Probe Minimization by Schedule Optimization: Supporting Top-K Queries with Expensive Predicates

Seung-won Hwang and Kevin Chen-Chuan Chang

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, ranked queries often accompany Boolean ranking conditions, which may turn predicates into expensive ones, as the index structure on the predicate built on the base table may be no longer effective in retrieving the filtered objects in order. Fourth, 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. To enable *probe minimization*, we develop the problem as cost-based optimization of searching over potential probe schedules. In particular, we decouple probe scheduling into object and predicate scheduling problems and develop an analytical object scheduling optimization and a dynamic predicate scheduling optimization, which combined together form a cost-effective probe schedule.

Index Terms—Database query processing, distributed information systems, database systems.

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 $\texttt{house}(id, price, size, zip, age)$ 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
order by $min(new(age), cheap(price, size), large(size))$
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.[1] For each object, each predicate maps the input attributes to a score in [0:1]. For example, a house a with $age = 2$ years, $price = \$150\text{k}$, and $size = 2,000$ sqft may score $new(2) = 0.9$, $cheap(150\text{k}, 2,000) = 0.85$, and $large(2,000) = 0.75$. 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 five 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 *nonindex* predicates: When a predicate is dynamically defined or externally accessed, a preconstructed 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

- *S.-w. Hwang is with the Department of Computer Science and Engineering, Pohang University of Science and Technology, San 31, Hyoja dong, Nam gu, Pohang 790-784, Korea. E-mail: swhwang@postech.ac.kr.*
- *K.C.-C. Chang is with the Computer Science Department, University of Illnois at Urbana-Champaign, 201 N. Goodwin Ave., Urbana, IL 61801-2302. E-mail: kcchang@cs.uiuc.edu.*

Manuscript received 12 Feb. 2006; revised 19 Aug. 2006; accepted 3 Oct. 2006; published online 25 Jan. 2007.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0094-0206.
Digital Object Identifier no. 10.1109/TKDE.2007.1007.

1. We distinguish *search predicate* x and *expensive predicates* p_c and p_l in naming for the sake of presentation later on.

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 *post-filtering predicates* in ranking queries; as we will discuss, Boolean filtering conditions in ranking queries may turn fuzzy predicates into expensive ones. Fourth, it will enable *join* operations across multiple tables; as we will see, a fuzzy join predicate is essentially expensive. Our framework aims at generally supporting expensive predicates in the context of ranked queries, in order to handle the following expensive predicates:

- **User-defined Predicates**. 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 nonstandard predicates. While user-defined functions are now commonly supported in Boolean systems, such functions are clearly lacking for ranked queries, though they are more important for ranking based on similarity, relevance, and preference (e.g., as in [1]). As these ranking concepts are inherently imprecise and user (or application) specific, a practical system should support ad hoc user-defined ranking criteria. To illustrate, 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.
- **External Predicates**. A middleware system can integrate an "external" predicate that can only be evaluated by probing an autonomous source for each object. For instance, a middleware may integrate Web sources, say, to look for houses in "safe" areas as in Example 1, by querying some Web source to compute the "safety" based on the crime rate of the area retrieved.
- **Post-filtering Predicates**. Although our focus is on ranking, such query can often mix with Boolean query conditions, as they effectively filter out irrelevant tuples. For instance, continuing Example 1, the user can effectively focus the house search into only those located in Chicago, by specifying a Boolean filtering condition on zip as Query 2 below illustrates:

select id **from** house
where $zip = 60603$
order by $min(new(age), cheap(price, size), large(size))$
stop after 5 (**Query 2**)

While such queries can be supported in various ways, e.g., fundamental change of optimizer as in [4], an immediate and easy extension of relational DBMS for supporting Query 2 is to leverage relational query optimizer to effectively process the Boolean filtering condition and then evaluate fuzzy predicates. However, such extension may turn fuzzy predicates into expensive ones: To illustrate, suppose there exist index structures to access houses in the decreasing order of new, $cheap$, and $large$ scores. Now that the user is interested only in houses in Chicago, the indices built on a base table (of all houses) may no longer be efficient in retrieving high-scored Chicago houses, especially when Chicago houses are very few in the database.

- **Join Predicates**. Join predicates are expensive, because they are inherently user-defined operations: In the Boolean context, joins may arbitrarily associate attributes from multiple tables using user-defined Boolean join condition. Similarly, fuzzy joins may associate multiple attributes with an user-defined join predicate. Since a search mechanism cannot be precomputed for such fuzzy joins, fuzzy joins require expensive probes to evaluate each combined tuple (of the Cartesian product), as is the case in Boolean queries. To generally support fuzzy joins, a ranked-query system thus needs to support expensive predicates. To illustrate, 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
order by $min(new(h.age), rating(s.name), close(h.zip, s.zip))$
stop after 5 (**Query 3**)

These predicates, be they user defined or externally accessed, can be arbitrarily expensive to probe, potentially requiring complex computation or access to networked sources. Our key challenge is thus to minimize the cost of such expensive probes by 1) minimizing the objects to evaluate and 2) the predicate evaluations for such objects, while existing approaches miss either (e.g., TAz) or both (e.g., complete probing) of these two minimizations.

This paper pursues this goal of *probe minimization*, based on and expanding the "necessary-probe principle" studied in our preliminary work [5], which we compare to and contrast with in Section 5. Due to space limitation, the main text focuses on the core principles of probe minimization, while we also studied important relevant research issues, such as scalability or parallelization, which we leave in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm. To highlight, we summarize our 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, postfiltering predicates, and fuzzy joins. We

symbol	meaning
$\mathcal{F}$	scoring function
$\mathcal{D}$	dataset
k	retrieval size (or, as score threshold θ_k)
n	number of query predicates
m	number of data objects, *i.e.*, $\|\mathcal{D}\|$
$\mathcal{P}$	predicate schedule
$\mathcal{O}_\mathcal{P}$	object schedule with respect to $\mathcal{P}$
$\mathsf{pr}(u,p)$	probe to evaluate $p[u]$
$\mathcal{PC}$	probe cost

Fig. 1. Notation used in this paper.

are not aware of any previous work that formulates the general problem of supporting expensive predicates for *top-k* queries.

- **Cost-based optimization**. We enable probe minimization by optimizing probe schedule in a cost-based sense, by searching the space of schedules for the minimal-cost one. In particular, we decouple probe scheduling into object and predicate scheduling problems and develop an analytical object scheduling optimization and a dynamic predicate scheduling optimization, which combine together to form a cost-effective probe schedule.
- **Optimal Algorithm**. We develop Algorithm *MPro* which realizes our cost-based schedule optimization framework.
- **Experimental evaluation**. We report extensive experiments using both real-life and synthesized data sets. Our experimental study validates the optimality of our analytic object scheduling as "lower bound" performances and the effectiveness of our dynamic predicate scheduling over the existing schemes without a principled optimization approach.

We first define ranked queries in Section 2, and the baseline processing schemes. Section 3 then presents our cost-based schedule optimization framework, based on which we develop Algorithm *MPro*. Section 4 reports our experimental evaluation. We briefly discuss related work in Section 5 and conclude in Section 6.

2 Ranked Query Model

To establish the context of our discussion, this section describes the query semantics and cost model for expensive predicates (Section 2.1) and discusses existing frameworks to motivate and contrast our work (Section 2.2).

2.1 Query Semantics and Cost Model

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. As Fig. 1 summarizes, a ranked query is denoted as 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.

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 (Data set 1) of objects $\{a,b,c,d,e\}$. Fig. 2 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

$$\begin{aligned}\mathcal{F}(x,p_c,p_l)[a] &= \mathcal{F}(x[a],p_c[a],p_l[a])\\ &= min(0.9,0.85,0.75)=0.75.\end{aligned}$$

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

Fig. 2. Data set 1.

We can distinguish between selection and join predicates, just as in relational queries, i.e., depending on if the operation involves one or more objects. That is, 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 (of Example 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. (Conceptually, a join operation is a selection over joined objects in the Cartesian product of the joining tables.) For instance, p_j in Query 3 (in Section 1) 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, while we develop the extensions for joins in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm.

In this paper, we focus on an important class of scoring functions that are *monotonic*, which is typical for ranked queries [6]. Intuitively, in a monotonic function, all the predicates *positively* influence 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 [7] may support user query $\mathcal{F}=min(t_1,t_2)$ for fuzzy conjunction. An image or text [8] database may combine various features with a user-transparent function, such as Euclidean distance or weighted average.

As a result, a ranked query returns the *top-k* objects with the highest scores and, thus, it is 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 Data set 1 (Fig. 2) 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.

Alternatively, *top-k* queries can be viewed as retrieving objects u that score no less than some threshold θ_k, i.e., $\mathcal{F}[u] \geq \theta_k$. Note this *thresholding view* will be semantically equivalent to *top-k* query view when θ_k corresponds to the lowest score of *top-k* results: Though such θ_k is not known a priori in practice, this semantically equivalent view helps conceptualize important insights, as we will revisit in Section 3.

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 nonindex predicates that require per-object probes. First, system built-in predicates (e.g., [9], [10], [11]) can use precomputed 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 Query 1 (in Section 1), we assume x to be a search predicate. Fig. 2 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 such search indexes. As Section 1 explains, such *probe predicates* generally represent user-defined functions, external predicates, post-filtering predicates, and joins. Unlike search predicates that are virtually free,[2] 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, the well-known Fagin's algorithm [6] (Section 2.2 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 over the given database $\mathcal{D}$. Let $\mathcal{A}$ be an algorithm, we denote its probe cost by $\mathrm{PC}(\mathcal{A}, \mathcal{F}, \mathcal{D}, k)$. (Note the notation is used interchangeably with its shorthanded notion $\mathrm{PC}(\mathcal{A})$ for convenience.) Assuming that the per-probe cost for p_i is C_i and that $\mathcal{A}$ performs m_i probes for p_i, we model the probe cost as $\mathrm{PC}(\mathcal{A}) = \sum_{i=1}^{n} m_i C_i$. In particular, for a database of size m, a complete probing system (as Section 2.2 will discuss) will cost $\sum_{i=1}^{n} mC_i$. 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 guarantees minimal probe cost. That is, if $\mathcal{A}$ is probe-optimal, it does not waste any single probes and every probe it performs is thus necessary (by any algorithms). We will develop a probe-optimal algorithm in Section 3. In addition, Section 4 empirically shows that the probe-optimality of our algorithm makes the probe cost "proportional" to retrieval size k rather than database size m. This proportional cost can be critical, since users are typically only interested in a small number of top results.

2. While search predicates do incur cost, in this scenario of expensive predicates, the cost is nominal compared to probe costs that clearly dominate.

