Minimal Probing: Supporting Expensive Predicates for Top-k Queries

Kevin Chen-Chuan Chang, Seung-won Hwang
Computer Science Department, University of Illinois at Urbana-Champaign
{kcchang,shwang5}@cs.uiuc.edu

ABSTRACT

This paper addresses the problem of evaluating ranked *top-k* queries with expensive predicates. As major DBMSs now all support expensive user-defined predicates for Boolean queries, we believe such support for ranked queries will be even more important: First, ranked queries often need to model user-specific concepts of preference, relevance, or similarity, which call for dynamic user-defined functions. Second, middleware systems must incorporate external predicates for integrating autonomous sources typically accessible only by per-object queries. Third, fuzzy joins are inherently expensive, as they are essentially user-defined operations that dynamically associate multiple relations. These predicates, being dynamically defined or externally accessed, cannot rely on index mechanisms to provide zero-time sorted output, and must instead require per-object probe to evaluate. The current standard sort-merge framework for ranked queries cannot efficiently handle such predicates because it must completely probe all objects, before sorting and merging them to produce *top-k* answers. To minimize expensive probes, we thus develop the formal principle of "necessary probes," which determines if a probe is absolutely required. We then propose Algorithm *MPro* which, by implementing the principle, is provably optimal with minimal probe cost. Further, we show that *MPro* can scale well and can be easily parallelized. Our experiments using both a real-estate benchmark database and synthetic datasets show that *MPro* enables significant probe reduction, which can be orders of magnitude faster than the standard scheme using complete probing.

1. INTRODUCTION

In the recent years, we have witnessed significant efforts in processing ranked queries that return *top-k* results. Such queries are crucial in many *data retrieval* applications that retrieve data by "fuzzy" (or "soft") conditions that basically model similarity, relevance, or preference: A multimedia database may rank objects by their "similarity" to an example image. A text search engine orders documents by their "relevance" to query terms. An e-commerce service may sort their products according to a user's "preference" criteria [1] to facilitate purchase decisions. For these applications, Boolean queries (*e.g.*, as in SQL) can be too restrictive as they do not capture partial matching. In contrast, a ranked query computes the scores of individual *fuzzy predicates* (typically normalized in [0:1]), combines them with a *scoring function*, and returns a small number of *top-k* answers.

Example 1: Consider a real-estate retrieval system that maintains a database house(id, $price$, $size$, zip, age) with houses listed for sale. To search for *top*-5 houses matching her preference criteria, a user (*e.g.*, a realtor or a buyer) may formulate a ranked query as:

select id **from** house
where $new(age)\ x$, $cheap(price, size)\ p_c$,
$large(size)\ p_l$
order by $min(x, p_c, p_l)$ **stop after** 5 **(Query 1)**

Using some interface support, the user describes her preferences over attributes *age*, *price*, and *size* by specifying fuzzy predicates new, $cheap$, and $large$ (or x, p_c, and p_l for short). For each object, each predicate maps the input attributes to a score in [0:1]. For example, a house a with age=2 years, $price$ = \$150k, and $size$ = 2000 sqft may score $new(2)$=0.9, $cheap$(150k, 2000) = 0.85, and $large$(2000)=0.75. The query specifies a scoring function for combining the predicates, *e.g.*, the overall score for house a is min(0.9, 0.85, 0.75) = 0.75. The highest-scored 5 objects will be returned. ∎

This paper studies the problem of supporting expensive predicates for ranked queries. We characterize *expensive predicates* as those requiring a call, or a *probe*, of the corresponding function to evaluate an object. They generally represent any *non-index* predicates: When a predicate is dynamically defined or externally accessed, a pre-constructed access path no longer exists to return matching objects in "zero time." For instance, in Example 1, suppose predicate *cheap* is a user-defined function given at query time, we must invoke the function to evaluate the score for each object. We note that, for Boolean queries, similar expensive predicates have been extensively studied in the context of extensible databases [2, 3]. In fact, major DBMSs (*e.g.*, Microsoft SQL Server, IBM DB2, Oracle, and PostgreSQL) today all support user defined functions (which are essentially expensive predicates) allowing users to implement predicate functions in a general programming language. We believe it is important for ranked queries to support such predicates, which is the specific contribution of this paper.

In fact, there are many good reasons for supporting expensive predicates, because many important operations are essentially expensive. First, supporting expensive predicates will enable *function extensibility*, so that a query can employ user or application-specific predicates. Second, it will enable *data extensibility* to incorporate external data sources (such as a Web service) to answer a query. Third, it will enable *join* operations across multiple tables; as we will see, a fuzzy join predicate is essentially expensive. As far as

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA
Copyright 2002 ACM 1-58113-497-5/02/06 ...\$5.00.

we know, we are the first to generally support expensive predicates in the context of ranked queries, in order to handle these operations:

- **Function Extensibility:** User-defined functions are expensive because they are dynamically defined and thus require per-object evaluation. Note that user-defined functions are critical for function extensibility of a database system, to allow queries with non-standard predicates. While user-defined functions are now commonly supported in Boolean systems, we believe such functions will be even more important for ranked queries, because they are mainly intended for data retrieval based on similarity, relevance, and preference (*e.g.*, as in [1]). As these concepts are inherently imprecise and user (or application) specific, a practical system should support ad-hoc criteria to be specifically defined (by users or application programmers). Consider our real-estate example. Although the system may provide *new* as built-in, users will likely have different ideas about *cheap* and *large* (say, depending on their budget and family sizes). It is thus desirable to support these ad-hoc criteria as user-defined functions to express user preferences.
- **Data Extensibility:** A middleware system can integrate an "external" predicate that can only be evaluated by probing an autonomous source for each object. Such integration may take place within a loosely coupled system (*e.g.*, a relational DBMS and an image search engine). Further, a middleware may integrate Web sources, which is more common than ever in the Internet age. For instance, in Example 1, to look for "safe" areas, the user may use an external predicate $safe(zip)$ that computes the "safety" score for a given *zip* code by querying some Web source (*e.g.*, www.apbnews.com) for the crime rate of the area.
- **Joins:** Join predicates are expensive, because they are inherently user-defined operations: In general, a join operation can dynamically associate any attributes from multiple tables (as in the Boolean context such as SQL), and in addition the associating function can be user-defined. Consequently, since a search mechanism cannot be pre-computed, fuzzy joins requires expensive probes to evaluate each combined tuple (in the Cartesian product), as is the case in Boolean queries. To generally support fuzzy joins, a ranked-query system must support expensive predicates. For instance, continuing Example 1, to find new houses near a high-rating park, the following query joins another relation park(*name*, *zip*) with the predicate *close*:

  select $h.id$, $s.name$ **from** house h, park s
  where $new(h.age)\ x$, $rating(s.name)\ p_r$,
  $close(h.zip, s.zip)\ p_j$
  order by $min(x, p_r, p_j)$ **stop after** 5 **(Query 2)**

While widely supported for Boolean queries in major DBMSs, expensive predicates have not been considered for ranked queries. These predicates, be they user defined or externally accessed, can be arbitrarily expensive to probe, potentially requiring complex computation or access to networked sources. Note that it may appear straightforward to "transform" an expensive predicate into a "normal" one: By probing every object for its score, one can build a search index for the predicate (to access objects scored above a threshold or in the sorted order), as required by the current processing frameworks [4, 5, 6, 7, 8, 9, 10] (Section 3.3). This naive approach requires a sequential scan, or *complete probing*, over the entire database: A database of N objects will need N sequential probes for each expensive predicate. Such complete probing at query time is clearly unacceptable in most cases.

This paper addresses probe predicates for ranked queries. When a ranked query contains expensive predicates, the key challenge is to minimize the number of probes. As users only ask for some *top-k* answers, is complete probing necessary? Our results show that the vast majority of the probes may be unnecessary, thus making expensive predicates practical for ranked queries. For instance, to retrieve *top*-10 (*i.e.*, $k = 10$) from a benchmark database constructed with real-life data, the naive scheme can waste 97% of the complete probes (Section 6). To enable *probe minimization*, we develop the formal principle of *necessary probes*, which determines if a particular probe is absolutely necessary for finding k top answers. We thus present Algorithm *MPro* which ensures minimal probing, *i.e.*, it performs only necessary probes and is provably optimal.

Further, we discuss several useful extensions of the algorithm. First, we show that the algorithm directly supports incremental processing, to return top answers progressively, paying only incremental cost. Second, it is easily "parallelizable," to enhance performance over large datasets. The parallelization can in principle give linear speedup. Third, we show that the algorithm can scale well; the cost will scale sublinearly in database size. In addition, *MPro* can be immediately generalized for approximate processing. Since approximate answers are often acceptable in ranked queries (which are inherently "imprecise"), we extend *MPro* to enable trading efficiency with accuracy– which we report in [11].

Note that this paper concentrates on the algorithmic framework for supporting expensive predicates, and not on other related issues. In particular, a practical system must provide a friendly interface for users or application programmers to easily specify user-defined predicates. To study this issue, we are currently building a GUI front-end for our prototype testbed, the real-estate retrieval system. There are also other extensions to our algorithm: It can easily take advantage of *predicate caching*, where expensive probes can be reused, *e.g.*, during query refinement when predicates are repeated in subsequent queries. To highlight, we summarize the main contributions of this paper as follows:

- **Expensive predicates for ranked queries:** We identify, formulate, and formally study the *expensive predicate* problem for ranked queries, which generally abstracts user-defined functions, external predicates, and fuzzy joins. We are not aware of other previous work that formulates the general problem of supporting expensive predicates for *top-k* queries.
- **Necessary-probe principle:** We develop a simple yet effective principle for determining necessary probes, which can be critical for any algorithms that attempt probe optimization.
- **Probe-optimal algorithm:** We present Algorithm *MPro*, which minimizes probe cost for returning *top-k* answers.
- **Experimental evaluation:** We report extensive experiments using both real-life and synthesized datasets. Our experimental study indicates the effectiveness and practicality of our approach.

We briefly discuss related work in Section 2, and then start in Section 3 by defining ranked queries, the cost model, and the baseline processing scheme. Section 4 presents the necessary-probe principle for optimization, based on which we develop Algorithm *MPro*. Section 5 then discusses several extensions of the basic algorithm. Section 6 reports our experimental evaluation. Due to space limitations, we leave some additional results (*e.g.*, proof, approximation, and more experiments) to an extended report [11].

2. RELATED WORK

Expensive predicates have been studied for Boolean queries to support user-defined functions. Several works (*e.g.*, [2, 3]) address processing expensive predicates efficiently. As Section 1 discussed, all current major DBMSs (*e.g.*, Microsoft SQL Server, IBM DB2,

Oracle, and PostgreSQL) support such predicates.

Top-k queries have been developed recently in two different contexts. First, in a middleware environment, Fagin [7, 8] pioneered ranked queries and established the well-known $\mathcal{A}_0$ algorithm (with its improvements in [9, 10]). [12] generalizes to handling arbitrary θ-joins as combining constraints. As Section 3 discusses, these works assume sorted access of search predicates. This paper thus studies probe predicates without efficient sorted access.

Secondly, ranked queries were also proposed as a layer on top of relational databases, by defining new constructs (for returning "top" answers) and their processing techniques. For instance, [13] proposes new SQL clauses **order by** and **stop after**. Carey et al. [14, 15] then present optimization techniques for exploiting **stop after**, which limits the result cardinalities of queries.

