Optimizing Top-k Queries for Middleware Access: A Unified Cost-Based Approach

SEUNG-WON HWANG
Pohang University of Science and Technology
and
KEVIN CHEN-CHUAN CHANG
University of Illinois at Urbana-Champaign

This article studies optimizing top-k queries in middlewares. While many assorted algorithms have been proposed, none is generally applicable to a wide range of possible scenarios. Existing algorithms lack both the "generality" to support a wide range of access scenarios and the systematic "adaptivity" to account for runtime specifics. To fulfill this critical lacking, we aim at taking a cost-based optimization approach: By runtime search over a space of algorithms, cost-based optimization is *general* across a wide range of access scenarios, yet *adaptive* to the specific access costs at runtime. While such optimization has been taken for granted for relational queries from early on, it has been clearly lacking for ranked queries. In this article, we thus identify and address the barriers of realizing such a unified framework. As the first barrier, we need to define a "comprehensive" space encompassing all possibly optimal algorithms to search over. As the second barrier and a conflicting goal, such a space should also be "focused" enough to enable efficient search. For SQL queries that are explicitly composed of relational operators, such a space, by definition, consists of schedules of relational operators (or "query plans"). In contrast, top-k queries do not have *logical tasks*, such as relational operators. We thus define the logical tasks of top-k queries as building blocks to identify a comprehensive and focused space for top-k queries. We then develop efficient search schemes over such space for identifying the optimal algorithm. Our study indicates that our framework not only unifies, but also outperforms existing algorithms specifically designed for their scenarios.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Retrieval models*; H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Performance evaluation (efficiency and effective)*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Top-k query processing, middlewares

Authors' addresses: S.-W. Hwang, Department of Computer Science and Engineering, Pohang University of Science and Technology, San 31, Hyoja Dong, Nam Gu, Pohang, Gyungbuk, Korea 790-784; email: swhwang@postech.ac.kr; K. C.-C. Chang, Computer Science Department, University of Illinois at Urbana-Champaign, 306 Engineering Hall, MC 266, 1308 West Green Street, Urbana, IL 61801; email: kcchang@cs.uiuc.edu.

ACM Reference Format:
Hwang, S.-W. and Chang, K. C.-C. 2007. Optimizing top-*k* queries for middleware access: A unified cost-based approach. ACM Trans. Database Syst. 32, 1, Article 5 (Mar. 2007), 41 pages. DOI = 10.1145/1206049.1206054 http://doi.acm.org/10.1145/ 1206049.1206054

1. INTRODUCTION

To enable nontraditional *fuzzy retrieval* which naturally arises in many new applications, *top-k* (or *ranked*) queries are crucial for matching data by "soft" conditions. A top-k query selects the k top answers among a database of n data objects, each of which evaluates m soft *predicates* $p_1, \ldots, p_m$ to scores in [0:1] to be aggregated by some *scoring function* $\mathcal{F}$ (e.g., *min*). In particular, this article focuses on top-k queries in a middleware system, that is, a *middleware* processes queries over subsystems (e.g., multimedia subsystems [Fagin 1996]) or external systems (e.g., web sources [Bruno et al. 2002]), which we will generally refer to as *sources*. For such middleware querying scenarios, due to the inherent "data retrieval" nature of retrieving and combining data from multiple sources, top-k queries have emerged to be of particular importance.

For top-k queries in such settings, a middleware relies on *accessing* sources for query processing. Since a middleware cannot manipulate data directly, it must use some access methods (for finding objects and their scores) supported by sources to gather predicate scores. For access *methods*, a source may support for each predicate p_i: (1) *sorted access*, which returns object scores in descending order, one in each access, or (2) *random* access, which returns the score for a given object. As a main motivation of this article, in a middleware setting, such accesses are typically expensive (compared to local computation) with varying latencies, which we denote as cs_i and cr_i for sorted and random access, respectively, for each predicate p_i. To illustrate a concrete real scenario, consider a travel agent scenario over the web middleware sources in Example 1 (which will be used as benchmark queries, as well for experiments in Section 9):

Example 1. A user may ask a ranked query to find the top-5 restaurants (say, in the Chicago area) that are highly rated and close to the user's preferred location `"myaddr"`, as Q_1 illustrates (in SQL-like syntax):

select name **from** `restaurant r`
order by *min*(p_1 : *rating*(r.stars), p_2 : *close*(r.addr, `myaddr`))
stop after 5 (**Query** Q_1)