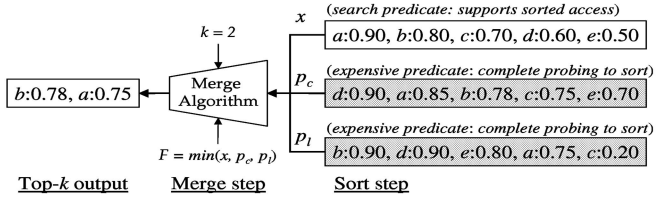


Fig. 3. The baseline scheme $SortMerge$.

2.2 Baselines: Sort-Merge and TAz 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 [6] as well as several later versions [12], [13], [14]. 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. To contrast, consider adopting the sort-merge framework for expensive predicates: As the name suggests, the scheme will require complete probing to *sort* objects for each expensive predicate, before *merging* the sorted streams—referred to as $SortMerge$, this naive scheme fully "materializes" probe predicates into search predicates. Fig. 3 illustrates Scheme $SortMerge$ for Query 1 (Example 1), where x is a search predicate while p_c and p_l are 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. That is, rather than the desired "proportional cost," $SortMerge$ always requires complete probing, regardless of the small k; i.e., $\mathrm{PC}(SortMerge) = \sum_{i=1}^{n} mC_i$, for a database of size m.

Later, Fagin [14] modified the scheme (referred to as *TAz*) to support "probe-only" predicates like p_c and p_l without "full materialization" of $SortMerge$: Instead, Algorithm *TAz* iteratively accesses each object from x and computes its final score (by completely probing p_c and p_l for this object), only until k evaluated objects have higher final scores than the maximal-possible scores of the rest.

We consider these two schemes as baselines for performance comparison (Section 4.1), as they represent two interesting "upper bound" costs: First, by completely probing every object, $SortMerge$ defines the absolute upper bound cost of complete probing. Second, by halting as soon as *top-k* results are identified, *TAz* minimizes the number of objects evaluated (*object minimization*). However, by always completely probing objects, its cost represents the upper bound cost of algorithms with object minimization. In contrast, this paper identifies the problem of *probe minimization*, which not only optimizes with respect to the number of objects evaluated, but also how many predicates

Algorithm *MPro*$(\mathcal{F}, k, \mathcal{D})$**:** 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.
- $\mathcal{D}$: input database // *assume selection predicates over single relation for simplicity.*

Output: $\mathcal{K}$, the *top-k* answers with respect to $\mathcal{F}$.
Procedure:
(1) *PSch*: identifies predicate schedule $\mathcal{P}$ for *OSch*.
// *can precede or interleave with object scheduling. see Section 4.3 for details.*
(2) *OSch*: identifies the optimal object schedule $\mathcal{O}_\mathcal{P}$ with respect to $\mathcal{P}$ from *PSch*.
(2-1) **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.*

(2-2) **Necessary Probing**:
// *set up the stop condition* $\mathcal{SC}$ *for determining whether to stop probing.*
- $\mathcal{SC} \leftarrow$ "$|\mathcal{K}| \geq k$,*i.e.*, at least k completely evaluated 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 completely evaluated:
    - $u.score \leftarrow u.ceiling$; append u to $\mathcal{K}$ // *add* u *to be the top-k output.*
  - else: // u *must be probed further.*
    - $p \leftarrow$ next unevaluated predicate of u on schedule $\mathcal{P}$
    - $p[u] \leftarrow$ probe $\mathsf{pr}(u, p)$ // $\mathsf{pr}(u,p)$ *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* $u.ceiling$.

(2-3) **Top-k Output**: return in order each (u:$u.score$) in $\mathcal{K}$

Fig. 4. Algorithm *MPro*.

are processed for each such object. As a result, as we will see in experiments (Section 4), most probes in $SortMerge$ and TAz are simply unnecessary in our probe-optimal algorithm.

3 Minimal Probing: Cost-Based Schedule Optimization

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 probe cost in processing the query. In this section, we tackle the problem as a cost-based schedule optimization of identifying the optimal probe schedule in the search space. Further, we propose Algorithm *MPro* (for minimal probing) which realizes the framework. Section 3.1 overviews our cost-based optimization framework and identifies object and predicate scheduling as subproblems, each of which corresponds to two "phases" of $OSch$ and $PSch$ of our algorithm *MPro* (Fig. 4). Sections 3.2 and 3.3 then tackle object and predicate scheduling, respectively.

3.1 Schedule Optimization: Overview

In this section, we develop *top-k* query processing as a cost-based schedule optimization and identify the focus of object and predicate scheduling as its subproblems (which will be discussed in Sections 3.2 and 3.3, respectively). That is, this section overviews how the problem can be decomposed into subproblems without compromising generality.

To begin with, we focus on a general class of *sequential* algorithms, as any algorithm can be considered sequential by flattening out the operations performed. (We also develop a "simple extension" to parallel probing in Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, though it is not the focus of this paper.) A sequential algorithm executes probes one at a time and stops if no further probes are necessary. Each probe evaluates a predicate p for an object u, designated by $\mathrm{pr}(u, p)$, to determine the score $p[u]$. Given a query with n expensive predicates, for a database with m objects, there are thus $m \cdot n$ possible probes (for each pair of an object and a predicate). A sequential algorithm will execute these probes in some order (and possibly terminate before exhausting all probes), forming a particular *probe schedule*. We will thus view any sequential algorithm as equivalent to a probe schedule $\mathcal{S}$ (which the algorithm produces). That is, an optimal sequential algorithm must in effect determine the optimal probe schedule that requires the least probe cost. Our task of sequential-algorithm optimization is thus precisely a probe scheduling problem, i.e., finding the minimal-cost probe schedule.

Example 2 (Scheduling Space). For our example Query 1 (Example 1): $\mathcal{F}(x, p_c, p_l)$ over Data set 1, there are two expensive predicates, p_c and p_l, and five objects $(a, \ldots, e)$, and thus $(5 \cdot 2) = 10$ probes. Different sequential algorithms will schedule these probes in different orders. The following are two possible orders:

$$\mathcal{S}_1 : \mathrm{pr}(a, p_c), \mathrm{pr}(a, p_l), \mathrm{pr}(b, p_c), \mathrm{pr}(b, p_l), \ldots, \mathrm{pr}(e, p_c), \mathrm{pr}(e, p_l) \text{ and}$$

$$\mathcal{S}_2 : \mathrm{pr}(a, p_c), \mathrm{pr}(a, p_l), \mathrm{pr}(b, p_l), \mathrm{pr}(b, p_c), \mathrm{pr}(c, p_c), \mathrm{pr}(c, p_l), \mathrm{pr}(d, p_l), \mathrm{pr}(d, p_c), \mathrm{pr}(e, p_c), \mathrm{pr}(e, p_l).$$

All together, there are $(5 \cdot 2)! = 3,628,800$ possible schedules.

Depending on the particular databases, some schedules may terminate earlier before exhausting all 10 probes (as Example 5 later will illustrate). As a result, different algorithms, by using different probe schedules, will result in different costs.

As Example 2 shows, given a query, possible sequential algorithms or probe schedules are numerous in any practical settings, resulting in a large search space for optimization. In particular, given $m \cdot n$ probes, the "space" of sequential algorithms, which we designate as Ω, consists of $(m \cdot n)!$ different schedules (each of which sequences the probes in a different way), i.e., $|\Omega| = (m \cdot n)!$. (Strictly speaking, the value is a tight lower bound of $|\Omega|$, as an algorithm may terminate earlier with less than $m \cdot n$ probes as Example 5 will illustrate.) Among all the algorithms $\mathcal{S}$ in Ω, with its probe cost denoted as $\mathrm{PC}(\mathcal{S}, \mathcal{F}, \mathcal{D}, k)$ with respect to the given scoring function $\mathcal{F}$, data set $\mathcal{D}$, and retrieval size k, our goal is to find the optimal one with the least cost, i.e.,

$$\mathrm{argmin}_{\mathcal{S} \in \Omega} \mathrm{PC}(\mathcal{S}, \mathcal{F}, \mathcal{D}, k). \quad (1)$$

Such optimization is clearly challenging as the space is prohibitively large, e.g., even for a toy setting illustrated in Example 2. (To reinforce, for a slightly larger setting of, say, 50 objects and 2 predicates, $|\Omega| = (50 \cdot 2)! = 9.3e + 157$).

Conceptually, we view a probe schedule $\mathcal{S}$ as consisting of object and predicate scheduling: During execution, any sequential algorithm must schedule the next probe $\mathrm{pr}(u, p)$ by selecting both an object u and a predicate p, or *object*

scheduling and *predicate scheduling*, respectively. Example 3 illustrates such dual schedules.

Example 3 (Object versus Predicate Scheduling). Continue Example 2. $\mathcal{S}_1$ executes an object schedule

$$\mathcal{O}_1 = (a, a, b, b, \ldots, e, e),$$

in which each object appears twice, i.e., once for each predicate. Further, $\mathcal{S}_1$ evaluates the predicates in the same order (p_c, p_l) for each object, i.e., with a predicate schedule

$$\mathcal{P}_1 = [a : (p_c, p_l), b : (p_c, p_l), \ldots, e : (p_c, p_l)].$$

Note that $\mathcal{P}_1$ and $\mathcal{O}_1$ together determine the complete schedule, which we denote by $\mathcal{S}_1 = (\mathcal{P}_1, \mathcal{O}_1)$. Similarly, we can consider another schedule $\mathcal{S}_2 = (\mathcal{P}2, \mathcal{O}2)$, interleaving evaluations on objects a and b, e.g., $\mathcal{O}_2 = (a, b, b, a, c, c, d, d, e, e)$, with object-specific predicate scheduling

$$\mathcal{P}_2 = [a : (p_c, p_l), b : (p_l, p_c), c : (p_c, p_l), d : (p_l, p_c), e : (p_c, p_l)].$$

The scheduling problem thus intrinsically couples object with predicate scheduling. That is, the dual schedulings among m objects and n predicates combine to form the space Ω. For a better understanding of this coupling, we study the space of object and predicate scheduling, which we designate as $\Omega_{\mathcal{O}}$ and $\Omega_{\mathcal{P}}$, respectively. First, how many different object schedules are possible in the space of $\Omega_{\mathcal{O}}$? Since an object schedule can be any permutation of the m objects, each of which appears n "identical" times, $|\Omega_{\mathcal{O}}| = \frac{(m \cdot n)!}{(n!)^m}$. Second, how many different predicate schedules are possible in the space of $\Omega_{\mathcal{P}}$? For each object, there are $(n!)$ different permutations of predicates; all together, there are $|\Omega_{\mathcal{P}}| = (n!)^m$ for all m objects. Note that $\Omega_{\mathcal{O}}$ and $\Omega_{\mathcal{P}}$ together form the combinatorial space Ω (as illustrated earlier), i.e., $|\Omega| = |\Omega_{\mathcal{P}}| \cdot |\Omega_{\mathcal{O}}| = (m \cdot n)!$.