In this relational context, references [4, 5] study more general ranked queries using scoring functions. In particular, [4] exploits indices for query search, and [5] maps ranked queries into Boolean range queries. Recently, PREFER [6] uses materialized "views" to evaluate preference queries defined as linear sums of attribute values. These works assume that scoring functions directly combine attributes (which are essentially search predicates). We aim at supporting expressive user-defined predicates, which we believe are essential for ranked queries, as Section 1 discussed.

This paper identifies and formulates the general problem of supporting expensive predicates for ranked queries, providing unified abstraction for user-defined functions, external predicates, and fuzzy joins. [16] develops an IR-based similarity-join; we study general fuzzy joins as arbitrary probe predicates. More recently, around the same time of our work [11], some related efforts emerge, addressing the problems of θ-joins [12] (which are Boolean and not fuzzy, as just explained) and external sources [17]. In contrast, we address a more general and unified problem of expensive predicates.

Our general framework, in searching for *top-k* answers, adopts the branch-and-bound or best-first search techniques [18]. In particular, Algorithm *MPro* can be cast as a specialization of $A*$. Several other works [17, 12, 16] also adopt the same basis. Our work distinguishes itself in several aspects: First, we aim at a different or more general problem (as contrasted above). Second, our framework separates *object search* (finding objects to probe) from *predicate scheduling* (finding predicates to execute). We believe this "clean" separation allows us to develop the formal notion of necessary probes (Section 4.1) and consequently a probe-optimal algorithm (Section 4.2). The scheduling (Section 4.3) is thus separated as a sampling-based *optimization* phase before (or reoptimization during) execution. Further, based on the simple framework, we develop several useful extensions, *e.g.*, parallelization (Section 5.3) and approximation [11], as well as analytical study of probe scalability (Section 5.4).

3. RANKED QUERY MODEL

To establish the context of our discussion, this section describes the semantics (Section 3.1) of ranked queries and a cost model for expensive predicates (Section 3.2). Section 3.3 then discusses query processing using the standard sort-merge framework to motivate and contrast our work.

3.1 Query Semantics

Unlike Boolean queries where results are flat sets, ranked queries return sorted lists of *objects* (or tuples) with scores indicating how well they match the query. A ranked query is specified by a scoring function $\mathcal{F}(t_1, \ldots, t_n)$, which combines several fuzzy *predicates* $t_1, \ldots, t_n$ into an overall *query score* for each object. Without loss of generality, we assume that scores (for individual predicates or entire queries) are in $[0 : 1]$. We denote by $t[u]$ the score of predicate t for object u, and $\mathcal{F}[u]$ the query score.

OID	x	p_c	p_l	$\mathcal{F}(x, p_c, p_l)$
a	0.90	0.85	0.75	0.75
b	0.80	0.78	0.90	0.78
c	0.70	0.75	0.20	0.20
d	0.60	0.90	0.90	0.60
e	0.50	0.70	0.80	0.50

Figure 1: Dataset 1 for query $\mathcal{F}(x, p_c, p_l) = min(x, p_c, p_l)$.

We will use Query 1 (of Example 1) as a running example, which combines predicates x, p_c, and p_l with $\mathcal{F}= min(x, p_c, p_l)$. We will illustrate with a toy example (dataset 1) of objects $\{a, b, c, d, e\}$. Figure 1 shows how they score for each predicate (which will not be known until evaluated) and the query; *e.g.*, object a has scores $x[a] = 0.9$, $p_c[a] = 0.85$, $p_l[u] = 0.75$, and overall $\mathcal{F}(x, p_c, p_l)[a] = \mathcal{F}(x[a], p_c[a], p_l[a]) = min(0.9, 0.85, 0.75) = 0.75$.

We can distinguish between selection and join predicates, as in relational queries, depending on if the operation involves one or more objects. A *selection* predicate evaluates some attributes of a single object, and thus discriminates (or "selects") objects in the same table by their scores; *e.g.*, in Query 1, x determines how "new" the *age* of a house is. In contrast, a *join* predicate evaluates (some attributes of) multiple objects from multiple tables. (We can thus view a join operation as a selection over joined tuples in the Cartesian product of the joining tables.) For instance, p_j in Query 2 evaluates each pair of `house` and `park` to score their closeness. Our framework can generally handle both kinds of predicates– we will focus on selections for simplicity, and discuss the extensions for joins in Section 5.2.

In this paper we focus on an important class of scoring functions that are *monotonic*, which is typical for ranked queries [7]. Intuitively, in a monotonic function, all the predicates influence *positively* the overall score. Formally, $\mathcal{F}$ is monotonic if $\mathcal{F}(t_1, \ldots, t_n) \geq \mathcal{F}(s_1, \cdots, s_n)$ when $\forall i : t_i \geq s_i$. Note that this monotonicity is analogous to disallowing negation (*e.g.*, $t_1 \wedge \neg t_2$) in Boolean queries. Since negation is used only infrequently in practice, we believe that monotonic functions will similarly dominate for ranked queries. Note that a scoring function may be given explicitly (in a query) or implicitly (by the system). For instance, a system that adopts fuzzy logic [19] will use $\mathcal{F} = min(t_1, t_2)$ as the fuzzy conjunction. An image or text [20] database may combine various features with a user-transparent function, such as Euclidean distance or weighted average.

As results, a ranked query returns the *top-k* objects with highest scores, and thus also referred to as a *top-k* query. More formally, given *retrieval size* k and scoring function $\mathcal{F}$, a ranked query returns a list $\mathcal{K}$ of k objects (*i.e.*, $|\mathcal{K}| = k$) with query scores, sorted in a descending order, such that $\mathcal{F}(t_1, \cdots, t_n)[u] > \mathcal{F}(t_1, \cdots, t_n)[v]$ for $\forall u \in \mathcal{K}$ and $\forall v \notin \mathcal{K}$. For example, the *top*-2 query over dataset 1 (Figure 1) will return the list $\mathcal{K}$ = (b:0.78, a:0.75). Note that, to give deterministic semantics, we assume that there are no ties – otherwise, a *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by unique object IDs.

Note that the top-k interface is fundamental in supporting other interfaces. It is straightforward to extend our top-k algorithm (Section 4) to support the *incremental access* (or iterator) as well as *threshold* interface. We discuss these extensions in Section 5.1.

3.2 Cost Model

Given a query, a processing engine must combine predicate scores to find the *top-k* answers. We can generally distinguish between index predicates that provide efficient search and non-index pred-

icates that require per-object probes. First, system built-in predicates (*e.g.*, object attributes or standard functions adopted in [4, 5, 6]) can use pre-computed indexes to provide *sorted access* of objects in the descending order of scores. Such *search predicates* are essentially of zero cost, because they are not actually evaluated, rather the indexes are used to search "qualified" objects. For instance, our real-estate example may provide $nearcity(h, C)$ for sorting houses h closest to a given city C (*e.g.*, k-nearest-neighbor search). For Query 1, we assume x to be a search predicate. Figure 1 orders objects to stress the sorted output of x.

In contrast, expensive predicates must rely on per-object *probes* to evaluate, because of the lack of search indexes. As Section 1 explains, such *probe predicates* generally represent user-defined functions, external predicates, and joins. Unlike search predicates that are virtually free, such predicates can be arbitrarily expensive to probe, potentially requiring complex computation (for user-defined functions), networked access to remote servers (for external predicates), or coping with combinatorial Cartesian products (for joins). Our goal is thus clear: to minimize probe cost.

This paper thus presents a framework for the evaluation of rank queries with probe predicates. To stress this focus, we assume (without loss of generality) queries of the form $\mathcal{F}(x, p_1, \ldots, p_n)$, with a search predicate x and some probe predicate p_i. Note that, when there are several search predicates $(x_1, \ldots, x_m)$, the well-known Fagin's algorithm [7] (Section 3.3 will discuss this standard "sort-merge" framework) can be compatibly applied to merge them into a single sorted stream x (thus in the above abstraction), which also minimizes the search cost.

We want to minimize the cost of probing $p_1, \ldots, p_n$ for answering a *top-k* query. Let $\mathcal{A}$ be an algorithm, we denote its probe cost by $\mathcal{PC}(\mathcal{A})$. Assuming that the per-probe cost for p_i is C_i and that $\mathcal{A}$ performs N_i probes for p_i, we model the probe cost as $\mathcal{PC}(\mathcal{A}) = \sum_{i=1}^{n} N_i C_i$. In particular, for a database of size N, a complete probing system (as Section 3.3 will discuss) will cost $\sum_{i=1}^{n} N C_i$. Such complete probing cost represents an *upper bound* of any algorithms; many probes are obviously unnecessary for finding a small number of *top-k* answers.

Our goal is to develop a *probe-optimal* algorithm, which will sufficiently guarantee minimal probe cost. If $\mathcal{A}$ is probe-optimal, it does not waste any single probes; every probe it performs is necessary (by any algorithms). Since probe-optimality implies that each N_i is minimal, $\mathcal{PC}(\mathcal{A})$ overall must be minimal. We will present a probe-optimal algorithm that performs only necessary probes.

We stress that, for ranked queries, it is important to make the cost "proportional" to retrieval size k rather than database size N, as k can often be several orders of magnitude smaller (*e.g.*, $k = 5$ out of $N = 1000$). (Section 6 shows that our algorithm indeed illustrates this property.) This proportional cost can be critical, since users are typically only interested in a small number of top results.

3.3 Baseline: the Sort-Merge Framework

There have been significant efforts in processing ranked queries with "inexpensive" search predicates. The most well-known scheme (in a middleware system) has been established by Fagin [7] as well as several later versions [9, 10]. Assuming only search predicates, these schemes access and merge the sorted-output streams of individual predicates, until *top-k* answers can be determined. We refer to such processing as a *sort-merge* framework.

In this paper we consider ranked queries with expensive predicates. Can we simply adopt the sort-merge framework? As the name suggests, this standard framework will require complete probing to *sort* objects for each expensive predicate, before *merging* the sorted streams– Referred to as *SortMerge*, this naive scheme fully

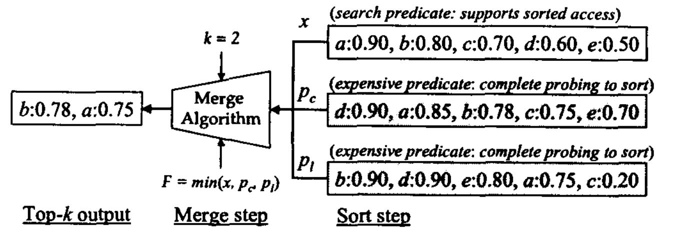


Figure 2: The baseline scheme *SortMerge*.

"materializes" probe predicates into search predicates. Figure 2 illustrates Scheme *SortMerge* for Query 1, where x is a search predicate while p_c and p_l expensive ones. The scheme must perform complete probing for both p_c and p_l to provide the sorted access, and then merge all the three streams to find the *top*-2 answers.

Note that, rather than the desired "proportional cost," *SortMerge* always requires complete probing, regardless of the small k; *i.e.*, $\mathcal{PC}(\textit{SortMerge}) = \sum_{i=1}^{n} N C_i$, for a database of size N. Such "sequential-scan" cost can be prohibitive in most applications, when the database is of an interesting size. This paper addresses the problem of minimizing this probe cost. In fact, as we will see, most probes in complete probing are simply unnecessary– Section 6 will experimentally compare the baseline scheme to our algorithm.