To answer this query, our middleware will access web sources to evaluate predicate p_1 and p_2. Figure 1(a) shows one possible scenario: For evaluating *close*: superpages.com is capable of (1) returning the *close* score for a specific restaurant (i.e., random access) and (2) returning restaurants in their descending order of scores (i.e., sorted access). For *rating*: dineme.com similarly provides both sorted and random accesses.[1]

[1]While these sources may support a multidimensional sorted access, for example, sorted by *price* + *location*, such access is ordered by a fixed and implicit combining function. As it is not clear

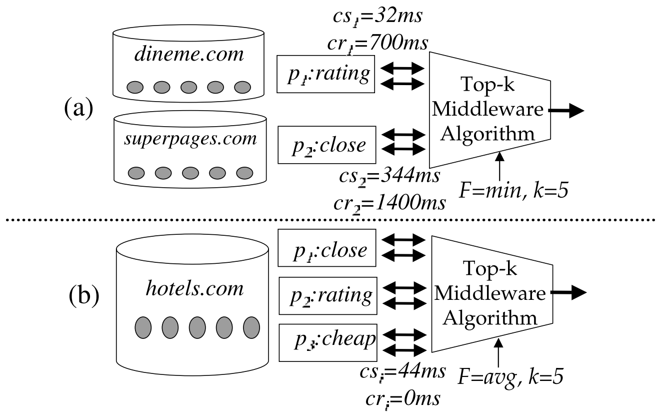


Fig. 1. Scenarios for (a) Q_1 and (b) Q_2.

The middleware will coordinate these accesses to find the top results. To characterize this particular scenario, Figure 1(a) shows the average access latency for each predicate p_i: In this scenario, random accesses are more expensive in both sources (i.e., $cr_i > cs_i$), but with varying scales (i.e., $cr_1 = 700ms = \frac{1}{2} \cdot cr_2$) and ratios (i.e., $\frac{cr_1}{cs_1} = \frac{700ms}{32ms} \sim 22; \frac{cr_2}{cs_2} = \frac{1400ms}{344ms} \sim 4$).

Access scenarios can vary widely, depending on the sources involved, due to source heterogeneity: By contrast, consider another scenario in which our middleware now works with a different source, for example, hotel.com, to answer query Q_2:

select name **from** `hotel` h
order by $avg(p_1 : close($h.addr, `myaddr`$), p_2 : rating($h.stars$),$
$p_3 : cheap($h.price$))$
stop after 5 (**Query** Q_2)

In this setting, since a sorted access (e.g., for *close*) on a hotel object also retrieves its other attributes (e.g., stars and price), the subsequent random accesses[2] incur zero access costs, for example, by locally computing *rating* and *cheap* on stars and price already retrieved. We note that this scenario of expensive sorted accesses significantly contrasts with that of expensive random accesses in Figure 1(a).

Existing algorithms. To support these top-*k* queries in middlewares, many algorithms have been proposed for various cost scenarios. Figure 2 summarizes a "matrix" of access scenarios that have been studied, each characterized by how sources relatively support each type of access, such as, *cheap* ($cost = 1$), *expensive* ($= h$), or *impossible* ($= \infty$). As the matrix summarizes, existing algorithms are designed with a specific cost scenario in mind. For instance, a pioneering

how to leverage this to support arbitrary ranking functions, we focus on leveraging one-dimensional accesses, as do other middleware top-*k* algorithms.

[2]In a middleware, random accesses to an object h can only occur after h is first seen from sorted accesses, or "no wild guess" [Fagin et al. 2001].

	Random Access		
Sorted Access	cheap: $cr_i = 1$	expensive: $cr_i = h$	impossible: $cr_i = \infty$
cheap: $cs_i = 1$	FA, TA, Quick-Combine	CA, SR-Combine	NRA Stream-Combine
expensive: $cs_i = h$	none	FA, TA, Quick-Combine	
impossible: $cs_i = \infty$	TA$_Z$, MPro, Upper		

Fig. 2. Access scenarios and algorithms.

and influential existing algorithm TA [Fagin et al. 2001] is intended for scenarios where sorted and random access costs are comparable, while Algorithm MPro [Chang and Hwang 2002] is specifically designed for scenarios where sorted access is impossible.[3] While these algorithms are appropriate for their specific scenarios, we are clearly lacking "generality" such that an algorithm is not generally applicable to real-life scenarios where sources differ in cost characteristics (e.g., random access is inexpensive in some source, while impossible in another) which may even change over time (e.g., depending on server loads). Second, existing algorithms generally lack runtime "adaptivity" such that an algorithm typically cannot adapt to the specific scenario at hand, except for some limited attempts with rather ad hoc heuristic-based adaptation (as Section 2 discusses). In contrast, as we will argue to follow, a systematic adaptation to runtime specifics can make a significant difference in performance.