Example 4 ($\Omega_{\mathcal{P}}$ and $\Omega_{\mathcal{O}}$). Continue our running example with $m = 5$ objects and $n = 2$ predicates. First, consider $\Omega_{\mathcal{P}}$: Since each of the five objects has 2! predicate schedules, which are (p_c, p_l) and (p_l, p_c), there are combinatorially $(2!)^5 = 32$ predicate schedules, i.e., $|\Omega_{\mathcal{P}}| = 32$. Second, consider $\Omega_{\mathcal{O}}$: Any permutation of $\mathcal{O}_1 = (a, a, b, b, \ldots, e, e)$ is also a valid object schedule and, therefore, there are $|\Omega_{\mathcal{O}}| = \frac{(5 \cdot 2)!}{(2!)^5} = 113,400$. Together, we see that $|\Omega_{\mathcal{P}}| \cdot |\Omega_{\mathcal{O}}| = |\Omega|$ (as Example 2 computed).

Can separating object and predicate schedulings help decouple their combinatorial complexity? In fact, our approach exploits such separation to reduce the search space: As Section 3.2 will discuss, the crucial foundation of our approach is the Necessary-Probe principle (Theorem 1) which enables to analytically determine the best object schedule, denoted $\mathcal{O}_{\mathcal{P}}$, *for any predicate schedule* $\mathcal{P}$ (Theorem 2). Therefore, to find the optimal complete schedule $\mathcal{S}$ (1), we can specifically focus on $\mathcal{S} = (\mathcal{P}, \mathcal{O}_{\mathcal{P}})$, where each predicate schedule $\mathcal{P}$ pairs with its corresponding optimal object schedule $\mathcal{O}_{\mathcal{P}}$, instead of arbitrary $\mathcal{O}$. Our task is thus reduced to finding the optimal predicate schedule $\mathcal{P}$ which, together with its implied object schedule $\mathcal{O}_{\mathcal{P}}$, results in the least probing cost. Note that, our Algorithm *MPro* (Fig. 4) also reflects this separation in design: $PSch$ phase finds the optimal predicate schedule $\mathcal{P}$ and $OSch$ phase identifies the optimal object schedule $\mathcal{O}_{\mathcal{P}}$ with respect to $\mathcal{P}$. Formally, we thus transform our optimization problem into

$$\text{argmin}_{\mathcal{P} \in \Omega_{\mathcal{P}}} \text{PC}((\mathcal{P}, \mathcal{O}_{\mathcal{P}}), \mathcal{F}, \mathcal{D}, k). \quad (2)$$

Note that the transformation from (1) to (2) significantly reduces the search space from $|\Omega| = (m \cdot n)!$ to

$$|\Omega| = |\Omega_{\mathcal{P}}| = (n!)^m,$$

e.g., for our toy example of $m = 5$ and $n = 2$: from 3,628,800 to 32; for $m = 50$ and $n = 2$: from $9.3e + 157$ to $1.1e + 15$.

Our next focus is thus to optimize predicate scheduling in the space of $\Omega_{\mathcal{P}}$. Toward the goal, we first develop the context-independence of predicate scheduling (Theorem 3), which states that the scheduling of each object is independent of other objects. In other words, for our database with m objects, $\mathcal{D} = \{u_1, \ldots, u_m\}$, our goal is to simply find the optimal "subschedule" $\mathcal{P}^i$ for each object u_i, such that these subschedules form the complete best schedule denoted as $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}m$.

What is then the objective of per-object scheduling $\mathcal{P}^i$? To understand, we introduce "thresholding view" (Section 2) of finding the stopping threshold, which we denote as θ_k. When it is set to the $\mathcal{F}$ score of the kth result, it is semantically equivalent of identifying *top-k* results. While its exact value cannot be known during processing, this alternative view is helpful in discussing optimal scheduling. In this thresholding view, our goal is now to identify the optimal subschedule $\mathcal{P}^i$ for each object u_i which minimizes the cost in identifying whether u_i is a *top-k* result or not, i.e., $\mathcal{F}u_i \geq \theta_k$ or not. As the optimal subschedule $\mathcal{P}^i$ is identified *independently* from its per-object predicate scheduling space, which we denote as $\omega_{\mathcal{P}}$, i.e., $|\omega_{\mathcal{P}}| = n!$, optimization problem is further reduced into finding $\mathcal{P}^i$ minimizing "per-object" probe cost on the corresponding object u_i, which we denote as $\text{C}(\mathcal{P}^i, \mathcal{F}, u_i, \theta_k)$. We formally state our goal as follows:

$$\begin{aligned} &\text{argmin}_{\mathcal{P}=\mathcal{P}^1 \times \cdots \times \mathcal{P}^m \in \omega_{\mathcal{P}} \times \cdots \times \omega_{\mathcal{P}}} \text{PC}((\mathcal{P}, \mathcal{O}_{\mathcal{P}}), \mathcal{F}, \mathcal{D}, k) \\ = \; &[\text{argmin}_{\mathcal{P}^1 \in \omega_{\mathcal{P}}} \text{C}(\mathcal{P}^1, \mathcal{F}, u_1, \theta_k)] \times \ldots \times \\ &[\text{argmin}_{\mathcal{P}^m \in \omega_{\mathcal{P}}} \text{C}(\mathcal{P}^m, \mathcal{F}, u_m, \theta_k)]. \end{aligned} \quad (3)$$

As each $\mathcal{P}^i$ is some permutation of the n predicates, $|\omega_{\mathcal{P}}| = n!$, this transformation of finding each $\mathcal{P}^i$ independently reduces the search space from $(n!)^m$ to $(n!) \cdot m$, i.e., making an $n!$-decision for m times. To illustrate its significance, the reduction is from $(2!)^5 = 32$ to $(2! \cdot 5) = 10$ in our toy example of $m = 5$ and $n = 2$, and $1.1e + 15$ to $2! \cdot 50 = 100$ for $m = 50$ and $n = 2$.

In summary, we have overviewed how we significantly reduce the search space of probe scheduling Ω, by decoupling it into that of object and predicate schedulings as the following sections will discuss in details: Section 3.2 will discuss our provably optimal object scheduling, which analytically pairs predicate schedule $\mathcal{P}$ with its corresponding optimal object schedule $\mathcal{O}_{\mathcal{P}}$ and thus reduces the search space into that of predicate scheduling, i.e., $|\Omega| = |\Omega_{\mathcal{P}}| = (n!)^m$. Section 3.3 will then follow to discuss how to optimize in the space of $\Omega_{\mathcal{P}}$. The section

will first develop context-independence to further reduce the search space into $|\Omega| = (n!) \cdot m$, as we discussed. It will then develop a precise analytic cost model and argue to focus on finding a global schedule, i.e., identical $\mathcal{P}^i$ for every object, which further reduces the search space to only $n!$.

3.2 Object Scheduling: Analytic Principle

As overviewed in Section 3.1, this section first focuses on object scheduling problem: We first present the Necessary-Probe principle (Theorem 1) which analytically determines the best object schedule $\mathcal{O}_\mathcal{P}$ for the given predicate schedule $\mathcal{P}$. We then implement this into an object scheduling phase $OSch$ of our probe-optimal algorithm *MPro*.

3.2.1 Necessary-Probe Principle

Given a predicate schedule $\mathcal{P}$, how shall we proceed to probe to minimize the probe cost? Clearly, since only some *top-k* answers are requested, complete probing (for every object) might not be necessary. In fact, some probes may not need to be performed at all, if they can never be the top answers. Thus, to enable probe minimization, we must determine if a particular probe of predicate p on object u, designated by $\mathrm{pr}(u, p)$, is truly *necessary* for finding the *top-k* answers, as Definition 1 below formalizes.

Definition 1 (Necessary Probes). *Consider a ranked query with scoring function $\mathcal{F}$ and retrieval size k with respect to a given predicate schedule $\mathcal{P}$. A probe $\mathrm{pr}(u, p)$, which probes the next unevaluated predicate in $\mathcal{P}$ for object u, is* necessary, *if no other algorithm with the same predicate schedule $\mathcal{P}$ can determine the correct* top-k *answers without performing the probe, regardless of the results of other probes.*

We stress that, with given predicate schedule $\mathcal{P}$ determining the next predicate p to probe for each object u, determining whether a probe $\mathrm{pr}(u, p)$ is necessary is essentially "object scheduling" of deciding whether u needs to be scheduled. Consequently, a probe $\mathrm{pr}(u, p)$ being necessary indicates the following in object scheduling: First, u needs to be probed independent of algorithms—any correct algorithm with predicate schedule $\mathcal{P}$ must pay the probe. Second, u needs to be probed independent of the outcomes of other objects—it may be performed before or after others, but in either case, the probe is required. While this notion of necessary probes seems intuitively appealing, some questions remain to be addressed: First, how can we determine if a *particular* probe is absolutely required? Second, given there are many possible probes (at least one for each object at any time), which ones of them are actually necessary, and how to effectively find *all* the necessary probes? We next develop a simple principle for addressing 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 Data set 1, given schedule $\mathcal{P} = [a : (p_c, p_l), b : (p_c, p_l), c : (p_l, p_c), d : (p_l, p_c), e : (p_l, p_c)]$. Our starting point, before any probes, is the sorted output of search predicate x (as in Fig. 2 with objects sorted by their x scores). As $\mathcal{P}$ specifies, we thus can choose to probe the next predicate p_c for any object a or b, or predicate p_l for c, d, or e. (Note that $\mathcal{P}$ only specifies *predicate* orders, but not which *object* to probe next.) Let's consider the top object a and determine if $\mathrm{pr}(a, p_c)$ 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. To prove, assume we can somehow determine the *top*-1 answer to be some object u without probing $\mathrm{pr}(a, p_c)$.

- 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 Fig. 2). Consequently, $\mathcal{F}[u] \leq 0.8$. However, without probing $\mathrm{pr}(a, p_c)$ and then $\mathrm{pr}(a, p_l)$, 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 $\mathrm{pr}(a, p_c)$, 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 (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 gives the upper bound, when only T is evaluated, i.e., $\mathcal{F}[u] \leq \overline{\mathcal{F}}_T[u]$.

$$\overline{\mathcal{F}}_t(t_1, \ldots, t_n)[u] = \mathcal{F}\begin{pmatrix} t_i = t_i[u] & \text{if } t_i \in T \\ t_i = 1.0 & \text{otherwise} \end{pmatrix} \forall i. \quad (4)$$

To further illustrate, after $\mathrm{pr}(a, p_c)$, what shall we probe next? Intuitively, at least we have choices of $\mathrm{pr}(a, p_l)$ (to complete a) or $\mathrm{pr}(b, p_c)$ (to start probing b). Similarly to the above, we can reason that an additional probe on a, i.e., $\mathrm{pr}(a, p_l)$, is still necessary. To contrast, we show that $\mathrm{pr}(b, p_c)$ is not necessary *at this point*, according to Definition 1. (However, as more probes are done, at a later point, $\mathrm{pr}(b, p_c)$ 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 $\mathrm{pr}(a, p_l)$ of a. (Note that a already has $x[a] = 0.9$ and $p_c[a] = 0.85$ evaluated.)