4. MINIMAL PROBING: ALGORITHM *MPro*

Given a ranked query characterized by scoring function $\mathcal{F}(x, p_1, \ldots, p_n)$ and retrieval size k, with a search predicate x and probe predicates $p_1, \ldots, p_n$, our goal is to minimize the probe cost for answering the query. Toward this goal, we must confront two key issues: 1) What is the minimal-possible probe cost? 2) How can we design an algorithm that minimizes the cost. This section develops a formal principle (the Necessary-Probe Principle) in Section 4.1 for answering the former and proposes an algorithm (Algorithm *MPro*) in Section 4.2 for the latter. Section 4.3 discusses the scheduling of probe predicates.

4.1 Necessary Probes

In order to minimize probe cost, we must determine if a probe is necessary for finding *top-k* answers with respect to scoring function $\mathcal{F}(x, p_1, \ldots, p_n)$. This section presents our results on such a principle (*i.e.*, Theorem 1).

To begin with, for each object, we sequentially execute probes and incrementally determine if further probes are necessary. (Section 5.3 will extend to parallel probing.) We thus need a *predicate schedule* as a sequence of $p_1, \ldots, p_n$, which defines their execution order (if probed at all). For our example $\mathcal{F}(x, p_c, p_l)$, two schedules are possible: (p_c, p_l) if p_c is evaluated before p_l or (p_l, p_c) otherwise. In general, each object u may define its own *object schedule* $\mathcal{H}^u$, or all objects may share the same *global schedule* $\mathcal{H}$. Our main algorithm (Algorithm *MPro*) assumes as input such schedules (object or global) and is thus independent of the scheduling. For simplicity, this section refers only to a single global schedule $\mathcal{H}$; the same results hold when objects have their own schedules. While such scheduling is NP-hard in general [11], Section 4.3 will discuss an online sampling-based scheduler that effectively (almost always) finds the best schedules by greedily scheduling more "cost-effective" predicates.

Given a schedule $\mathcal{H}$, starting from the sorted output of x, how shall we proceed to probe the predicates for each object, in order to minimize such accesses? On one hand, it is clear that *some* objects must be fully probed (for every p_i), which include at least the *top-k* answers in order to determine their query scores and ranks in the query results (Section 3.1). On the other hand, since only some

top-k answers are requested, is complete probing (for every object) necessary? To avoid this prohibitive cost, our goal is to stop as early as possible for *each* object. In fact, some objects may not need to be probed at all, if they can never be the top answers.

To enable probe minimization, we must determine if a particular probe is truly *necessary* for finding the *top-k* answers. During query processing, for any object u, if p is the next predicate on the schedule (global or object-specific), we may probe p for u in order to determine the score $p[u]$. We want to determine if such a probe, designated by $\mathsf{pr}(u,p)$, is necessary. The following definition formalizes our notion of a necessary probe.

Definition 1 (Necessary Probes): Consider a ranked query with scoring function $\mathcal{F}$ and retrieval size k. A probe $\mathsf{pr}(u,p)$, which probes predicate p for object u, is *necessary*, if the *top-k* answers with respect to $\mathcal{F}$ cannot be determined by any algorithm without performing the probe, regardless of the results of other probes. ■

We stress that this notion of necessary probes is essential for optimizing probe cost. According to the definition, a probe is necessary only if it is *absolutely* required– thus it must be performed by an optimal algorithm: First, the probe is required independent of algorithms– any algorithm that returns correct *top-k* answers must pay the probe. Second, the probe is required independent of the outcomes of other probes– it may be performed before or after others, but the particular order will not change the fact that the probe is required. While this notion of necessary probes seems intuitively appealing, how can we determine if a *particular* probe is absolutely required? Further, given there are many possible probes (at least one for each object at any time), how many of them are actually necessary, and how to effectively find *all* the necessary probes? We next develop a simple principle for answering these questions.

Let's start with the first question: *how to determine if a probe is necessary?* To illustrate, consider finding the *top*-1 object for $\mathcal{F}(x, p_c, p_l)$ with dataset 1 (Figure 1), given schedule $\mathcal{H} = (p_c, p_l)$. Our starting point, before any probes, is the sorted output of search predicate x (Figure 1 sorts objects by their x scores). We thus can choose to probe the next predicate p_c for any object a, b, c, d, or e. Let's consider the top object a and determine if $\mathsf{pr}(a, p_c)$ is necessary. (We can similarly consider any other object.) The probe is actually necessary for answering the query, no matter how other probes turn out: Assume we can somehow determine the *top*-1 answer to be object u without probing $\mathsf{pr}(a, p_c)$.

- Suppose $u \neq a$: Note that $\mathcal{F}[u] = min(x[u],\ p_c[u],\ p_l[u]) \leq x[u]$, and $x[u] \leq 0.8$ for $u \neq a$ (see Figure 1). Consequently, $\mathcal{F}[u] \leq 0.8$. However, without probing $\mathsf{pr}(a, p_c)$ and then $\mathsf{pr}(a, p_l)$, u may not be safely concluded as the *top*-1. For instance, suppose that the probes would return $p_c[a] = 1.0$ and $p_l[a] = 1.0$, then $\mathcal{F}[a] = min(0.9, 1.0, 1.0) = 0.9$. That is, a instead of u should be the *top*-1, a contradiction.
- Suppose $u = a$: In order to output a as the answer, we must have fully probed a, including $\mathsf{pr}(a, p_c)$, to determine and return the query score.

Observe that, we determine if the probe on a is necessary essentially by comparing the upper bound or "ceiling score" of a to others. That is, while $\mathcal{F}[a]$ can be as high as its ceiling score of 0.9, any other object u cannot score higher than 0.8 (which is the ceiling score of b). In general, during query processing, before an object u is fully probed, the evaluated predicates of u can effectively bound its ultimate query score. Consider a scoring function $\mathcal{F}(t_1, \ldots, t_n)$. We define $\overline{\mathcal{F}}_T[u]$, the *ceiling score* of u with respect to a set T of evaluated predicate ($T \subseteq \{t_1, \ldots, t_n\}$), as the maximal-possible score that u may eventually achieve, given the predicate scores in T. Since $\mathcal{F}$ is monotonic, the ceiling score can be generally obtained by Eq. 1 below, which simply substitutes unknown predicate scores with their maximal-possible value 1.0. The monotonicity of $\mathcal{F}$ ensures that the ceiling score give the upper bound when only T is evaluated, *i.e.*, $\mathcal{F}[u] \leq \overline{\mathcal{F}}_T[u]$.

$$\overline{\mathcal{F}}_T(t_1, \ldots, t_n)[u] = \mathcal{F}\left(\begin{array}{ll} t_i = t_i[u] & \text{if } t_i \in T \\ t_i = 1.0 & \text{otherwise.} \end{array} \forall i \right) \quad (1)$$

To further illustrate, after $\mathsf{pr}(a, p_c)$, what shall we probe next? Intuitively, at least we have choices of $\mathsf{pr}(a, p_l)$ (to complete a) or $\mathsf{pr}(b, p_c)$ (to start probing b). Similarly to the above, we can reason that the further probe on a, $\mathsf{pr}(a, p_l)$, is still necessary. To contrast, we show that $\mathsf{pr}(b, p_c)$ is not necessary at this point, according to Definition 1. (However, as more probes are done, at a later point, $\mathsf{pr}(b, p_c)$ may become necessary.) With $x[b] = 0.8$ evaluated, we compute the ceiling score of b as $\overline{\mathcal{F}}_{\{x\}}[b] = 0.8$. Whether we need to further probe b in fact *depends* on other probes, namely the remaining probe $\mathsf{pr}(a, p_l)$ of a. (Note that a already has $x[a] = 0.9$ and $p_c[a] = 0.85$ evaluated.)

- Suppose that $\mathsf{pr}(a, p_l)$ returns $p_l[a] = 1$ and thus $\mathcal{F}[a] = 0.85$. For finding the *top*-1 answer, we do not need to further evaluate b because $\overline{\mathcal{F}}_{\{x\}}[b] = 0.8 < \mathcal{F}[a] = 0.85$, and thus b cannot make to the *top*-1. That is, depending on $p_l[a] = 1$, we can answer the query without probing $\mathsf{pr}(b, p_c)$.
- Suppose that $\mathsf{pr}(a, p_l)$ returns $p_l[a] = 0$ and thus $\mathcal{F}[a] = 0$. Now b becomes the "current" top object (with the highest ceiling score $\overline{\mathcal{F}}_{\{x\}}[b] = 0.8$). That is, depending on $p_l[a] = 0$, we can reason that $\mathsf{pr}(b, p_c)$ is necessary, similar to $\mathsf{pr}(a, p_c)$ above.

Further, we consider the second question: *how to find all the necessary probes?* Let u be any object in the database, and p be the next unevaluated predicate (on the schedule $\mathcal{H}$) for u. Potentially, any probe $\mathsf{pr}(u, p)$ might be necessary. However, it turns out that at any point during query processing, there will be *at most* k probes that are necessary, for finding *top-k* answers. That is, we can generalize our analysis (see Theorem 1) to show that only those probes for objects that are currently ranked at the *top-k* in terms of their ceiling scores are necessary. Note that this conclusion enables an efficient way to "search" necessary probes: by ranking objects in the order of their current ceiling scores. (As Section 3 discussed, we assume that a deterministic tie-breaker will determine the order of ties.) For any object u in the *top-k* slots, its next probe $\mathsf{pr}(u, p)$ is necessary. Theorem 1 formally states this result (see [11] for a proof).

Theorem 1 (Necessary-Probe Principle): Consider a ranked query with scoring function $\mathcal{F}$ and retrieval size k. Given a predicate schedule $\mathcal{H}$, let u be an object and p be the next unevaluated predicate for u in $\mathcal{H}$. The probe $\mathsf{pr}(u, p)$ is necessary, if there do not exist k objects $v_1, \ldots, v_k$ such that $\forall v_i : \overline{\mathcal{F}}_{T_u}[u] < \overline{\mathcal{F}}_{T_{v_i}}[v_i]$ with respect to the evaluated predicates T_u and T_{v_i} of u and v_i respectively. ■

We stress that the notion of necessary probes directly defines the minimal probe cost of any correct algorithm. First, Definition 1 generally isolates a class of (necessary) probes as those required by any algorithm. Further, Theorem 1 provides an "operational" definition to actually determine if a given probe is necessary as well as to effectively search those probes. Putting together, we immediately conclude that an algorithm will be probe-optimal (Section 3.2) if it only performs necessary probes, which we formally state in Lemma 1 (see [11] for a proof). Our goal is thus to design such an algorithm.

