified ranking function. To enable query reranking while minimizing the number of queries issued to the underlying database, we develop 1D-RERANK and MD-RERANK for user-specified ranking functions that involve only one attribute and any arbitrary set of attributes, respectively. Theoretic analysis and extensive experimental results on real-world databases, in offline and online settings, demonstrate the effectiveness of our techniques and their superiority over baseline solutions.

9. ACKNOWLEDGMENTS

10. REFERENCES

[1] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das. Discovering the skyline of web databases. *VLDB*, 2016.

[2] N. Bruno, S. Chaudhuri, and L. Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *TODS*, 2002.

[3] K. C.-C. Chang and S.-w. Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *SIGMOD*. ACM, 2002.

[4] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD*, 2000.

[5] J. Chomicki. Preference formulas in relational queries. *TODS*, 2003.

[6] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-k queries using views. In *VLDB*, 2006.

[7] A. Dasgupta, G. Das, and H. Mannila. A random walk approach to sampling hidden databases. In *SIGMOD*, 2007.

[8] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das. Unbiased estimation of size and other aggregates over hidden web databases. In *SIGMOD*, 2010.

[9] R. Fagin, A. Lotem, and M. Naor. Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences*, 66(4):614–656, 2003.

[10] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB Journal*, 2004.

[11] W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das. Aggregate estimation over dynamic hidden web databases. *VLDB*, 2014.

[12] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy. Google's deep web crawl. *VLDB*, 2008.

[13] A. Marian, N. Bruno, and L. Gravano. Evaluating top-k queries over web-accessible databases. *ACM Trans. Database Syst.*, 29(2), 2004.

[14] S. Raghavan and H. Garcia-Molina. Crawling the hidden web. *VLDB*, 2000.

[15] C. Sheng, N. Zhang, Y. Tao, and X. Jin. Optimal algorithms for crawling a hidden database in the web. *VLDB*, 2012.

[16] S. Thirumuruganathan, N. Zhang, and G. Das. Breaking the top-k barrier of hidden web databases. In *ICDE*. IEEE, 2013.

[17] S. Thirumuruganathan, N. Zhang, and G. Das. Rank discovery from web databases. *VLDB*, 2013.

[18] F. Wang and G. Agrawal. Effective and efficient sampling methods for deep web aggregation queries. In *EDBT*, 2011.

[19] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In *VLDB*, 2006.

[20] P. B. Yale. *Geometry and symmetry*. Courier Corporation, 1968.