- Suppose that $\mathrm{pr}(a, p_l)$ 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 $\mathrm{pr}(b, p_c)$.

- Suppose that $\mathrm{pr}(a, p_l)$ 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 $\mathrm{pr}(b, p_c)$ is necessary, similar to $\mathrm{pr}(a, p_c)$ 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 predicate schedule $\mathcal{P}$) for u. Potentially, any probe $\mathrm{pr}(u, p)$ 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 2 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 $\mathrm{pr}(u, p)$ for p specified by the given predicate schedule $\mathcal{P}$ is necessary. Theorem 1 formally states this result (see the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, 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 $\mathcal{P}$, let u be an object and p be the next unevaluated predicate for u in $\mathcal{P}$. The probe $\mathrm{pr}(u, p)$ 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.*

Theorem 1 provides an "operational" definition to actually determine if a given probe is necessary as well as to effectively search those probes. As the theorem isolates only those probes *absolutely* required by any correct algorithm, we can immediately conclude that an algorithm will be probe-optimal (Section 2.1) with respect to $\mathcal{P}$, if it only performs necessary probes defined in Definition 1. Lemma 1 formally states this intuition (see the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, for a formal proof).

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

Lemma 1 plays an essential role in probe minimization. First, Lemma 1 shows that probe minimization boils down to cost-optimal predicate schedule $\mathcal{P}$. Second, Lemma 1 then enables us to compute the lower bound cost of various schedules, which, in turn, enables us to identify a cost-minimal schedule among them, as Section 3.3 will further discuss. Summing up, Lemma 1 enables us to find a truly optimal algorithm in principle, by simplifying the optimality into predicate schedule optimization and enabling to identify the optimal cost for each predicate schedule.

Our goal is thus to develop such algorithm. However, note such algorithm is not necessarily unique, as at any point, there can be multiple necessary probes to choose from. However, any such object schedule will be "equally" optimal. Section 3.2 develops one such probe-optimal algorithm *MPro*.

3.2.2 Algorithm MPro

We next introduce our probe-optimal algorithm *MPro* based on the Necessary-probe principle just developed. Algorithm *MPro* consists of the two phases of predicate and object scheduling, i.e., $PSch$ and $OSch$, respectively, as Fig. 4 illustrates. Note, however, that these two phases are only separated conceptually and the two can be interleaved as we will revisit in Section 3.3.

This section first focuses on developing a probe-optimal object scheduling scheme $OSch$. Later, Section 3.3 will develop a predicate scheduling scheme, identifying the optimal predicate schedule $\mathcal{P}$ for $OSch$, to complete Algorithm *MPro*. $OSch$ essentially implements the Necessary-probe principle which identifies the optimal object schedule $\mathcal{O}_\mathcal{P}$ with respect to predicate schedule $\mathcal{P}$, i.e., $\mathcal{O}_\mathcal{P} = OSch(\mathcal{P})$. (Note, for notational simplicity, we use $OSch$ both as a phase of *MPro* and as the object schedule $\mathcal{O}_\mathcal{P}$ it produces.) Given predicate schedule $\mathcal{P}$ (identified from $PSch$ phase), the $OSch$ phase optimizes probe cost by ensuring that every probe performed is indeed necessary (for finding k top answers). Essentially, during query processing, the $OSch$ phase keeps "searching" for a necessary probe to perform next. Progressing with more probes, eventually the $OSch$ phase 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 $\mathrm{pr}(u, p)$ 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—e.g., a priority queue [15] that buffers objects using their ceiling scores as priorities.

We thus design the $OSch$ phase to operate on such a queue, called *ceiling queue* and denoted as $\mathcal{Q}$, of objects prioritized by their ceiling scores. As Fig. 4 shows, the $OSch$ phase mainly consists of two steps: First, in the *queue initialization* step, starting with the sorted output stream $\mathcal{X}$ of search predicate x, $\mathcal{Q}$ is initialized based on initial ceiling scores $\overline{\mathcal{F}}_{\{x\}}[\cdot]$ with x being the only evaluated predicate (see (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, $\mathcal{X}$ can be accessed 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, the $OSch$ phase keeps on requesting from $\mathcal{Q}$ the top-priority object u, which has the highest ceiling score. Let u be *completely evaluated* if $\overline{\mathcal{F}}_{T_u}[u] = \mathcal{F}[u]$, i.e., either u is fully probed or $\overline{\mathcal{F}}_{T_u}[u]$ has

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 $\mathrm{pr}(a, p_c)$	a:0.85, b:0.80, c:0.70, d:0.60, e:0.50	{}
3.	probe $\mathrm{pr}(a, p_l)$	b:0.80, $\underline{a}$:0.75, c:0.70, d:0.60, e:0.50	{}
4.	probe $\mathrm{pr}(b, p_c)$	b:0.78, $\underline{a}$:0.75, c:0.70, d:0.60, e:0.50	{}
5.	probe $\mathrm{pr}(b, p_l)$	$\underline{b}$: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

Fig. 5. Illustration of $OSch$.

already reached minimal-possible score 0. Thus, if u is completely evaluated when it surfaces 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 and will be moved to an output queue, denoted as $\mathcal{K}$ in Fig. 4. Otherwise, If u is incomplete with the next unevaluated predicate p, according to Theorem 1, $\mathrm{pr}(u, p)$ is necessary. Thus, the $OSch$ phase will perform this probe, update the ceiling score of u, and insert it back to $\mathcal{Q}$ by the new score.

Incrementally, more objects will complete and surface to $\mathcal{K}$, and the $OSch$ phase will eventually stop when there are k such objects (which will be the *top-k* answers). As Fig. 4 shows, the $OSch$ phase checks this *stop condition*, designated as $\mathcal{SC}$, to halt 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 Appendix 3.1, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, how the stop condition can be "customized" for approximate queries.)

Fig. 5 illustrates how the $OSch$ phase actually works for our example of finding the top two objects when $\mathcal{F} = min(x, p_c, p_l)$ and

$$\mathcal{P} = [a : (p_c, p_l), b : (p_c, p_l), c : (p_l, p_c), d : (p_l, p_c), e : (p_l, p_c)]$$

over Data set 1 (Fig. 2). While we show the ceiling queue $\mathcal{Q}$ as a sorted list, full sorting is not necessary for a priority queue. After it is initialized from the sorted output of x, we simply keep on probing the top incomplete object in $\mathcal{Q}$, resulting in the probes $\mathrm{pr}(a, p_c)$, $\mathrm{pr}(a, p_l)$, $\mathrm{pr}(b, p_c)$, and $\mathrm{pr}(b, p_l)$. Each probe will update the ceiling score of the object and its priority in ceiling queue. Note that Fig. 5 marks objects with an underline (e.g., $\underline{a : 0.75}$) when they are completely evaluated, at which point their ceiling score are 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 the $OSch$ phase is both correct and optimal, as we state in Theorem 2. First, it will correctly return the *top-k* answers. $OSch$ phase stops when all the k objects with highest ceiling scores are all completely evaluated (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.

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

Fig. 6. Data set 2.

Second, $OSch$ phase is probe-optimal with respect to $\mathcal{P}$. Note that the $OSch$ phase 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, the $OSch$ phase halts). It thus follows from Lemma 1 that the $OSch$ phase is probe-optimal with respect to $\mathcal{P}$, because it only performs necessary probes, which, in turn, implies the optimality of the overall algorithm *MPro* if predicate schedule $\mathcal{P}$ (identified from $PSch$) is optimal, as Theorem 2 states.

Theorem 2 (Correctness and Conditional-Optimality of MPro). *Given scoring function $\mathcal{F}$ and retrieval size k, Algorithm* MPro *will correctly return the* top-k *answers.* MPro *is 1) probe-optimal with respect to the given predicate schedule $\mathcal{P}$, which, in turn, implies it is 2) probe-optimal if predicate schedule $\mathcal{P}$ is optimal.* □

3.3 Predicate Scheduling: Dynamic Optimization

In Section 3.2, we discussed our provably optimal object scheduling, which analytically determines the optimal object schedule $\mathcal{O}_\mathcal{P}$ with respect to the given predicate schedule $\mathcal{P}$. This section completes the picture by studying predicate scheduling problem and proposing effective dynamic predicate scheduling schemes that identify the best predicate schedule for $OSch$ phase.

Our optimization task now is thus to identify the best schedule $\mathcal{P}$ as (2) indicates. Recall that, given probe predicates $p_1, \ldots, p_n$, there are $n!$ possible predicate ordering (each as a permutation of p_i) for each object; for a database with m objects, i.e., $\mathcal{D} = \{u_1, \ldots, u_m\}$, there are thus $(n!)^m$ predicate schedules. Different predicate schedules incur different probe cost in the $OSch$ phase, as Example 5 below shows.

Example 5. To illustrate how predicate scheduling affects costs, consider $\mathcal{F} = min(x, p_c, p_l)$, using Data set 2 (Fig. 6) of $m = 3$ (objects) and $n = 2$ (expensive predicates. Among $(2!)^3 = 8$ predicate schedules of varying costs, we contrast two schedules $\mathcal{P}_1 = [a : (p_c, p_l), b : (p_c, p_l), c : (p_l, p_c)]$ and $\mathcal{P}_2 = [a : (p_l, p_c), b : (p_l, p_c), c : (p_l, p_c)]$. To find *top*-1, when given $\mathcal{P}_1$ and $\mathcal{P}_2$, the $OSch$ phase will perform six and four probes, respectively (as shown below); $\mathcal{P}_2$ is better, with 33 percent fewer probes.

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

3.3.1 Compositional Scheduling: Context Independence

Toward the goal of identifying the minimal-cost predicate schedule, we first develop the context-independence of predicate scheduling, which justifies to significantly reduce the complexity of predicate scheduling without compromising generality, from the combinatorial space of optimizing

probe scheduling for all objects, i.e., $(n!)^m$ possible schedules, to optimizing predicate scheduling for each object at a time, i.e., $m \times (n!)$.

To illustrate, observe predicate schedule $\mathcal{P}_1$ in Example 5 which consists of subschedules $\mathcal{P}_1^1 = (p_c, p_l), \mathcal{P}_1^2 = (p_c, p_l)$, and $\mathcal{P}_1^3 = (p_l, p_c)$ for object a, b, and c, respectively. Our first question is: Does the best predicate schedule of an object, say a, depends on those of others, say b and c? The following theorem of context independence addresses this question.