Algorithm *MPro*$(\mathcal{F}, k, \mathcal{H}, \mathcal{D})$: Minimal-probing algorithm
Input:
- $\mathcal{F}(x, p_1, \ldots, p_n)$: scoring function // *with expensive predicates* $p_1, \ldots, p_n$.
- k: retrieval size, *i.e.*, to return *top-k* answers.
- $\mathcal{H}$: schedule of $p_1, \ldots, p_n$.
- $\mathcal{D}$: input database // *assume selection predicates over single relation for simplicity.*

Output: $\mathcal{K}$, the *top-k* answers with respect to $\mathcal{F}$.
Procedure:
(1) Queue Initialization:
// *search* x *over* $\mathcal{D}$ *to prepare sorted output queue* $\mathcal{X}$.
- $\mathcal{X} \leftarrow$ evaluate x over $\mathcal{D}$
- $\mathcal{K} \leftarrow \{\}$; $\mathcal{Q} \leftarrow \{\}$ // $\mathcal{K}$: *output;* $\mathcal{Q}$: *ceiling queue to prioritize by ceiling score.*

// *initialize* $\mathcal{Q}$ *to buffer objects prioritized by their ceiling scores from* x.
// *this "full" initialization is only conceptual;* $\mathcal{X}.top()$ *can be on demand.*
- while ($\mathcal{X}$ is not empty):
  - $u \leftarrow \mathcal{X}.top()$ // *pop next top object out of* $\mathcal{X}$.
  - $T_u \leftarrow \{x\}$; $u.ceiling \leftarrow \overline{\mathcal{F}}_{T_u}[u]$ // *initialize ceiling score with* x.
  - $\mathcal{Q}.insert(u, u.ceiling)$ // *insert* u *into* $\mathcal{Q}$ *prioritized by its ceiling score.*

(2) Necessary Probing:
// *set up the stop condition* $\mathcal{SC}$ *for determining if to stop probing.*
- $\mathcal{SC} \leftarrow$ "$|\mathcal{K}| \geq k$, *i.e.*, there are at least k complete objects seen on the top"
- while ($\mathcal{SC} = False$): // *keep performing necessary probes until* $\mathcal{SC}$ *becomes true.*
  - $u \leftarrow \mathcal{Q}.top()$ // *the current top object with the highest ceiling score.*
  - if u is complete: // u *is among the first* k *objects to be completed.*
    - $u.score \leftarrow u.ceiling$; append u to $\mathcal{K}$ // *add* u *to be the top-k output.*
  - else: // u *as the top incomplete object must be probed further.*
    - $p \leftarrow$ next unevaluated predicate of u on schedule $\mathcal{H}$
    - $p[u] \leftarrow$ probe pr(u, p) // pr(u, p) *must be necessary by Theorem 1.*
    - $T_u \leftarrow T_u \cup \{p\}$; $u.ceiling \leftarrow \overline{\mathcal{F}}_{T_u}[u]$

      // *update the ceiling score of* u, *as* $p[u]$ *is just obtained.*
    - $\mathcal{Q}.insert(u, u.ceiling)$ // *insert* u *back to* $\mathcal{Q}$ *prioritized by* $u.ceiling$.

(3) Top-k Output: return in order each (u:$u.score$) in $\mathcal{K}$

Figure 3: Algorithm *MPro*

Lemma 1 (Probe-Optimal Algorithms): Consider a ranked query with scoring function $\mathcal{F}$ and retrieval size k. Given a predicate schedule $\mathcal{H}$, an algorithm $\mathcal{A}$ for processing the query is probe-optimal if $\mathcal{A}$ performs only the necessary probes as Theorem 1 specifies. ∎

4.2 Algorithm *MPro*

We next develop our Algorithm *MPro* (for minimal probing), which finds *top-k* answers with respect to scoring function $\mathcal{F}(x, p_1, \ldots, p_n)$. To be probe-optimal, based on Lemma 1, *MPro* optimizes probe cost by ensuring that every probe performed is indeed necessary (for finding k top answers).

Essentially, during query processing, *MPro* keeps "searching" for a necessary probe to perform next. Progressing with more probes, eventually *MPro* will have performed all (and only) the necessary probes, at which point the *top-k* answers will "surface." Note that Theorem 1 identifies a probe pr(u, p) as necessary if u is among the current *top-k* in terms of ceiling scores. Thus, a "search mechanism" for finding necessary probes can return top ceiling-scored objects when requested – *i.e.*, a priority queue [21] that buffers objects using their ceiling scores as priorities.

We thus design *MPro* to operate on such a queue, called *ceiling queue* and denoted $\mathcal{Q}$, of objects prioritized by their ceiling scores. As Figure 3 shows, *MPro* mainly consists of two steps: First, in the *queue initialization* step, starting with the sorted output stream $\mathcal{X}$ of search predicate x, *MPro* initializes $\mathcal{Q}$ based on initial ceiling scores $\overline{\mathcal{F}}_{\{x\}}[\cdot]$ with x being the only evaluated predicate (see Eq. 1). Note that, although for simplicity our discussion assumes that $\mathcal{Q}$ is fully initialized (by drawing every object from $\mathcal{X}$), this initialization is only conceptual: It is important to note that $\overline{\mathcal{F}}_{\{x\}}[\cdot]$ will induce the *same* order in $\mathcal{Q}$ as the $\mathcal{X}$ stream, *i.e.*, if $x[u] \leq x[v]$, then $\overline{\mathcal{F}}_{\{x\}}[u] \leq \overline{\mathcal{F}}_{\{x\}}[v]$, since $\mathcal{F}$ is monotonic. Thus *MPro* can access $\mathcal{X}$ incrementally to move objects into $\mathcal{Q}$ when more are needed for further probing. (It is entirely possible that some objects will not be accessed from $\mathcal{X}$ at all.)

Second, in the *necessary probing* step, *MPro* keeps on requesting from $\mathcal{Q}$ the top-priority object u, which has the highest ceiling score. If u is incomplete with the next unevaluated predicate p, according to Theorem 1, pr(u, p) is necessary. Thus *MPro* will perform this probe, update the ceiling score of u, and insert it back to $\mathcal{Q}$ by the new score. On the other hand, if u is already complete when it surfaces to the top, u must be among the *top-k* answers, because its *final* score is higher than the *ceiling* scores of objects still in $\mathcal{Q}$. That is, u has "surfaced" to the *top-k* answers, which *MPro* will move to an output queue, denoted $\mathcal{K}$ in Figure 3.

step	action	ceiling queue $\mathcal{Q}$	output($\mathcal{K}$)
1.	initialize $\mathcal{Q}$ and $\mathcal{K}$	a:0.90, b:0.80, c:0.70, d:0.60, e:0.50	{} (empty)
2.	probe pr(a, p_c)	a:0.85, b:0.80, c:0.70, d:0.60, e:0.50	{}
3.	probe pr(a, p_l)	b:0.80, $\underline{a{:}0.75}$, c:0.70, d:0.60, e:0.50	{}
4.	probe pr(b, p_c)	b:0.78, $\underline{a{:}0.75}$, c:0.70, d:0.60, e:0.50	{}
5.	probe pr(b, p_l)	$\underline{b{:}0.78}$, $\underline{a{:}0.75}$, c:0.70, d:0.60, e:0.50	{}
6.	pop top complete objects from $\mathcal{Q}$ into $\mathcal{K}$	c:0.70, d:0.60, e:0.50	$\underline{b{:}0.78}$, $\underline{a{:}0.75}$
7.	stop condition holds output $\mathcal{K}$	c:0.70, d:0.60, e:0.50	$\underline{b{:}0.78}$, $\underline{a{:}0.75}$

Figure 4: Illustration of Algorithm *MPro*.

Incrementally, more objects will complete and surface to $\mathcal{K}$, and *MPro* will eventually stop when there are k such objects (which will be the *top-k* answers). As Figure 3 shows, *MPro* checks this *stop condition*, designated $\mathcal{SC}$, to halt further probing. It is interesting to observe the "dual" interpretations of $\mathcal{SC}$: On one hand, $\mathcal{SC}$ tells that there are already k answers in $\mathcal{K}$, and thus no more probes are necessary. On the other hand, when $\mathcal{SC}$ holds, it follows from Theorem 1 that no more probes can be necessary, and thus the *top-k* answers must have fully surfaced, which is indeed the case. (We discuss in [11] how the stop condition can be "customized" for approximate queries.)

Figure 4 illustrates Algorithm *MPro* for our example of finding the top 2 object when $\mathcal{F} = min(x, p_c, p_l)$ and $\mathcal{H} = (p_c, p_l)$ over dataset 1 (Figure 1). While we show the ceiling queue $\mathcal{Q}$ as a sorted list, full sorting is not necessary for a priority queue. After initialized from the sorted output of x, we simply keep on probing the top incomplete object in $\mathcal{Q}$, resulting in the probes pr(a, p_c), pr(a, p_l), pr(b, p_c), and pr(b, p_l). Each probe will update the ceiling score of the object, and thus changing its priority in ceiling queue. Note that Figure 4 marks object with an underline (*e.g.*, $\underline{a{:}0.75}$) when it is fully probed, at which point its ceiling score is actually the final score. Finally, we can stop when $k = 2$ objects (in this case, a and b together) have completed and surfaced to the top.

It is straightforward to show that Algorithm *MPro* is both correct and optimal, as we state in Theorem 2. First, it will correctly return the *top-k* answers. *MPro* stops when all the k objects with highest ceiling scores are all complete (as they surface to $\mathcal{K}$). This stop condition ensures that all these k answers have final scores higher than the ceiling score of any object u still in $\mathcal{Q}$. Thus, any such u, complete or not, cannot outperform the returned answers, even with more probes, which implies the correctness. Second, Algorithm *MPro* is probe-optimal. Note that *MPro* always selects the top ceiling-scored incomplete object to probe. Theorem 1 asserts that every such probe is necessary before the stop condition $\mathcal{SC}$ becomes *True* (and thus *MPro* halts). It follows from Lemma 1 that *MPro* is probe-optimal, because it only performs necessary probes.

Theorem 2 (Correctness and Optimality of *MPro*): Given scoring function $\mathcal{F}$ and retrieval size k, Algorithm *MPro* will correctly return the *top-k* answers. With respect to the given schedule $\mathcal{H}$, *MPro* is also probe-optimal. ∎

4.3 Online Sampling-based Scheduling

Algorithm *MPro* assumes a given schedule $\mathcal{H}$, with respect to which the probe cost is guaranteed optimal. Given probe predicates $p_1, \ldots, p_n$, there are $n!$ possible schedules (each as a permutation

OID	x	p_c	p_l	$\mathcal{F}(x, p_c, p_l)$
a	0.8	0.9	0.2	0.2
b	0.7	0.8	0.2	0.2
c	0.6	0.6	0.3	0.3

Figure 5: Dataset 2.

of p_i). Different schedules can incur different probe cost, as Example 2 below shows. This section discusses the problem of identifying the best schedule that minimizes the probe cost and proposes an effective algorithm that provides such schedule to *MPro*.

Example 2: Consider $\mathcal{F} = min(x, p_c, p_l)$, using dataset 2 (Figure 5). For probe predicates $\{p_c, p_l\}$, two schedules $\mathcal{H}_1 = (p_c, p_l)$ and $\mathcal{H}_2 = (p_l, p_c)$ are possible. To find the top answer, when given $\mathcal{H}_1$ and $\mathcal{H}_2$, *MPro* will perform 6 and 4 probes respectively (as shown below); $\mathcal{H}_2$ is thus a better schedule (with 33% less probes).