Our approach. To fulfill this critical lack of generality and adaptivity, this article aims at a *general* framework over such various scenarios that is *adaptive* to the given scenario so as to minimize *access costs*. We note that such access costs (much like I/O in relational DBMS) dominate the overall query processing in a middleware context, and thus their minimization is critical for query efficiency. For this objective, we first model the cost of algorithm $\mathcal{M}$ as the costs of all access, parameterized by cs_i and cr_i, which vary over cost scenarios at runtime (e.g., Figure 1). Such an aggregate cost model is rather standard in many top-k querying settings (e.g., Fagin et al. [2001]). In other words, given some algorithm $\mathcal{M}$, and letting. S_i and R_i be the number of sorted and random accesses, respectively, for predicate p_i, the total cost is

$$\mathcal{C}(\mathcal{M}) = \sum_i S_i \cdot cs_i + R_i \cdot cr_i. \qquad (1)$$

In particular, as the main thesis of this article, we propose a systematic and unified "cost-based optimization" framework which will adapt to various cost scenarios. Specifically, given a specific setting of cost parameters cr_i and cs_i (e.g., Figure 1 represents two different settings), we want to generate an algorithm $\mathcal{M}$ that minimizes the cost for the given setting. While cost-based optimization has been taken for granted for relational queries from early on [Selinger et al. 1979], it has been clearly lacking for top-k queries.

[3]We note that our approach does not make the "no wild guess" assumption and thus works both with and without constraint, as discussed in Section 8.

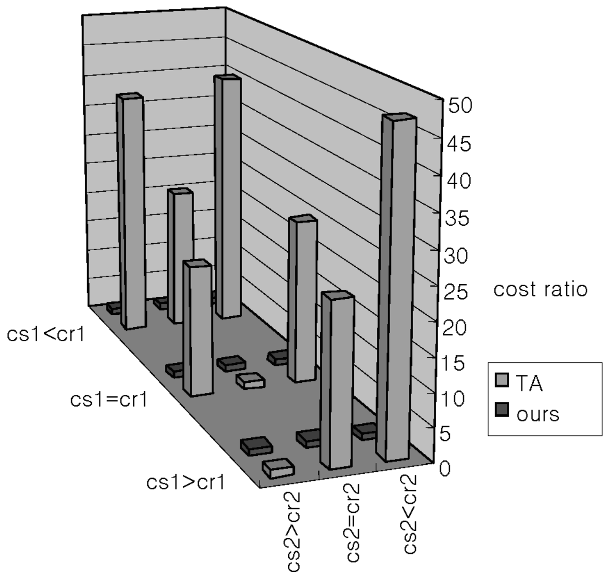


Fig. 3. Adaptivity over costs.

Our cost-based optimization framework complements existing algorithms with its *generality* and *adaptivity*: First, our framework aims at providing a unified framework that is general over any arbitrary scenario. We prove the generality by showing that any possible algorithm has a counterpart algorithm generated by our framework, which performs the same job with no more access (Theorem 4). Our framework, being general, not only generates the behaviors of the existing algorithm (as we further discuss in Section 8), but also handles unstudied scenarios. One example of such unstudied scenarios is illustrated in Figure 1(b), in which sorted accesses (cost 44ms per access) are more expensive than random accesses (zero cost), namely, corresponding to the "none" cell in Figure 2.

Second, in terms of *performance*, our framework is adaptive and thus enables a potentially significant speedup from existing algorithms by a systematic adaptation to runtime specifics. As an illustration, Figure 3 summarizes our evaluation results (reported in detail in Section 9, Figure 18(b), comparing the cost of the existing algorithm TA and ours over a systematic enumeration of a wide range of cost scenarios. Observe that our framework (by runtime adaptation) is robust over a wide range of scenarios and significantly outperforms TA by orders of magnitude. This cost difference results from our ability to adapt to runtime parameters, such as query size or varying access costs of predicates.