Theorem 3 (Context Independence of Per-Object Predicate Scheduling). *For a ranked query over a database $\mathcal{D} = \{u_1, \ldots, u_m\}$, consider any predicate schedules $\mathcal{P}_1 = \mathcal{P}_1^1 \times \cdots \times \mathcal{P}_1^m$ and $\mathcal{P}_2 = \mathcal{P}_2^1 \times \cdots \times \mathcal{P}_2^m$. Given either $\mathcal{P}_1$ or $\mathcal{P}_2$, the OSch phase performs exactly the same probes for object u_i if $\mathcal{P}_1^i = \mathcal{P}_2^i$.*

Theorem 3 states that the probe cost of an object depends only on its own schedule and will not be affected by others. For instance, consider $\mathcal{P}_1$ and $\mathcal{P}_2$ in Example 5: Since object c has the same schedule in both, i.e., $\mathcal{P}_1^3 = \mathcal{P}_2^3 = (p_l, p_c)$, the $OSch$ phase performs exactly the same probes on c, i.e., $\mathrm{pr}(c, p_l)$ and $\mathrm{pr}(c, p_c)$, in either $\mathcal{P}_1$ or $\mathcal{P}_2$.

This result follows directly from Lemma 2, which states that the probe cost of $OSch$ on any object u_i depends only on its *own* subschedule $\mathcal{P}^i$ and the lowest *top-k* score θ_k. Intuitively, $OSch$ performs probes on u_i either to compute its final score $\mathcal{F}[u_i]$ (if u_i is a *top-k* answer) or disqualify it as a *top-k* answer (otherwise). To illustrate, consider predicate schedule $\mathcal{P}_2$ in Example 5: the *top*-1 answer c is probed completely, while the rest is only partially probed just enough to show their disqualification from *top-k* results, e.g., $\overline{\mathcal{F}}_{\{x,p_l\}}[a] < \theta_k = 0.3$. More formally, Lemma 2 states that each object u_i will be either completely evaluated, or partially up to *j-prefix* of its predicate schedule $\mathcal{P}^i$, denoted as $\mathcal{P}^i(j)$, such that $\overline{\mathcal{F}}_{\mathcal{P}^i(j)}[u_i] < \theta_k$. As θ_k is determined per query and thus independent from schedules of other objects, we can conclude that each subschedule impacts only the cost of its corresponding object. (See the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, for a proof.)

Lemma 2 (Per-Object Probe Cost for $OSch$). *For a ranked query with retrieval size k over a database $\mathcal{D} = \{u_1, \ldots, u_m\}$, consider any predicate schedules $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}m$. Suppose the lowest score of the* top-k *results is θ_k. Consider any object u_i; let $\mathcal{P}^i = (p_1, \ldots, p_n)$. With respect to $\mathcal{P}$, the cost of probes that the OSch phase will perform upon each object u_i is $\mathrm{C}(\mathcal{P}^i, \mathcal{F}, u_i, \theta_k) = C_1 + \cdots + C_j$, where C_i is probe cost for $\mathcal{P}^i$, such that*

$$\begin{cases} j \text{ is the smallest in } [1{:}n) \ s.t. \overline{\mathcal{F}}_{\{x,p_1,\ldots,p_j\}}[u_i] < \theta_k & \text{if such } j \text{ exists} \\ j{=}n & \text{otherwise.} \end{cases}$$

Such context independence (Theorem 3) enables a *compositional* approach (rather than a combinatorial one) over the predicate scheduling problem. Recall that the goal of predicate scheduling is to construct the optimal overall schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}m$ that minimizes the cost. In fact, the cost $PC(\cdot)$ to minimize is the *sum* of the per-object cost of each u_i. Since the per-probe cost of each object is independent from the schedules of others (as Lemma 2 states), the construction of $\mathcal{P}$ is a simple *modular composition*: $\mathcal{P}$ can be constructed by independently constructing the optimal subschedule $\mathcal{P}^i$ for each object u_i, as (3) expresses (and as repeated below). As Section 3.1 explained, this compositional view reduces the search space to $(n!) \cdot m$.[3]

$$\begin{aligned} & \mathrm{argmin}_{\mathcal{P}=\mathcal{P}^1\times\cdots\times\mathcal{P}^m\in\omega\mathcal{P}\times\cdots\times\omega\mathcal{P}} \mathrm{PC}((\mathcal{P}, OSch(\mathcal{P})), \mathcal{F}, \mathcal{D}, k) \\ = \ & \mathrm{argmin}_{\mathcal{P}=\mathcal{P}^1\times\cdots\times\mathcal{P}^m\in\omega\mathcal{P}\times\cdots\times\omega\mathcal{P}} \mathrm{C}(\mathcal{P}^1, \mathcal{F}, u_1, \theta_k) + \ldots + \\ & \quad \mathrm{C}(\mathcal{P}^m, \mathcal{F}, u_m, \theta_k) \\ = \ & [\mathrm{argmin}_{\mathcal{P}^1\in\omega\mathcal{P}} \mathrm{C}(\mathcal{P}^1, \mathcal{F}, u_1, \theta_k)] \times \ldots \times \\ & [\mathrm{argmin}_{\mathcal{P}^m\in\omega\mathcal{P}} \mathrm{C}(\mathcal{P}^m, \mathcal{F}, u_m, \theta_k)]. \end{aligned}$$

3.3.2 Global Scheduling

To enable a cost-based search for the optimal predicate schedule $\mathcal{P}$, this section develops an analytic cost model, based on which we argue to focus on global predicate scheduling. We first develop a cost model for per-object cost, to identify the optimal subschedule $\mathcal{P}^i$ for each object u_i, which, in turn, identifies the complete best schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n$: Given $\mathcal{P}^i = (p_1, \ldots, p_n)$, the $OSch$ phase will probe each p_j in turn for u_i. Lemma 2 asserts that the $OSch$ phase will perform probes in the shortest j-prefix such that $\overline{\mathcal{F}}_{\mathcal{P}^i(j)}[u_i] < \theta_k$, or otherwise the entire $\mathcal{P}^i$. In other words, predicate p_r will be evaluated (and thus incurs its per-probe cost C_r) only when $\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i] \geq \theta_k$. As we must construct $\mathcal{P}^i$ *before* actually executing the probe, we compare the *expected* cost (without knowing predicate scores) in terms of the probabilities that the above inequality will hold, denoted as $Pr(\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i])$. Our task is thus to identify $\mathcal{P}^i$ that minimizes:

$$E[\mathrm{C}(\mathcal{P}^i, \mathcal{F}, u_i, \theta_k)] = \sum_{r=1}^{n} Pr(\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i] \geq \theta_k) \cdot C_r. \qquad (5)$$

Now, we face an important design choice between *per-object* scheduling which tailors a scheduling decision for each object leveraging "object-specific" information, and *global* scheduling which makes the optimal decision for all objects as a whole. This decision involves a tradeoff of benefit and overhead. While our development so far (the optimization formalism in general and the algorithm *MPro* in particular) can generally deal with both per-object and per-query scheduling, we choose to focus on the global strategy as we believe that the overhead of per-object scheduling generally offsets its benefit.

In terms of *overhead*, per-object scheduling inherently incurs at least m-fold scheduler overhead to generate $\mathcal{P}^i$ specifically to each of the m objects. However, in practice, the overhead often reaches up to mn-fold as per-object scheduling may actually require *per-probe* optimization (as we explain later). Such overhead, linear to the database and query sizes (m and n, respectively), is often unacceptable in most scenarios (other than in rare cases when probes are extremely expensive).

3. As the complexity of predicate scheduling depends on the number of "expensive predicate" (i.e., only truly ad hoc predicates), which is typically not large, the overhead is typically nominal, as we also evaluate empirically in Section 4.

Now the question is: In terms of *benefit*, can the per-object optimal scheduling be significantly "more optimal" then the global optimal schedule, to merit the overhead? As (5) suggests, an optimal schedule depends on the "oracle" of some values unknown during optimization, e.g., θ_k and $\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i]$. From an "information-theoretic" viewpoint, per-object scheduling will work better only if object-specific prediction of these values results in better oracles. (In particular, scheduling will be truly optimal if both parameters can be exactly predicted without executing any probe, which is infeasible in practice.) Thus, to evaluate benefit, we observe whether the prediction of θ_k and $\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i]$ benefit from object-specific information on u_i: First, the prediction of θ_k cannot benefit, as the parameter is determined per-query and thus not specific to each object by nature. Though the prediction can benefit from per-object invocation by leveraging more up-to-date score information, it does not argue for per-object scheduling, as global scheduling as well can similarly benefit from "multiple invocations" of schedulers to adaptive the predicate using more information accumulated from each evaluation, as we will develop into an adaptive global scheduling scheme *MPro-Adaptive* later.

Second, the prediction of $\overline{\mathcal{F}}_{\mathcal{P}^i(r-1)}[u_i]$ can benefit *only if* its already known scores $p[u_i]$, i.e., $p \in \mathcal{P}^i(r-2)$, can predict unknown ones. Such prediction thus requires 1) preconstructed information to correlate known scores to unknown ones and 2) an incremental probe-by-probe invocation of the scheduler to take advantage of known scores in decision. However, it is often practically impossible to construct such object-specific correlation information, and also per-probe invocation potentially increases the scheduler overhead up to mn-fold. Because of these difficulties, while there have been several per-object schemes, their overhead rarely pays off, as our experiments empirically confirm (Section 4).

We thus decide to focus on finding a global schedule $\mathcal{P}$ of every object following the same per-object schedule $\mathcal{P}^g$, i.e., $\mathcal{P} = \mathcal{P}^g \times \cdots \times \mathcal{P}^g$, which minimizes the cost across all objects in general. As we no longer discriminate objects in scheduling, our task is rephrased for any object u in the database as follows, which reduces the search space to $n!$:

$$E[\mathrm{PC}(\mathcal{P}, \mathcal{F}, \mathcal{D}, \theta_k)] = m \cdot \sum_{r=1}^{n} Pr(\overline{\mathcal{F}}_{\mathcal{P}^g(r-1)}[u] \geq \theta_k) \cdot C_r. \quad (6)$$

We now develop predicate scheduling algorithms that identify the optimal global schedule $\mathcal{P}g$ which minimizes the expected probe cost above. Observe from the equation that, $\mathrm{PC}((\mathcal{P}, OSch(\mathcal{P})), \mathcal{F}, \mathcal{D}, k)$ can be exactly computed if the number of object $u \in \mathcal{D}$ that satisfies $\overline{\mathcal{F}}_{\mathcal{P}^g(r-1)}[u] \geq \theta_k$ is known. We thus define the *aggregate selectivity* $\mathcal{S}_{\mathcal{F}}\theta(T)$ for a set of predicates T as the *ratio* of database objects u that "pass" $\overline{\mathcal{F}}_T[u] \geq \theta$. (This selectivity notion, unlike its Boolean-predicate counterpart, depends on the "aggregate filtering effect" of all the predicates evaluated.) With the aggregate selectivity notion, we can formally state the exact probe cost of global schedule $\mathcal{P} = \mathcal{P}^g \times \cdots \times \mathcal{P}^g$ as:

$$\mathrm{PC}(\mathcal{P}, OSch(\mathcal{P}), \mathcal{F}, \mathcal{D}, \theta_k) = m \cdot \sum_{r=1}^{n} \mathcal{S}_{\mathcal{F}}^{\theta_k}(\mathcal{P}^g(r-1)) \cdot C_r.$$

Consequently, the key challenge in developing predicate scheduling schemes is to estimate this aggregate selectivity: Although preconstructed statistics are often used in Boolean context, such requirement is unlikely to be realistic in our context, because predicates are either "dynamic" or "external" (Section 1). We develop three scheduling schemes with different strategies on when and how to collect statistics for estimating aggregate selectivity:

1. **Single invocation scheme**. First, we develop single invocation schemes *MPro-Sampling* and *MPro-EP* identifying an ideal predicate scheduling once prior to object scheduling, with varying strategies on how they gather statistics.