$\mathcal{H}_1$: $\mathrm{pr}(a, p_c)$, $\mathrm{pr}(a, p_l)$, $\mathrm{pr}(b, p_c)$, $\mathrm{pr}(b, p_l)$, $\mathrm{pr}(c, p_c)$, $\mathrm{pr}(c, p_l)$
$\mathcal{H}_2$: $\mathrm{pr}(a, p_l)$, $\mathrm{pr}(b, p_l)$, $\mathrm{pr}(c, p_l)$, $\mathrm{pr}(c, p_c)$ ■

As Section 4.1 discussed, Algorithm *MPro* can generally work with either global or object-specific schedules. Our framework chooses to focus on global scheduling. Note that scheduling can be expensive (it is NP-hard in the number of predicates, as we show in [11]). As we will see, our approach is essentially *global, predicate-by-predicate* scheduling, using sampling to acquire predicate selectivities (and costs) for constructing a global schedule online at query startup. Note such *online* scheduling will add certain overhead to query run time. Per-object scheduling will thus incur N-fold scheduling cost for a database of N objects (which may far offset its benefit); in addition, it may complicate the algorithm and potentially interfere with parallelism (Section 5.3).

As Section 3.2 discussed, the probe cost of Algorithm *MPro* can be written as $\mathcal{PC}(MPro) = \sum_{i=1}^{n} N_i \cdot C_i$. Note that N_i (the number of necessary-probes for p_i) depends on the specific schedule $\mathcal{H}$ (*e.g.*, in Example 2, for $\mathcal{H}_2$, $N_{p_l} = 3$ and $N_{p_c} = 1$). To find the optimal schedule, we must further quantify how $\mathcal{H}$ determines N_i. In particular, when will an object u necessarily probe p_i under $\mathcal{H}$? At various stages of $\mathcal{H}$, we denote $\mathcal{H}_k$ for predicates evaluated up to the *k-prefix*, *i.e.*, $\mathcal{H}_k = \{x, p_1, \ldots, p_k\}$. Further, let θ be the lowest score of the *top-k* results (which we will not know *a priori* until the query is fully evaluated). According to Theorem 1, after probing $\mathcal{H}_{i-1} = \{x, p_1, \ldots, p_{i-1}\}$, object u will continue to probe p_i if $\overline{\mathcal{F}}_{\mathcal{H}_{i-1}}[u]$ is among the current *top-k* scores. Observe that $\overline{\mathcal{F}}_{\mathcal{H}_{i-1}}[u]$ will *eventually* be on the *top-k* if $\overline{\mathcal{F}}_{\mathcal{H}_{i-1}}[u] \geq \theta$ (since eventually only the final answers will surface to and remain on the top). That is, *MPro* will only probe $\mathrm{pr}(u, p_i)$ when $\overline{\mathcal{F}}_{\mathcal{H}_{i-1}}[u] \geq \theta$. We can then determine N_i for p_i as the number of object u that satisfies $\overline{\mathcal{F}}_{\mathcal{H}_{i-1}}[u] \geq \theta$.

We thus define the *aggregate selectivity* $\mathcal{S}^{\theta}_{\mathcal{F}}(T)$ for a set of predicates T as the *ratio* of database objects u that "pass" $\overline{\mathcal{F}}_{T}[u] \geq \theta$ (and thus will continue to be probed beyond T). (This selectivity notion, unlike its Boolean-predicate counterpart, depends on the aggregate "filtering" effect of all the predicates evaluated.) Thus the necessary probes of p_i is $N_i = N \cdot \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{H}_{i-1})$, and

$$\mathcal{PC}(MPro) = \sum_{i=1}^{n} N \cdot \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{H}_{i-1}) \cdot C_i = N \cdot \sum_{i=1}^{n} \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{H}_{i-1}) \cdot C_i.$$

Our goal is to find a schedule that minimizes $\mathcal{PC}(MPro)$. However, as our extended report [11] shows, this optimal scheduling problem is NP-hard and thus generally intractable. Since an exhaustive method may be too expensive and impractical, we propose a *greedy* algorithm that always picks the most "cost-effective" predicate with a low aggregate selectivity (thus high filtering rate) and a low cost. We thus use the intuitive *rank* metric (as similarly used in [2] using single-predicate selectivity) to represent the cost-effectiveness of *executing* p_i *after some* T *predicates* as below (note the rank depends on the "context" T). Our scheduler thus greedily selects the highest-ranked predicate to incrementally build a schedule.

$$rank(p_i \mid T) = \frac{1 - \mathcal{S}^{\theta}_{\mathcal{F}}(T \cup \{p_i\})}{C_i}.$$

However, how can we determine the selectivity with respect to a *top-k* threshold θ? (The cost C_i can be provided by users or measured by performing some sample probes of p_i.) Although preconstructed statistics are often used for query optimization, such requirement is unlikely to be realistic in our context, because predicates are either "dynamic" or "external" (Section 1). Our scheduling thus performs online sampling to estimate selectivities. The scheduler will sample a *small* number of objects and perform complete probing for their scores. While such sampling may add "unnecessary" probes, finding a good schedule can well justify this overhead (Section 6). In fact, some of the probes will later be necessary anyway (*MPro* can easily reuse those sampling probes).

Using the samples, we can estimate the selectivities with respect to the *top-k* threshold θ. The uniform sampling will select some k' *top-k* objects proportional to the sample size n, *i.e.*, $k' = \lceil k \cdot \frac{n}{N} \rceil$. That is, the sampling transforms a *top-k* query on the database into a *top-k'* query on the samples. Thus θ can be estimated as the lowest $\mathcal{F}$ score of the *top-k'* sampled objects.

To illustrate, suppose sampling results in the samples in Figure 5 for $\mathcal{F} = min(x, p_c, p_l)$. Assume that $k' = 1$ for the sample size, and thus θ= 0.3 (the *top*-1 $\mathcal{F}$ score); let the relative costs $C_{p_c} = 1$ and $C_{p_l} = 3$. To schedule p_c and p_l after $\{x\}$, we compute their ranks. Since all sampled objects satisfy $\overline{\mathcal{F}}_{\{x,p_c\}} \geq 0.3$, it follows that $\mathcal{S}^{0.3}_{\mathcal{F}}(\{x, p_c\}) = \frac{3}{3} = 1$, and similarly $\mathcal{S}^{0.3}_{\mathcal{F}}(\{x, p_l\}) = \frac{1}{3}$. Consequently, since $rank(p_c \mid \{x\}) = \frac{1-1}{1} = 0$ and similarly $rank(p_l \mid \{x\}) = \frac{2}{9}$, the scheduler will select p_l before p_c (resulting in $\mathcal{H}_2 = (p_l, p_c)$ as in Example 2).

Our scheduler thus performs sampling-based online scheduling, by continuing such greedy scheduling (as just showed) to construct a complete schedule $\mathcal{H}$ for *MPro*. Note that it is also possible to activate the same scheduler to *reschedule* after *MPro* performs more probes and thus acquire more accurate statistics (of selectivities and costs). Our study (Section 6) shows that the simple scheme of scheduling *once* at query startup with a small sample size (*e.g.*, 0.1%) works very well and the net overhead is negligible.

5. EXTENSIONS AND SCALABILITY

Based on the basic algorithm *MPro*, we next discuss several useful extensions. First, Section 5.1 discusses *iMPro* for supporting incremental access by the *next* interface. While we assume selection predicates so far, Section 5.2 generalizes to handling joins as well. Section 5.3 then shows that *MPro* can be easily parallelized to exploit available resources with linear speedup. Finally, Section 5.4 analytically develops the scalability of *MPro*, showing that its cost growth is sub-linear in database size. (In addition, in [11] we also extend *MPro* for approximate queries.)

5.1 Incremental and Threshold Interfaces

Incremental access can be essential for ranked queries, as users often want to sift through top answers until satisfied. We can immediately extend the *top-k* interface of Algorithm *MPro* to support incremental access by the *next* interface (or more generally *next-k*). In this mode, the system can continue at where it left off, without

starting from scratch. This incremental extension, referred to as Algorithm *iMPro*, is essentially the same as *MPro* in Figure 3, except now the ceiling queue is "persistent" during incremental access. Thus *iMPro* will initialize $\mathcal{Q}$ only at the very first *next* call. For any subsequent call, *iMPro* will continue to populate $\mathcal{K}$ with the next top answer just like *MPro*.

Further, it is also desirable to support *threshold* queries, where users specify a threshold θ to retrieve objects u such that $\mathcal{F}[u] \geq \theta$. We can support this interface simply by extending *iMPro* to output incrementally until all objects that score above θ are returned.

5.2 Fuzzy Joins

Join predicates are inherently expensive, as they generally require a probe for each combination of objects from participating relations. Having studied Algorithm *MPro* for selection predicates over a single table, we show that essentially the same algorithm can handle join predicates over multiple tables as well. We thus have a unified framework for both selection and join predicates, under the abstraction of expensive predicates.

Intuitively, to unify both selections and joins, we consider them as operations over the "entire" input of the query. When a query involves multiple relations, we consider the Cartesian product of all the relations as the input. With this conceptual modeling, all predicates are simply selections over the Cartesian table. Thus, at least conceptually, Algorithm *MPro* can be applied for all expensive predicates- selections or joins alike.

To illustrate this conceptual unification, consider Query 2 (Section 1), which involves two relations `house` and `park`. The query uses a join predicate $p_j = close(h.zip,\ s.address)$ over pairs of h from `house` and s from `park`. For instance, suppose the relations, as sets of objects, are `house` = $\{a,\ b\}$ and `park` = $\{e,\ f\}$. The Cartesian product is thus $\{\langle a,e\rangle,\ \langle a,f\rangle,\ \langle b,e\rangle,\ \langle b,f\rangle\}$. Note we use $\langle d_1, \cdots, d_m\rangle$ to denote a *Cartesian object* that joins object d_i from relation $\mathtt{r_i}$, and we refer to d_i as the $\mathtt{r_i}$ *dimension*; *e.g.*, $\langle a,e\rangle$ joins a and e as the `house` and `park` dimensions respectively. As example data, Figure 6 shows the predicate scores for each Cartesian object. For instance, since x is a selection over `house` (and similarly p_r over `park`), it only *depends on* the corresponding dimension; *e.g.*, $x[\langle a,e\rangle] = x[\langle a,f\rangle] = 0.9$. In contrast, as a join predicate over both relations, p_j depends on both dimensions. With all Cartesian objects fully materialized, Algorithm *MPro* can directly apply as if all predicates are selections.

However, while conceptually operating on Cartesian objects, Algorithm *MPro* can incrementally materialize them. To incorporate such *lazy materialization*, we use a wildcard "$*$" for an unmaterialized dimension in a Cartesian object. For instance, in Figure 7, $u = \langle a,*\rangle$ has the first but not the second dimension materialized. We then *explode* unmaterialized dimensions on demand. Recall that *MPro* will need an object u only when it surfaces to the top of the ceiling queue $\mathcal{Q}$ (Figure 3)– *i.e.*, u has the highest ceiling score, and thus $\mathsf{pr}(u, p)$ is necessary (by Theorem 1) for the next predicate p. *MPro* can thus wait until this point to materialize the required dimensions (if not yet) for p. Consider $u = \langle a,*\rangle$ and assume p_r is to be probed next, which depends on the unmaterialized `park` dimension. Since the wildcard represents `park` = $\{e,\ f\}$, *MPro* will *explode* u on this newly needed dimension to materialize u into $\{\langle a,e\rangle,\ \langle a,f\rangle\}$.