   MPro-Sampling. One way to gather dynamic statistics for estimating selectivities is to perform online sampling: At optimization time (i.e., before actual execution), the scheduler will sample a *small* number of objects and perform complete probing for their dynamic scores. While such sampling may add "unnecessary" probes, finding a good schedule can well justify this overhead (Section 4). 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_k. The uniform sampling will select some k' *top-k* objects proportional to the sample size s, i.e., $k' = \lceil k \cdot \frac{s}{m} \rceil$. That is, the sampling transforms a *top-k* query on the database into a *top-k'* query on the samples. Thus, θ_k can be estimated as the lowest $\mathcal{F}$ score of the *top-k'* sampled objects. Aggregate selectivity $\overline{\mathcal{F}}_{\mathcal{P}^g(r-1)}[u] \geq \theta_k$ can be then computed out of samples, by counting the ratio of sample objects satisfying $\overline{\mathcal{F}}_{\mathcal{P}^g(r-1)}[u] \geq \theta_k$.

   To illustrate, consider Fig. 6 as samples for $\mathcal{F} = min(x, p_c, p_l)$. Assume $k' = 1$ and thus $\theta_k = 0.3$ (the *top*-1 $\mathcal{F}$ score); let the relative predicate costs be $C_{p_c} = 1$ and $C_{p_l} = 3$. Our scheduler then compares the costs of potential schedules $\mathcal{P}_1^g = (p_c, p_l)$ and $\mathcal{P}_2^g = (p_l, p_c)$. To begin with, we compute the cost of $\mathcal{P}_1^g$, which is $\overline{\mathcal{F}}_{\{x\}} \cdot C_{p_c} + \overline{\mathcal{F}}_{\{x,p_c\}} \cdot C_{p_l}$ according to (6). Since all sampled objects satisfy $\overline{\mathcal{F}}_{\{x\}} \geq 0.3$ and $\overline{\mathcal{F}}_{\{x,p_c\}} \geq 0.3$, it follows that

   $$\mathcal{S}_{\mathcal{F}}^{0.3}(\{x\}) = \mathcal{SF}^{0.3}(\{x, p_c\}) = \frac{3}{3} = 1.$$

   The cost of $\mathcal{P}_1^g$ is thus $1 \cdot 1 + 1 \cdot 3 = 4$. Similarly, $\mathcal{S}_{\mathcal{F}}^{0.3}(\{x, p_l\}) = (\frac{1}{3})$ and, thus, the cost of $\mathcal{P}_2^g$ can be computed as $1 \cdot 3 + 0.3 \cdot 1 = 3.3$. Consequently, the scheduler will select $\mathcal{P}_2^g$ over $\mathcal{P}_1^g$.

   MPro-EP. Alternatively, in some scenarios, we can employ external parameters which "hint" the scheduler on the selectivity. For instance, when scoring function is a weighted average, different weights indicate different impacts of each predicate toward the overall score. In particular, the scheduler can take advantage of such hints to schedule the predicate with high impact but low per-probe cost

first. Meanwhile, when such external information is not available, e.g., $\mathcal{F} = min$, this scheme can no longer differentiate predicate selectivities and, thus, it degenerates to a simplistic scheduling by per-probe costs C_i.

2. **Multiple invocation scheme**. Second, we develop a multipe invocation alternative. Recall, as mentioned earlier, although Fig. 4 shows conceptual separation of object and predicate scheduling, the two can be interleaved by multiple invocations of predicate schedulers for leveraging statistics accumulated during object scheduling, as the following scheme *MPro-Adaptive* develops.

MPro-Adaptive. Alternatively, we can wait to activate $PSch$ until $OSch$ performs some necessary probes, to "adapt" predicate scheduling to dynamic statistics gathered as a by-product of such probes. In particular, *MPro-Adaptive* consists of the following two phases: First, until accruing enough statistics to determine a desirable predicate scheduling, the $OSch$ phase keeps on performing a necessary probe on top-priority object u. However, as predicate schedule has not been determined, we randomly decide the next predicate p to probe for u for now. As $p[u]$ is evaluated from such probe, we add its score to the list $Scores_p$. Second, once we accrue enough statistics (controlled by user-defined parameter s deciding the size of $Scores_p$, e.g., $s = 0.1\%$ of data), as we will discuss in Section 4), we estimate selectivities from the $Scores$ lists collected: In essence, we build samples (as in Fig. 6) by corresponding each $Score_p$ to a column of p scores in the table. However, note the difference from online sampling that, each list now evaluates different sets of objects. We thus arbitrary associate scores into objects, e.g., ith item of every $Score$ list together form a virtual sample object s_i, and identify the optimal predicate schedule $\mathcal{P}^g$ over samples, just as demonstrated for online sampling. (The performance implication of building samples out of necessary probes will be further discussed in Section 4.) With $\mathcal{P}^g$ identified, the $OSch$ phase can resume with respect to the schedule.

We now complete the presentation of our core algorithm *MPro* (Fig. 4). It conceptually consists of object and predicate scheduling, which will be empirically evaluated against the optimal scheduling and other existing schemes in Section 4. For conciseness, we present other issues in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm. How scalable is the probe schedule identified? Can *MPro* handle various scenarios including joins or post-filtering queries, as overviewed in Section 1? Can *MPro* extend to support approximation or parallelization? Our study in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, formally validates the scalability of *MPro* and develops how *MPro* can support various scenarios and extensions.

4 Experiments

This section reports our extensive experiments for studying the optimality and practicality of Algorithm *MPro*. Our experiments thus have a two-fold interpretation. First, to study the optimality, recall that, Algorithm *MPro* implements provably optimal object scheduling. To study this "lower bound" performance, we first isolate object scheduling and evaluate its performance over various configurations. Second, to study the practicality, we evaluate the effectiveness of our dynamic predicate scheduling over existing works for extensive and practical performance evaluations, respectively. In particular, we extensively compare our predicate scheduling framework to other existing schemes, over both synthetic and real-life data.

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 m_i probes for each p_i. In contrast, complete probing will require m probes (where m is the database size) for every probe predicate. The overall probe ratio (in percentages) is thus

$$pratio(Q) = \frac{\sum_{i=1}^{n} m_i}{nm},$$

which is the sum of the predicate probe ratio $pratio(p_i) = \frac{m_i}{nm}$, i.e., $pratio(Q) = pratio(p_1) + \cdots + pratio(p_n)$.

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 synthetic data as well as a "benchmark" database (with real-life data). For extensive study, Sections 4.1 and 4.2 report "simulations" over data synthesized with standard distributions. To understand the performance of *MPro* in real-world scenarios, we evaluate with a real-life real-estate database (essentially the same as our examples), which we report in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, due to the space limitation. All experiments were conducted with a Pentium 2.4GHz PC with 512MB RAM.

4.1 Object Scheduling Effectiveness

In this section, we study the lower-bound cost of an optimal object scheduling, over various configurations, contrasted with upper-bound cost of baseline approaches (i.e., $SortMerge$ and TAz,[4] as discussed in Section 2.2). Among three instantiations of *MPro* (Section 3.3), we use *MPro-Adaptive* in particular, as it performs best overall (and thus closer to the lower-bound performance) as

4. While TA_z has a potential to be further optimized by adding a right threshold condition, adding a right condition is nontrivial, which essentially boils down to two subproblems of 1) deciding when to stop and 2) deciding the order of the probes to perform. These are exactly what this paper studies.

parameter	default value	meaning
D	norm	score distribution
$\mathcal{F}$	min	scoring function
m	1k	database size
n	3	number of expensive predicates

Fig. 7. Evaluation parameters.

Section 4.2 will discuss. All queries in this section are of the form $\mathcal{F}(x, p_1, p_2, p_3)$ (as in our benchmark queries in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm). Our configurations are characterized by the following parameters: 1) Score distribution D: the distributions 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), 2) Scoring function $\mathcal{F}$: min, avg (average), and $gavg$ (geometric average), and 3) Database size m: 1k, 10k, 100k, and 1M. For the sake of convenience, Fig. 7 summarizes the evaluation parameters and their default values.

4.1.1 D: Score Distribution

Fig. 8a presents 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). The second distribution funif simulates "filtering" predicates, which are likely to be used in real-life queries that "filter out" a certain portion of data; e.g., the $large$ predicate in our benchmark queries (to be discussed in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm) disqualifies 78 percent of objects with zero scores (as their $size$s are out of the desired range). We thus define $\mathrm{funif}(f)$ (for *filtered uniform*) to simulate such predicates, where f percent of objects score 0 and the rest are uniformly distributed. The right figure (Fig. 8a) plots the cost for funif(75).

Observe that, in contrast to $SortMerge$ which requires the same complete probing cost regardless of retrieval size, TAz and *MPro* show desired "proportional" cost behavior by minimizing the objects evaluated. However, *MPro* significantly outperforms TAz by short-circuiting some probes on objects evaluated. Observe that filtering (the right figure) makes *MPro* even more effective—as final scores of the zero-scored objects, which comprises a major portion of data (i.e., $f = 75\%$), can be computed with partial evaluation (when their upper-bound reach the minimal-possible score 0, as discussed in Section 3.2).

4.1.2 $\mathcal{F}$: 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 (Fig. 8a) and some representative average functions (Fig. 8b), namely, arithmetic average $avg : (x + p_1 + p_2 + p_3)/4$ and geometric average $gavg := (x \cdot p_1 \cdot p_2 \cdot p_3)^{1/4}$.