Figure 7 illustrates Algorithm *MPro* for evaluating Query 2, with search predicate x and expensive predicates when $\mathcal{H}=(p_r, p_j)$. Algorithm *MPro* begins by initializing $\mathcal{Q}$ with the x output stream (of `house` objects), leaving the `park` dimension unmaterialized. *MPro* then explodes the top object $u = \langle a,*\rangle$ for p_r, in order to perform necessary probe $\mathsf{pr}(u, p_r)$. The resulting objects $\langle a,e\rangle$ and $\langle a,f\rangle$ will be inserted back to (the top of) $\mathcal{Q}$. Since they share the same top ceiling score (as u), *MPro* will order them with the deterministic tie-breaker as Section 3 discussed. Suppose that $\langle a,e\rangle$ becomes the new top object; $\mathsf{pr}(\langle a,e\rangle, p_r)$ will then be the next necessary probe. Algorithm *MPro* will continue as usual and eventually output $\langle a,f\rangle$:0.80 (for `house` a and `park` f) as the *top*-1 answer.

OID	x	p_r	p_j	$\mathcal{F}(x, p_r, p_j)$
$\langle a,e\rangle$	0.90	0.50	0.70	0.50
$\langle a,f\rangle$	0.90	0.80	0.80	0.80
$\langle b,e\rangle$	0.70	0.50	0.90	0.50
$\langle b,f\rangle$	0.70	0.80	0.30	0.30

Figure 6: Dataset 3 for join query $\mathcal{F}= min(x, p_r, p_j)$.

step	action	ceiling queue $\mathcal{Q}$	output($\mathcal{K}$)
1.	initialize $\mathcal{Q}$ and $\mathcal{K}$	$\langle a,*\rangle$:0.90, $\langle b,*\rangle$:0.70	{} (empty)
2.	explode $\langle a,*\rangle$ for p_r into $\langle a,e\rangle$, $\langle a,f\rangle$	$\langle a,e\rangle$:0.90, $\langle a,f\rangle$:0.90, $\langle b,*\rangle$:0.70	{}
3.	probe $\mathsf{pr}(\langle a,e\rangle, p_r)$	$\langle a,f\rangle$:0.90, $\langle b,*\rangle$:0.70, $\langle a,e\rangle$:0.50	{}
4.	probe $\mathsf{pr}(\langle a,f\rangle, p_r)$	$\langle a,f\rangle$:0.80, $\langle b,*\rangle$:0.70, $\langle a,e\rangle$:0.50	{}
5.	probe $\mathsf{pr}(\langle a,f\rangle, p_j)$	$\langle a,f\rangle$:0.80, $\langle b,*\rangle$:0.70, $\langle a,e\rangle$:0.50	{}
6.	pop top complete objects from $\mathcal{Q}$ into $\mathcal{K}$	$\langle b,*\rangle$:0.70, $\langle a,e\rangle$:0.50	$\langle a,f\rangle$:0.80
7.	stop condition holds output $\mathcal{K}$	$\langle b,*\rangle$:0.70, $\langle a,e\rangle$:0.50	$\langle a,f\rangle$:0.80

Figure 7: Illustration of Algorithm *MPro* for a join query.

5.3 Parallel Processing

We discussed sequential *MPro* which performs necessary probes one after another. It may appear that such sequential probing (as ordered by the ceiling queue $\mathcal{Q}$) cannot allow parallelization. To the contrary, we can extend *MPro* to execute multiple probes concurrently or to process multiple chunks of data independently.

Probe-Parallel *MPro*: Given a *top-k* query, Section 4.1 observed that there are generally multiple necessary probes at any time during query processing. For Algorithm *MPro*, as Theorem 1 implies, every incomplete object among the current *top-k* (highest ceiling-scored in $\mathcal{K}$ and $\mathcal{Q}$) needs further probing. Thus the number of necessary probes will be k initially when all the objects are incomplete and decrease progressively to 0 until all *top-k* are completed.

Note that necessary probes must be performed sooner or later, as they are required independent of other probes (Definition 1)– We can thus parallelize *MPro* to execute many necessary probes concurrently to speedup performance while still maintaining probe minimality. That is, if the predicate subsystems (for evaluating probes) support concurrent probes (such as a multi-threaded local subsystem or Web server), the *probe-parallel MPro* can perform some or all the necessary probes (depending on the available concurrency level), update $\mathcal{Q}$ with all such probes, and continue to perform next batch of necessary probes. For instance, consider parallelizing our example in Figure 4 when $k = 2$. At line 1, *MPro* will execute both $\mathsf{pr}(a, p_c)$ and $\mathsf{pr}(b, p_c)$. Updating $\mathcal{Q}$ accordingly, *MPro* will still find a and b as the *top*-2 incomplete objects, and thus perform $\mathsf{pr}(a, p_l)$ and $\mathsf{pr}(b, p_l)$. *MPro* will then output a and b as the *top*-2 answers. Note that this probe-parallel algorithm performs the same set of necessary probes (as in Figure 4), although in a different order. When the probe time dominates, paralleling k probes concurrently will result in a k-fold speedup.

Data-Parallel *MPro*: Alternatively, we can also parallelize *MPro* by partitioning the database and processing all chunks concurrently. As Figure 8 shows, Algorithm *dpMPro* consists of two main steps: First, in *data distribution*, *dpMPro* will uniformly distribute the input database $\mathcal{D}$ into s *data chunks*, each of which is of size $1/s$ of

Algorithm *dpMPro*($\mathcal{F}, k, \mathcal{H}, \mathcal{D}, s$): Data-parallel *MPro*
Input: $\mathcal{F}$, k, $\mathcal{H}$, $\mathcal{D}$: same as Algorithm *MPro*.
• s: chunking factor, *i.e.*, number of data chunks
Output: $\mathcal{K}$, the *top-k* answers with respect to $\mathcal{F}$.
Procedure:
(1) **Data Distribution**: distribute objects in $\mathcal{D}$ uniformly into s chunks $\mathcal{D}_1, \ldots, \mathcal{D}_s$
(2) **Incremental Merging**:
• $\mathcal{K} \leftarrow \{\}$; $\mathcal{M} \leftarrow \{\}$
// $\mathcal{K}$: output; $\mathcal{M}$: merging queue which buffers the top object of every chunk.
• let I_i be the iterator for *iMPro*($\mathcal{F}, \mathcal{H}, \mathcal{D}_i$) // access $\mathcal{D}_i$ incrementally.
• for $i = 1$ to s in parallel: // initialize $\mathcal{M}$ with top objects.
 - $top_i \leftarrow I_i.next()$; $\mathcal{M}.insert(top_i, top_i.score)$
• while ($|\mathcal{K}| < k$): // until have generated enough top answers.
 - $u \leftarrow \mathcal{M}.top()$ // u outperforms any other objects still in $\mathcal{M}$.
 - if ($\forall top_i : u.score \geq top_i.score$):
 // implies that u indeed outperforms any objects in all $\mathcal{D}_i$.
  - append u to $\mathcal{K}$
 - else: // unseen objects may be better; bring new top objects into $\mathcal{M}$.
  - for $i = 1$ to s in parallel:
   - $top_i \leftarrow I_i.next()$; $\mathcal{M}.insert(top_i, top_i.score)$
(3) **Top-k Output**: return in order each (u:$u.score$) in $\mathcal{K}$

Figure 8: Algorithm *dpMPro*.

$\mathcal{D}$, for a given *chunking factor* s. (Of course, this partitioning can be done off-line as it is query independent.) To maximize work distribution (or concurrency), we wish that top answers will uniformly come from different chunks. That is, the data distributor must ensure that the chunks be as "similar" to each other as possible, by identifying and distributing similar objects (that will perform similarly in queries) in $\mathcal{D}$ to different chunks.

Second, during *incremental merging*, *dpMPro* will access and merge top answers from each $\mathcal{D}_i$ provided by an incremental *MPro* iterator I_i (as Section 5.1 discussed). *dpMPro* uses a *merging queue* $\mathcal{M}$ to sort top objects from all I_i by their query scores. If u is the top of $\mathcal{M}$, it has outperformed those still in $\mathcal{M}$. *dpMPro* checks if u also outperforms all unseen objects by comparing it to the last top scores of every I_i. (These top scores may have been output to $\mathcal{K}$ and not present in $\mathcal{M}$ any more.) If so, u must be the overall top (of entire $\mathcal{D}$). Otherwise, some unseen objects may be better, and *dpMPro* will request parallel access to load new top objects from all $\mathcal{D}_i$. Note that parallel loading will bring in more objects to $\mathcal{M}$ in just one access time, saving some accesses that might be needed later.

Observe that *dpMPro* speeds up by distributing work to s parallel *MPro* "processors": Consider a *top-sk* query over a database of size sN, denoted $\mathcal{DB}(sN)$. First, as the database is chunked uniformly, the answers should distribute uniformly among the s chunks– thus each *MPro* processor only needs to find (around) k instead of sk answers. Further, the chunking reduces the database from $\mathcal{DB}(sN)$ to $\mathcal{DB}(N)$ for each processor. Putting together, *dpMPro* parallelizes the time of finding *top-sk* over $\mathcal{DB}(sN)$ into that of *top-k* over $\mathcal{DB}(N)$, reducing both the database and retrieval sizes by s times for each processor. As Section 5.4 will quantify, this reduction results in an overall s-times speedup.

5.4 Minimal-Probe Scalability

Since necessary probes (Definition 1) are algorithm-independent cost for expensive predicates, we want to understand how much this required cost will be, and how it scales for larger databases. Note that Algorithm *MPro*, as a probe-optimal algorithm (Lemma 1), executes exactly only necessary probes. Therefore, *MPro* can be an effective vehicle for understanding minimal-probing costs. To this end, Section 6 will experimentally evaluate minimal-probing cost with respect to different queries (*i.e.*, k and $\mathcal{F}$) and databases (*i.e.*, how objects score under $\mathcal{F}$). This section seeks to understand *analytically* how the minimal-cost scales, by addressing an important question: Given a *top-k* query $\mathcal{F}(x, p_1, \ldots, p_n)$, if it requires $\mathcal{P}_i$ probes for each predicate p_i over a database of size N, or $\mathcal{DB}(N)$, how will the required necessary-probes increase when the database is scaled up s times, *i.e.*, $\mathcal{DB}(sN)$. As explained, we specifically study the scalability of *MPro* to generally answer this question.

To focus on scalability, we assume *uniform scaling*, such that $\mathcal{DB}(sN)$ will perform "statistically" similar to $\mathcal{DB}(N)$– *i.e.*, the two databases differ in *size* but not *nature*, so that we can isolate data size to study its effect. To allow analytical study, we approximate uniform scaling by simply replicating $\mathcal{DB}(N)$ s times to generate $\mathcal{DB}(sN)$. We thus obtain an interesting result: If a database is uniformly scaled up s times, *MPro* can retrieve s times more top answers, with s times more probe cost. While we leave a formal proof to [11], Section 6 experimentally verifies this result– over datasets that are only similar but not identical.