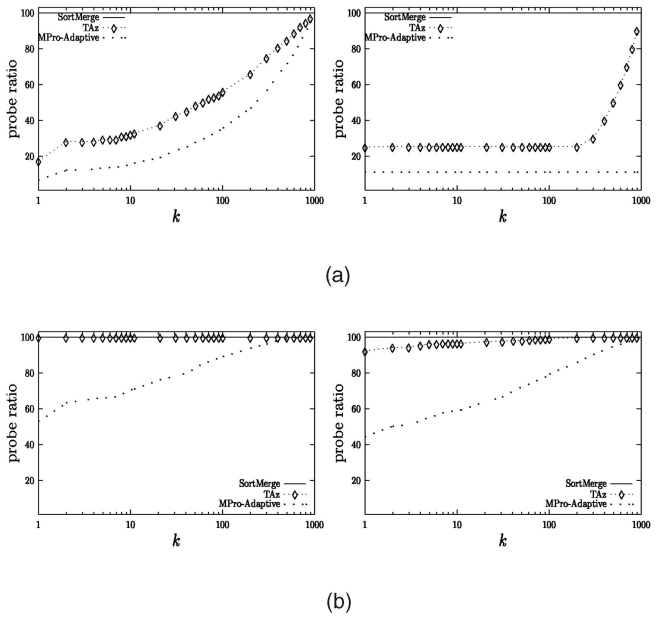


Fig. 8. Different score distributions (a) and scoring functions (b) with $m = 1\mathrm{k}$.

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.) Average functions are relatively less effective in filtering and thus adversely affect performance, especially that of TAz which degenerates close to complete probing. The two average functions perform similarly, with $\mathcal{F} = gavg$ being about 5 percent to 10 percent cheaper than avg.

4.1.3 m: Database Size

We evaluate the scalability of *MPro* over $m = 10\mathrm{k}$, 100k, and 1M when $\mathcal{F} = min$ and $D = \mathrm{norm}$, which we report in the Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm, due to the space limitation.

4.2 Predicate Scheduling Effectiveness

We then study the practicality of *MPro*. In particular, we evaluate the effectiveness of predicate scheduling schemes (i.e., $PSch$), compared to theoretical optimal schemes and existing per-object scheduling schemes. To experiment with various aspects, we generate 100 random configurations varying the following factors that affect predicate scheduling (as discussed in Section 3.3):

1. Scoring function $\mathcal{F}$. Though our schemes generally support any monotonic scoring function, to accommodate a comparison with existing schemes designed specifically for weighted average functions, we give favors by using weighted average functions for evaluation.[5] In particular, we set $\mathcal{F} = \frac{x + w_1 \cdot p_1 + w_2 \cdot p_2 + w_3 \cdot p_3}{1 + w_1 + w_2 + w_3}$ and generate 100 random weights (w_1, w_2, w_3), with w_i in [1:10].

5. Using weighted averages clearly favor Upper schemes, as we argue at the end of this section.

2. Predicates. In this experiment, we model predicate distributions of p_i as $\text{funif}(f_i)$, as such distributions have been practically observed in real benchmark queries (in Appendix, which can be found on the Computer Society Digital Library at http://computer.org/tkde/archives.htm), while they can also be easily synthesized with varying filtering ratio f_i: In particular, we generate 100 configurations of filtering ratios (f_1, f_2, f_3) and per-probe costs (C_1, C_2, C_3) of predicates (p_1, p_2, p_3), with f_i and C_i in [0:100](%) and [1:10], respectively.

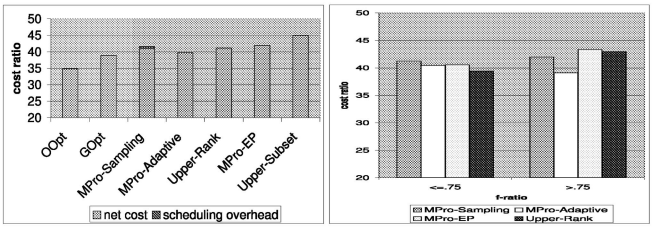
We quantify the effectiveness of predicate schedule $\mathcal{P}$, as the *average* cost of its corresponding optimal probe schedule $\mathcal{S} = (\mathcal{P}, \mathcal{O}_\mathcal{P})$, over various configurations discussed above. In particular, we compare the following predicate scheduling schemes.

4.2.1 Optimal Schedulers

To quantify how closely schedulers approximate the optimal scheduling, we consider global- and object-optimal schemes (denoted as *GOpt* and *OOpt*, respectively) which identify the optimal global and per-object predicate schedules. To guarantee the optimality, both schemes require complete knowledge of predicate scores: With such knowledge, *GOpt* can compute the exact selectivity, which enables the exact computation of the cost for the given global predicate schedule. *GOpt* can thus simply enumerate all $n!$ possible global predicate schedules and identify the minimal-cost one. Similarly, with complete knowledge, *OOpt* can exhaustively enumerate $n!$ possible subschedules to identify the optimal subschedule $\mathcal{P}^i$ of every object u_i, which compositionally construct complete predicate schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}m$, as discussed in Section 3.3. Note, these schemes are only of theoretical interest for comparison, as obtaining complete knowledge is infeasible.

4.2.2 Per-Object Schedulers

We consider existing per-object schemes—*Upper-Subset* [16] and *Upper-Rank* (its more recent extended version [17]), which specifically assumed weighted average scoring functions. Both schemes share the same object scheduling scheme as in *MPro*, but invoke predicate scheduler at each probe, i.e., mn invocations: At each probe, *Upper-Subset* determines the next predicate to evaluate, among the "candidate subset" of unevaluated predicates, by picking the one with highest $\frac{w_i}{C_i}$. Assuming each predicate p_i is expected to score e_i (which is typically set as 0.5), the candidate subset of object u is, among all subsets of its unevaluated predicates, the minimal-cost subset expected to be probed for qualifying (or disqualifying) u as a *top-k* answer. *Upper-Rank* later improves this scheme to further realize the benefit of per-object scheduling: Unlike *Upper-Subset* ordering predicates by global information $\frac{w_i}{C_i}$, *Upper-Rank* proposes a more sophisticated metric "rank" to leverage up-to-date per-object information in, not only in identifying candidate subset but also in ordering them.



(a) (b)

Fig. 9. Relative performance of MPro-Adaptive (to baseline). (a) Overall configuration. (b) Grouped.

4.2.3 Our Schedulers

We consider all three predicate scheduling schemes discussed in Section 3.3, namely, *MPro-Sampling*, *MPro-Adaptive*, and *MPro-EP*. For *MPro-Adaptive*, the only parameter we need to set is the sample size s, which we set as 0.1 percent of database size m. Note, for fair comparison, we design *MPro-Sampling* and *MPro-Adaptive* to use samples of the same size s. That is, predicate scheduling of *MPro-Adaptive* will be activated when s scores for every predicate have been collected. Also, given $\mathcal{F}$ as weighted average, we design *MPro-EP* to schedule predicates in the decreasing order of $\frac{w_i}{C_i}$.

To compare the effectiveness of these schemes, Fig. 9a compares the optimal probe costs of retrieving top $k = 100$ over $m = 10\text{k}$ objects, given the predicate schedules identified from the schemes above. In particular, as a cost metric, we extend probe ratio *cost ratio* (relative to complete probing) to take different per-probe cost C_i into consideration, i.e.,

$$\frac{\sum_{i=1}^{n} C_i m_i}{\sum_{i=1}^{n} C_i m}.$$

Our goals in comparison are three-fold: First, we want to quantify the theoretical bound of global scheme and evaluate how closely we approximate the bound. Second, similarly, we want to quantify how existing per-object schemes approximate the theoretical bound of per-object schemes. Last, we study the relative strengths of different schemes, by empirical analysis.

4.2.4 Global Scheduling Effectiveness

Observe that *MPro-Adaptive* closely approximates *GOpt* by 1 percent margin. Note that this margin is due to sampling inaccuracy. In principle, with perfect sampling, *MPro-Sampling* and *MPro-Adaptive* should be as good as *GOpt*, as our cost model for predicate scheduling (7) precisely models the actual probe cost.

4.2.5 Local Scheduling Effectiveness

Observe that, even though object-specific scheduling of *Upper-Subset* and *Upper-Rank* have potential to outperform *GOpt* and reach closer to *OOpt*, they fail to realize such potential, as it is nontrivial to obtain enough object-specific information. As a result, even a simplistic global scheduling of *MPro-EP* achieves comparable performance to *Upper-Rank*. Meanwhile, *Upper-Subset* and *Upper-Rank* have to pay

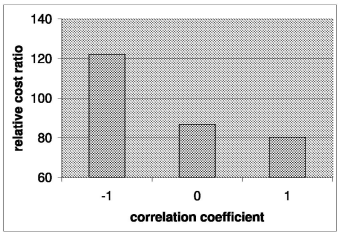


Fig. 10. Sensitivity of MPro-Adaptive over correlation (relative to MPro-Sampling).

extra mn-fold computational scheduling cost (as discussed in Section 3.3), which is not reflected by cost ratio metrics used in the figure.

4.2.6 Empirical Performance Analysis

We next empirically analyze relative strengths of scheduling schemes: For closer observation, we first divide our experiment configurations into two groups of the same size (i.e., 50 configurations each)—one group with rather similar predicate score distributions, i.e.,

$$f\text{-}ratio = \frac{max(f_i) - min(f_i)}{max(f_i)} \leq .75,$$

and the other with more contrasts, i.e., $f\text{-}ratio > .75$. Fig. 9b compares the average cost ratios of schemes in the two groups—observe that schemes relying solely on external parameters (i.e., *MPro-EP* and *Upper-Rank*) relatively suffer when score distributions differ significantly, as distributions start to dominate the scheduling decision, as reference [17] makes consistent observations. In contrast, observe such schemes are relatively strong when data distributions do not differ much (e.g., settings in Fig. 9a), and more desirably adhere to their assumption, e.g., uniform distribution with expected score 0.5, as reported in [17]. We next observe how *MPro-Sampling* and *MPro-Adaptive* compare: As observed in Figs. 9a and 9b, *MPro-Adaptive* outperforms *MPro-Sampling* in general, by overlapping scheduling overhead with necessary probes to perform anyway. However, note that, such overlap also introduces the bias of sampling toward the objects with high ceiling scores, which may produce different turnouts. Intuitively, if predicates have positive correlation, objects with high ceiling score are likely to score high for unevaluated predicates also. As a result, samples will be biased toward the objects with high overall scores, which are in fact objects of interest in *top-k* processing. In contrast, when predicates are negatively correlated, sampling will be biased toward uninteresting low-scored objects. Fig. 10 illustrates the performance implication of biases, by plotting the average probe cost of *MPro-Adaptive* (normalized to that of *MPro-Sampling* as 100 percent) over the same 100 configurations, when x and p_i's have perfect negative correlation (i.e., correlation coefficient $\rho = -1$), no correlation $(\rho = 0)$, or perfect positive correlation $(\rho = 1)$. As expected, with none or positive correlation, *MPro-Adaptive* outperforms *MPro-Sampling* by hiding scheduling overhead and biasing sampling toward objects of interest. However, negative correlation may offset the benefit of the overlap, by biasing samples toward low-scored objects.

In summary, our experimental results show that our predicate scheduling schemes closely approximate the theoretical lower bound cost. In contrast, while per-object schemes have the potential to outperform such bounds, existing schemes cannot realize such potential and only perform closely to global schemes with extra mn-fold computation overhead. Further, our extensive experiments reveal relative strengths of various schemes. Schemes considering predicate scores in scheduling (i.e., *MPro-Sampling* and *MPro-Adaptive*) generally justify their overhead with performance payoff, especially when distributions differ significantly across predicates. *MPro-Adaptive* further attempts to hide such overhead by overlapping the overhead of collecting predicate scores with actual probes, though it may bias sampling to undesirable objects when predicates are negatively correlated.

We conclude the section by stressing that experiments in this section favor *Upper-Subset* and *Upper-Rank* by 1) restricting experiments to scenarios they are designed for and 2) excluding their mn-fold per-probe computational overhead from the metric. In fact, it is not straightforward to extend these schemes for a general class of monotonic function. Marian et al. [17] propose an ad hoc extension for $\mathcal{F} = min$, yet with no w_i to rely on, the scheduling degenerates into a simplistic scheduling by C_i. In contrast, "principled" nature of our predicate scheduling (Section 3.3) enables general applicability over any arbitrary monotonic function.

5 Related Work

In the beginning, *top-k queries* have been actively studied for multimedia retrieval in the context of middlewares. Fagin [6], [18] first pioneered ranked queries and established the well-known *FA* algorithm. Algorithm *FA* performs a parallel sorted access for all predicates and completely evaluates the objects accessed (using random accesses), until at least k objects are seen from the all accesses, i.e., the k objects outscore the upper bound scores of objects yet to be accessed.

Soon after, many works have followed, which can be categorized in the following two categories: First, [14], [13], [12] followed to *FA* to be optimal in a stronger sense, by improving the stopping condition such that the upper bounds of the unseen objects can be more tightly computed. Second, [14], [16], [17], [19] then followed to propose algorithms for various access scenarios, beyond *FA* assuming the sorted accesses over all predicates. Recently, ranked retrieval research has been expanded, including, supporting ranking with arbitrary join conditions [20], probabilistic text retrieval [21], and peer-to-peer retrieval [22].

This paper falls into the second category of identifying a new scenario of supporting expensive predicates for ranked queries, providing unified abstraction for user-defined functions, external predicates, postfiltering predicates, and fuzzy joins. Concurrently with our preliminary work [5], some related efforts emerge to support expensive external predicates [16] (and its extended version [17]), with specific focus to external Web-based predicates in the form of weighted average functions. These works and our framework similarly adopt the branch-and-bound or best-first

search techniques [23]; however, our work clearly distinguishes itself in the following several aspects:

First, we aim at a different or more general problem of supporting general expensive predicates over arbitrary monotonic functions, unlike [16], [17] which support only weighted average functions. A key challenge in extending ranking function for arbitrary monotonic function comes from identifying an ideal scheduling scheme for the arbitrary given function, while [16], [17] develop a scheduling scheme by cost-weight ratio which is fundamentally restricted to weighted average functions. Second, our framework models *probe minimization* as probe scheduling optimization which decouples into *object scheduling* (finding objects to probe) from *predicate scheduling* (finding predicates to execute). In particular, we develop provably optimal object scheduling scheme, together with dynamic predicate scheduling schemes. Bruno et al. [16], [17] employ the same intuition in object scheduling and capture the insight of predicate scheduling with heuristics of maximizing "expected score decrease" for weighted scoring functions. In contrast, we believe our systematic optimization integrating object and predicate scheduling enables the formal foundation of probe-optimality and achieves generality to support arbitrary monotonic scoring functions. Third, while our framework can generally deal with both per-object and per-query predicate scheduling, we choose to focus on the global strategy unlike the per-object strategies proposed in [16], [17]. We analytically and empirically argue that the overhead of per-object scheduling generally offsets its benefit in Section 3.3 and Section 4.

This paper focuses on probe-only scenarios and, thus, provides a foundation for [19] uniformly supporting any arbitrary *top-k* access scenarios. A key difference is that [5] focuses on a probe-only scenario, which is a subset of arbitrary *top-k* scenarios supported by [19] and thus serves as the foundations of supporting arbitrary scenarios in the following two dimensions: First, the necessary-probe principle provides an analytic foundation for scheduling arbitrary accesses in [19]. Second, our predicate scheduling schemes developed in Section 3.3 is adopted in [19] in scheduling random accesses.

In summary, this paper is first to generally support expensive predicates. Further, we decouple probe scheduling into object and predicate scheduling, which together form a probe schedule. Our systematic development of schedule optimization, decoupled into object and predicate scheduling, provides a theoretical justification for the techniques proposed in our preliminary work [5]. Our proposed algorithm is thus optimal in principle, by analytically identifying a provably optimal object schedule with respect to which a given predicate schedule identified by a dynamic predicate scheduling optimization.

6 Conclusion

We have presented our framework and algorithms for evaluating ranked queries with expensive probe predicate. We identified that supporting probe predicates is essential, in order to incorporate user-defined functions, external predicates, post-filtering predicates, and fuzzy joins. Our work generally addresses supporting expensive predicates for ranked queries: More specifically, we enable *probe minimization* by a cost-based optimization over the space of probe schedules. In particular, we decouple probe scheduling into object and predicate scheduling, which together form a probe schedule. We also develop Algorithm *MPro*, which analytically determines the optimal object schedule with respect to a given predicate schedule. We develop an effective dynamic predicate scheduling scheme to efficiently identify a cost-effective predicate schedule.

We have implemented the algorithm described in this paper, based on which we performed extensive experiments with both real-life databases and synthetic data sets. Our experimental study validates the optimality of our analytic object scheduling as "lower bound" performances and the effectiveness of our dynamic predicate scheduling over the existing schemes.

Acknowledgments

The authors thank Divyakant Agrawal and Wen-Syan Li for their fruitful discussions during one of the authors' summer visit at NEC USA CCRL, which inspired us to pursue this work. They also thank Amelie Marian and Luis Gravano for helping with the implementation of their *Upper-Subset* and *Upper-Rank* algorithms. This paper is based on and significantly extends our preliminary works "Minimal Probing: Supporting Expensive Predicates for Top-*k* Queries" in the Proceedings of the ACM SIGMOD 2002.

References

[1] R. Agrawal and E. Wimmers, "A Framework for Expressing and Combining Preferences," *Proc. ACM SIGMOD,* 2000.
[2] J.M. Hellerstein and M. Stonebraker, "Predicate Migration: Optimizing Queries with Expensive Predicates," *Proc. ACM SIGMOD,* 1993.
[3] A. Kemper, G. Moerkotte, K. Peithner, and M. Steinbrunn, "Optimizing Disjunctive Queries with Expensive Predicates," *Proc. ACM SIGMOD,* 1994.
[4] C. Li, K.C. Chang, I.F. Ilyas, and S. Song, "RankSQL: Query Algebra and Optimization for Relational Top-k Queries," *Proc. ACM SIGMOD,* 2005.
[5] K.C. Chang and S.-W. Hwang, "Minimal Probing: Supporting Expensive Predicates for Top-k Queries," *Proc. ACM SIGMOD,* 2002.
[6] R. Fagin, "Combining Fuzzy Information from Multiple Systems," *Proc. ACM Symp. Principles of Database Systems,* 1996.
[7] L.A. Zadeh, "Fuzzy Sets," *Information and Control,* vol. 8, 1965.
[8] G. Salton, *Automatic Text Processing.* Addison-Wesley, 1989.
[9] S. Chaudhuri and L. Gravano, "Optimizing Queries over Multimedia Repositories," *Proc. ACM SIGMOD,* 1996.
[10] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k Selection Queries over Relational Databases: Mapping Strategies and Performance Evaluation," *ACM Trans. Database Systems,* 2002.
[11] V. Hristidis and Y. Papakonstantinou, "Algorithms and Applications for Answering Ranked Queries Using Ranked Vies," *VLDB J.,* 2004.
[12] S. Nepal and M.V. Ramakrishna, "Query Processing Issues in Image (Multimedia) Databases," *Proc. Int'l Conf. Data Eng.,* 1999.
[13] U. Guentzer, W. Balke, and W. Kiessling, "Optimizing Multi-Feature Queries in Image Databases," *Proc. 26th Int'l Conf. Very Large Data Bases,* 2000.
[14] R. Fagin, A. Lotem, and M. Naor, "Optimal Aggregation Algorithms for Middleware," *J. Computer and System Sciences,* vol. 66, no. 4, 2003.
[15] T.H. Cormen, C.E. Leiserson, and R.L. Rivest, *Introduction to Algorithms.* MIT Press, 2001.
[16] N. Bruno, L. Gravano, and A. Marian, "Evaluating Top-k Queries over Web-Accessible Databases," *Proc. Int'l Conf. Data Eng.,* 2002.

[17] A. Marian, N. Bruno, and L. Gravano, "Evaluating Top-k Queries over Web-Accessible Databases," *ACM Trans. Database Systems,* 2004.
[18] E.L. Wimmers, L.M. Haas, M.T. Roth, and C. Braendli, "Using Fagin's Algorithm for Merging Ranked Results in Multimedia Middleware," *Proc. Int'l Conf. Cooperative Information Systems,* 1999.
[19] S.-W. Hwang and K.C.-C. Chang, "Optimizing Access Cost for Top-k Queries over Web Sources: A Unified Cost-Based Approach," *Proc. Int'l Conf. Data Eng.,* 2005.
[20] A. Natsev, Y.-C. Chang, J.R. Smith, C.-S. Li, and J.S. Vitter, "Supporting Incremental Join Queries on Ranked Inputs," *Proc. 27th Int'l Conf. Very Large Data Bases,* 2001.
[21] M. Theobald, G. Weikum, and R. Schenkel, "Top-k Query Evaluation with Probabilistic Guarantees," *Proc. 30th Int'l Conf. Very Large Data Bases,* 2004.
[22] P. Cao and Z. Wang, "Efficient Top-k Query Calculation in Distributed Networks," *Proc. 23rd Ann. ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing (PODC '04),* 2004.
[23] S.J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach.* Prentice Hall, 1994.

Seung-won Hwang is an assistant professor in the Department of Computer Science and Engineering at POSTECH, Korea.

Kevin Chen-Chuan Chang is an assistant professor in the Computer Science Department at the University of Illinois at Urbana-Champaign.