Theorem 3 (Probe Scalability): Consider a ranked query $\mathcal{F}(x, p_1, \ldots, p_n)$ and a schedule $\mathcal{H}$. Suppose that $\mathcal{DB}(N)$ is a database of size N; let $\mathcal{DB}(sN)$ be the database containing the sN objects generated by replicating $\mathcal{DB}(N)$ s times. $\mathcal{PC}(MPro(\mathcal{F}, sk, \mathcal{H}, \mathcal{DB}(sN))) = s \cdot \mathcal{PC}(MPro(\mathcal{F}, k, \mathcal{H}, \mathcal{DB}(N)))$. ∎

Theorem 3 gives the scalability of necessary-probes in general as well as Algorithm *MPro* in particular. This result enables several interesting observations. First, it shows that *MPro* has good *data scalability*, in which cost increase is sublinear in database size: The probes required for finding k answers from $\mathcal{DB}(sN)$ will be *less than* s times that for finding the same number of answers from $\mathcal{DB}(N)$: As the stop condition ($\mathcal{SC}$ in Figure 3) ensures, Algorithm *MPro* will stop earlier and pay less probe cost for a smaller retrieval size, and thus $\mathcal{PC}(MPro(\mathcal{F}, k, \mathcal{H}, \mathcal{DB}(sN))) < \mathcal{PC}(MPro(\mathcal{F}, sk, \mathcal{H}, \mathcal{DB}(sN)))$. (Section 6 shows how necessary-probes increase over retrieval sizes.) With Theorem 3, we derive the sublinear growth of cost: $\mathcal{PC}(MPro(\mathcal{F}, k, \mathcal{H}, \mathcal{DB}(sN))) < s \cdot \mathcal{PC}(MPro(\mathcal{F}, k, \mathcal{H}, \mathcal{DB}(N)))$. Since complete probing (of the standard sort-merge framework) requires linear increase, *MPro* will scale better and reduce more probes for larger databases.

Second, *MPro* will enjoy good *resource utility*: If the computing resource scales up s times, our framework can receive linear speedup by using s concurrent *MPro* processors in parallel in *dpMPro*. As Section 5.3 discusses, by reducing both the retrieval and database size for each processor, *dpMPro* can reduce the probe cost for each processor from $\mathcal{PC}(MPro(\mathcal{F}, sk, \mathcal{H}, \mathcal{DB}(sN)))$ to $\mathcal{PC}(MPro(\mathcal{F}, k, \mathcal{H}, \mathcal{DB}(N)))$. As Theorem 3 asserts, the latter is $\frac{1}{s}$ of the former, which means a s-times speedup or linear to the resource increase.

Finally, it is important to note that these observations together indicate that our framework can take advantage of increasing computing resource to better scale to larger databases: If the database scales up s times (resulting in sublinear cost-growth), *dpMPro* with s-times resource (resulting in linear speedup) can fully offset the potential slowdown to achieve even faster processing.

6. EXPERIMENTS

This section reports our extensive experiments for studying the effectiveness and practicality of Algorithm *MPro*. Our experiments in fact have a two-fold interpretation. On one hand, the results specifically quantify the practical performance of Algorithm *MPro*. On the other hand, since *MPro* is provably probe-optimal, this empirical study also generally contributes to understanding the "lower bound" for supporting probe predicates. In fact, as Section 3.3 explains, we adopt the *SortMerge* scheme as our baseline for comparison – which requires complete probing and thus symmetrically defines the "upper bound."

```
Q1: select id from house
    where nearcity(zip, Chicago) x,
      roomy(bedroom) p1, cheap(price) p2, large(size) p3
    order by min(x, p1, p2, p3) stop after k
Q2: select h.id, u.name from house h, park s
    where nearcity(h.zip, Chicago) x,
      roomy(bedroom) p1, cheap(price) p2, close(h.zip, s.zip) p3
    order by min(x, p1, p2, p3) stop after k
Q3: select id from house
    where nearcity(zip, Chicago) x,
      roomy(bedroom) p1, cheap(price) p2, safe(zip) p3
    order by min(x, p1, p2, p3) stop after k
```

```
User-defined selections:
large(size):
  if (size<1500 or size>3500): return 0
  else: return (size-1500)/2000
Fuzzy joins:
close(zip1, zip2):
  // dist is a builtin function for computing distance.
  d = dist(zip1, zip2)
  if d>20: return 0
  else: return (20-d)/20
External predicates:
safe(zip):
  crime = query_apb(zip) // query apbnews.com.
  return 1-(crime-1)/9.0
```

Figure 9: Benchmark queries and the expensive predicates.

We measured two different performance metrics. Consider query $Q = \mathcal{F}(x, p_1, \ldots, p_n)$. First, to quantify the relative probe performance, we measure how *MPro* saves unnecessary probes with the *probe ratio* metric. Suppose that *MPro* performs N_i probes for each p_i. In contrast, *SortMerge* will require N probes (where N is the database size) for every probe predicate. The overall probe ratio (in %) is thus $pratio(Q) = \frac{\sum_{i=1}^{n} N_i}{nN}$. In addition, to understand the saving of each predicate, we also measured the predicate probe ratio as $pratio(p_i) = \frac{N_i}{nN}$. Note that $pratio(Q) = pratio(p_1) + \cdots + pratio(p_n)$.

Further, to quantify the "absolute" performance, our second metric measured the *elapsed time* (in seconds) as the total time to process the query. This metric helps us to gauge how the framework can be practically applied with reasonable response time. It also more realistically quantifies the performance when predicates are of different costs, because we measure the actual time, which cannot be shown by counting the number of probes as in probe ratios.

Our experiments used both a "benchmark" database (with real-life data) as well as synthetic data. To understand the performance of *MPro* in real-world scenarios, Section 6.1 experiments with a benchmark real-estate database (essentially the same as our examples). To extensively isolate and study various parameters (*e.g.*, database size and scoring functions), Section 6.2 reports "simulations" over data synthesized with standard distributions.

Our experience found it straightforward to implement Algorithm *MPro*. As Figure 3 shows, *MPro* essentially operates on a priority queue (the ceiling queue $\mathcal{Q}$), which we use a standard heap implementation. We build the queue on top of a DBMS, namely PostgreSQL 7.1.3, to take care of data storage. Our implementation adopts the Python programming language for defining expensive predicates, since it is interpreted and open-source. In principle, any languages or, more preferably, graphic user interfaces can be used to define user functions. Finally, all experiments were conducted with a Pentium III 933Mhz PC with 256MB RAM.

6.1 Benchmarks over Real-Life Database

We first report our experiments of benchmark queries accessing a real-estate system (as Example 1 introduced). To establish a practical scenario, we extracted real data from `realtor.com` (an authoritative real-estate source). In particular, we collected (by querying) all the for-sale houses in Illinois, resulting in $N = 20990$ objects for relation `house`, each with attributes *id*, *price*, *size*, *bedroom*, *bath*, *zip*, and *city*. In addition, we constructed a second relation `park` of about 110 Illinois state parks, each with attributes *name* and *zip*.

Our experiments considered a benchmark scenario of finding top houses around Chicago for a small family of four members. We thus created three benchmark queries, as Figure 9 (upper) shows. The queries use a search predicate $nearcity(zip, C)$ which returns houses closest to C (*e.g.*, C = `Chicago`) in order. All the others (*roomy*, *cheap*, *large*, *close*, and *safe*) are probe predicates; Figure 9 (lower) also shows some of them. Note that the three queries only differ in predicate p_3. In particular, *large*, *close*, and *safe* represent increasingly more complex and expensive operations– *i.e.*, simple user-defined functions, joins, and external predicates.

For each query, we measured the probe ratios and the elapsed time. Figures 10(a)–(c) plot the probe ratios (the y-axis) with respect to different retrieval size k (the x-axis), both logarithmically scaled. Each graph shows four curves for $pratio(p_1)$, $pratio(p_2)$, and $pratio(p_3)$, and their sum as the overall probe ratio. For instance, for Q_1 in Figure 10(a), when $k = 10$ (to retrieve *top*-10), p_1 requires 3% probes, p_2 0.4%, and p_3 0.1%, which sum up to 3.5%. In other words, since $N = 20990$ and $n = 3$, the ratios translate to 1889, 252, and 63 probes (with a total of 2204) out of the $nN = 62970$ complete probes. Observe that the vast majority of complete probes are simply unnecessary and wasted– in this case 96.5% or 60766 probes. As also expected, the probe ratio (relative cost) is smaller for smaller k; *e.g.*, for $k = 1$, the overall probe ratio is only 1.4%. As Section 3 discussed, such "proportional cost" is crucial for *top-k* queries, as users are typically interested in only a small number of top answers. In fact, Figure 10(b)–(c) only show the top range $k \leq 0.1\% \cdot N$ to stress this critical range of interest.

Figures 10(d)–(f) compare the elapsed times (for gauging the "absolute" performance) of the baseline *SortMerge* scheme and *MPro*. Note that we implemented *SortMerge* by completely probing all objects and at the same time computing the $\mathcal{F}$ scores to create a functional index [22]. (Thus the comparison in fact favors *SortMerge*, whose separate "merge" phase combining multiple predicate streams was not included.) The *SortMerge* cost thus consists of the startup probing and incremental index access costs. (The latter is not observable in Q_2 and Q_3 as the probing costs dominate.) Referring to Figure 10(f), when $k = 10$, *MPro* responded in 408 seconds, while *SortMerge* takes 21009 seconds. That is, we observe that, when probe predicates are truly expensive as in Q_3 (with external predicate *safe*), our framework can be orders of magnitude faster than the standard *SortMerge* scheme. Such speedup can potentially turn an overnight query into an interactive one. The fuzzy join of Q_2 demonstrates similar speedup, from 1368 seconds to 26 seconds or 1.9% time (for $k = 10$).

Finally, we note that our scheduler (Section 4.3) effectively identified the best schedule with 0.1% sampling, which corresponds to less than a second overhead at query startup. We will report more extensive scheduling results later with simulations.

6.2 Simulations over Synthetic Datasets

We next perform extensive experiments to study various performance factors. To isolate and control different parameters, our experiments in this section used synthetic datasets. Our configurations are characterized by the following parameters. (1) Database size N: 1k, 10k, and 100k. (2) Score distribution D: the distri-

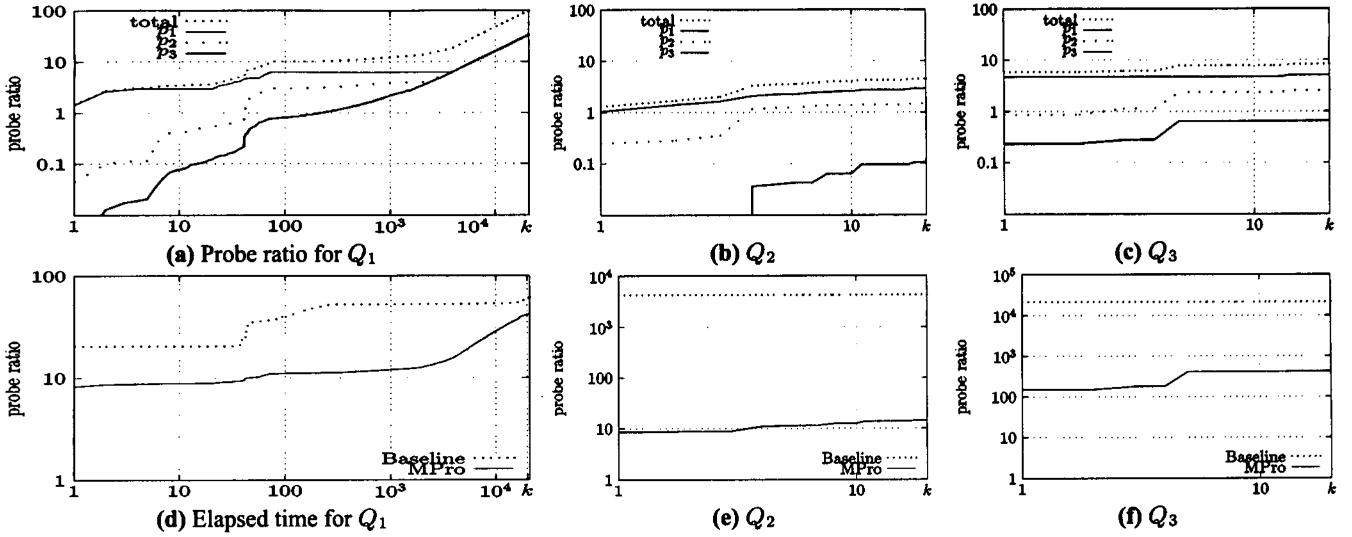


Figure 10: Results for benchmark queries.

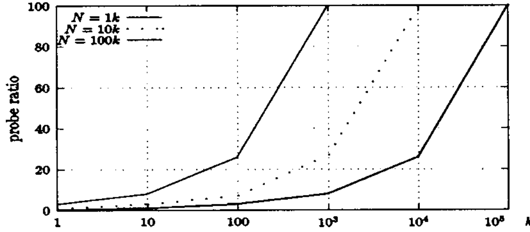


Figure 11: Different database sizes N with (D=norm, $\mathcal{F}$=min).

butions of individual predicate scores, including the standard unif (uniform) and norm (normal) distributions as well as funif, a home-brew "filtered-uniform" distribution (see below). (3) Scoring function $\mathcal{F}$: min, avg (average), and $gavg$ (geometric average).

We isolate each parameter to study its impact on *MPro*. All queries are of the form $\mathcal{F}(x, p_1, p_2, p_3)$ (as in Section 6.1). We conclude with quantifying the effectiveness of our scheduling algorithm (Section 4.3).

N: Database Size. Figure 11 presents the performance evaluation for N = 1k, 10k, and 100k. when $\mathcal{F}$=min and D=norm. It is interesting to observe that, the probe ratios are approximately the same if the *relative* retrieval size is the same, regardless of the database size. For instance, for $k = N \cdot 1\%$ (*i.e.*, k = 10, 100, and 1000 for N = 1k, 10k, and 100k respectively), the probe-ratios are all about 8%. Similarly, the probe-ratios for $k = N \cdot 0.1\%$ are about 3%. This observation is critical in evaluating the scalability of *MPro* over N: As Section 3 explained, it is presumable that the retrieval size will be independent of the database size N; *i.e.*, users will probably interested in only the very top hits. *MPro* will thus be *relatively* (compared to the baseline scheme) more efficient for larger databases, which shows its scalability over N.

Note that this "constant-ratio" observation also verifies Theorem 3. For (k = 10, N = 1k), the probe cost is $8\% \cdot n \cdot N$ (where n is the number of predicates). Now let s = 10; we observed that (sk = 100, sN = 10k) costs $8\% \cdot n \cdot sN$, *i.e.*, s times more. That is, when the database is scaled up s times, *MPro* retrieves s times more top answers, with s times more probe cost.

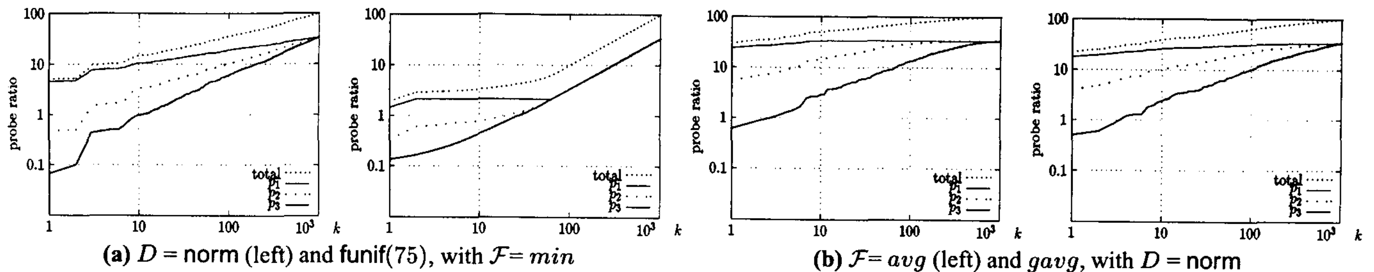
D: Score Distribution. Figure 12(a) present the results with different score distributions, which characterize predicates. The left figure presents the results for normal distribution (with mean 0.5 and variance 0.16), which show similar proportional-cost behavior over k (as in the benchmark queries). The second distribution funif simulates "filtering" predicates. As we observed in our benchmark queries, real-life predicates are likely to "filter out" a certain portion of data; *e.g.*, the *large* predicate (Figure 9) disqualifies 78% objects with zero scores (as their *sizes* are out of the desired range). We define funif(f) (for *filtered uniform*) to simulate such predicates, where f% objects score 0 and the rest are uniformly distributed. The right figure (Figure 12a) plots the cost for funif(75). Observe that such filtering makes *MPro* more effective– *MPro* can leverage the filtering to lower the ceiling scores of disqualified objects early and thus focus on the promising ones.

$\mathcal{F}$: Scoring Function. To understand the impact of scoring functions (which can form the basis of how a practical system may choose particular functions to support), we compare some common scoring functions: min (Figure 12a) and some representative average functions (Figure 12b), namely arithmetic average $avg : (x + p_1 + p_2 + p_3)/4$ and geometric average $gavg := (x \cdot p_1 \cdot p_2 \cdot p_3)^{1/4}$.

We found that min is naturally the least expensive, as it allows low scores to effectively decrease the ceiling scores and thus "filter" further probes. (In contrast, max will be the worst case, requiring complete probing.) The average functions perform similarly, with $\mathcal{F} = gavg$ being about 5% to 10% cheaper than avg.

Scheduling Effectiveness. We quantify the *average* scheduling performance over different predicate configurations (p_1, p_2, p_3) characterized by costs (C_1, C_2, C_3) and distributions (funif(f_1), funif(f_2), funif(f_3)). We randomly generated 100 configurations of (C_1, C_2, C_3) and (f_1, f_2, f_3), with C_i in [0:100] and f_i in [25:75]. For each schedule of a configuration, we measure the *cost ratio* (relative to complete probing), *i.e.*, $\frac{\sum_{i=1}^{n} C_i N_i}{\sum_{i=1}^{n} C_i N}$.

For k = 1, 10, and 100, our scheduler determines a schedule for each configuration, with 0.1% and 1% sampling (over N = 20k objects). Figure 13 compares the average cost ratios of our scheduler with those of the best and worst schedules (which were found by exhaustive enumeration). Note that the *net scheduling overhead* (the extra probe costs incurred by sampling) is show in dark at the top of the 0.1% and 1% bars. Observe that our scheduler generates the best schedules in most cases with a very small sample size of 0.1% (and thus the average probe cost closely approximates that of the best schedule). It is interesting to contrast with 1% sampling: First, while 1%, with more sampling, gives exactly the best schedule (as their light bars are of the same heights in Figure 13), its higher sampling overhead makes it overall less attractive than



(a) D = norm (left) and funif(75), with $\mathcal{F}= min$

(b) $\mathcal{F}= avg$ (left) and $gavg$, with D = norm

Figure 12: Different score distributions (a) and scoring functions (b) with N = 1k.

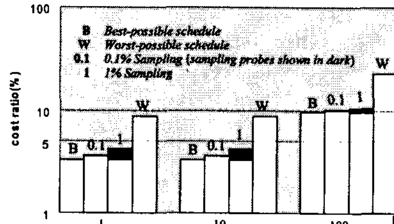


Figure 13: Scheduling for random configurations.

0.1%. Second, for larger k (*e.g.*, k = 100), the net overhead of 1% becomes less significant, because of the increasing probe reusing by *MPro*– thus larger sampling size can be justified by larger k.

7. CONCLUSION

We have presented our framework and algorithms for evaluating ranked queries with expensive probe predicate. We identified that supporting probe predicates are essential, in order to incorporate user-defined functions, external predicates, and fuzzy joins. Unlike the existing work which assumes only search predicates that provide sorted access, our work addresses generally supporting expensive predicates for ranked queries. In particular, we proposed Algorithm *MPro* which minimizes probe accesses as much as possible. As a basis, we developed the principle of necessary probes for determining if a probe is truly necessary in answering a *top-k* query. Our algorithm is thus provably optimal, based on the necessary-probe principle. Further, we show that *MPro* can scale well and can be easily parallelized (and it supports approximation [11]).

We have implemented the mechanism described in this paper, based on which we performed extensive experiments with both real-life databases and synthetic datasets. The results are very encouraging. Our experiments show that the probe cost of Algorithm *MPro* is desirably proportional to the retrieval size. It can eliminate as much as 97% probes for our benchmark queries when $k = 10$, which can be orders of magnitude faster than the standard scheme with complete probing. We believe that our framework can enable practical support for expensive predicates in ranked queries.

Acknowledgements: We thank Divyakant Agrawal and Wen-Syan Li for their fruitful discusions during one of the authors' summer visit at NEC USA CCRL, which inspired us to pursue this work.

8. REFERENCES

[1] R. Agrawal and E. Wimmers. A framework for expressing and combining preferences. *SIGMOD 2000*, pages 297–306, 2000.

[2] J. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. *SIGMOD 1993*, pages 267–276, 1993.

[3] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn. Optimizing disjunctive queries with expensive predicates. *SIGMOD 1994*, pages 336–347, 1994.

[4] S. Chaudhuri and L. Gravano. Optimizing Queries over Multimedia Repositories. *SIGMOD 1996*, pages 91–102, 1996.

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

[6] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD 2001*, 2001.

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

[8] E. Wimmers, L. Haas, M. Roth, and C. Braendli. Using Fagin's algorithm for merging ranked results in multimedia middleware. *International Conference on Cooperative Information Systems*, pages 267–278, 1999.

[9] S. Nepal and M. Ramakrishna. Query processing issues in image(multimedia) databases. *ICDE 1999*, pages 22–29, 1999.

[10] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. *PODS 2001*, 2001

[11] K. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. Technical Report UIUCDCS-R-2001-2258, University of Illinois, December 2001.

[12] A. Natsev, Y. Chang, J. Smith, C. Li, and J. Vitter. Supporting incremental join queries on ranked inputs. *VLDB 2001*, pages 281–290, 2001.

[13] R. Kimball and K. Strehlo. Why decision support fails and how to fix it. *SIGMOD Record*, 24(3):92–97, 1995.

[14] M. Carey and D. Kossmann. On saying "enough already!" in SQL. *SIGMOD 1997*, pages 219–230, 1997.

[15] M. Carey and D. Kossmann. Reducing the braking distance of an SQL query engine. *VLDB 1998*, pages 158–169, 1998.

[16] W. Cohen. Integration of heterogeneous databases without common domains using queries based on textual similarity. *SIGMOD 1998*, pages 201–212, 1998.

[17] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. *ICDE 2002*, 2002

[18] S. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1994.

[19] L. Zadeh. Fuzzy sets. *Information and Control*, 8:338–353, 1965.

[20] G. Salton. *Automatic Text Processing.* Addison-Wesley, 1989.

[21] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms.* MIT Press, 2001.

[22] The PostgreSQL Global Development Group. *The PostgreSQL 7.1. Reference Manual.*