Considering all these benefits of a unified framework, we now investigate and address the major barriers of realizing cost-based optimization for top-*k* queries. As the first barrier, we need to define a "comprehensive" space, which we denote as Ω, encompassing all possibly optimal top-*k* algorithms to search over. Second, as a conflicting goal, such Ω should also be "focused" enough to enable efficient search. For SQL queries, since a space is explicitly composed of relational operators (e.g., joins and selections), an algorithm space Ω is comprised of all query plans of such operators that are algebraically equivalent (e.g., by their commutativity and associativity). However, what is a "query plan"[4] for a top-k query? It is not obvious how a top-k query, as an arbitrary scoring function, such as $\mathcal{F}=\min(p_1, p_2)$, can be decomposed into *logical tasks*, analogously to relational queries.

To the best of our knowledge, our work is the first to realize systematic cost-based optimization for top-k queries by overcoming these dual barriers. First, to define a *comprehensive* space, we show that abstracting a top-k algorithm as an access scheduling problem enables us to define a comprehensive space Ω (Section 4). Second, to define a *focused* space, we develop an insight that the query processing can focus, without compromising any generality, only on *necessary choices* (Section 5). Based on this development, we define a comprehensive and focused algorithm space Ω (Section 6). We then further reduce the search space to reduce the overhead of finding the optimal algorithm in this space. In particular, we adopt systematic space reduction schemes, such as focusing on algorithms performing sorted access first and performing global random access scheduling, which we show reduces the space without significantly compromising comprehensiveness (Section 7). With this reduced space Ω, cost-based optimization is to identify the optimal algorithm $\mathcal{M}_{\text{opt}}$ in Ω with respect to the cost model $\mathcal{C}$, that is

$$\mathcal{M}_{\text{opt}} = argmin_{\mathcal{M}\in\Omega}\mathcal{C}(\mathcal{M}). \tag{2}$$

This article is based on and extends the "necessary-probe principle" studied in our preliminary work [Chang and Hwang 2002] (similar heuristics, but specific to *weighted sum* scoring functions, were also studied in Bruno et al. [2002]). However, as our preliminary work focuses on specific scenarios where random access cost dominates, it focuses only on scheduling random accesses. In a clear contrast, we generalize the techniques to arbitrary accesses (i.e., sorted and random access) and thus achieve general applicability to any top-k scenario. As we will discuss in Section 8, such generalization in fact requires considerable extensions, as sorted access fundamentally differs from random access and thus significantly complicates optimization.

We extensively validate the practicality and generality of our framework using both real-life sources (using our travel agent benchmark scenarios) and synthesized arbitrary middleware scenarios. The results are indeed encouraging; our framework not only unifies, but also outperforms the existing algorithms specifically designed for respective scenarios.

In summary, we highlight our contributions as follows:

—We define a comprehensive and focused *algorithm space* for top-k queries as an essential foundation for cost-based optimization.

[4]We view our top-k retrieval as a "query" with a "query plan"—in a different view, we may think of ranking as part of a relational query, thus corresponding to only an "operator." To clarify, we stress that we focus on middleware scenarios where a ranking query itself is a complete query specified by a scoring function $\mathcal{F}$ and retrieval size k. Thus, we view top-k optimization as identifying the optimal plan of scheduling various accesses as operators.

—We develop *runtime optimization schemes* for searching over the space to find a cost-minimal algorithm.

—We realize a *conceptual unification* of existing algorithms. Our framework unifies and generalizes beyond existing algorithms.

—We report *experimental evaluation* using both real-life and synthetic scenarios to validate the generality and adaptivity of our framework.

2. RELATED WORK

As overviewed in Section 1, many algorithms have been proposed to support top-*k* queries for various cost scenarios, as summarized in Figure 2. In particular, Fagin pioneered with Algorithm FA [Fagin 1996] for scenarios where random and sorted accesses are supported with uniform cost (the diagonal cells in Figure 2). Reference [Fagin et al. 2001] then followed to propose a suite of algorithms for various access scenarios with a stronger sense of optimality, such as TA (for uniform-cost scenarios), NRA (when random access is impossible), and TA_Z (when sorted access is impossible).

While these algorithms only follow statically designed behaviors and thus do not adapt to runtime access costs, CA [Fagin et al. 2001], SR-Combine [Balke et al. 2002], Quick-Combine [Guentzer et al. 2000], and Stream-Combine [Guentzer et al. 2001] attempt limited runtime optimization. In particular, a representative algorithm CA [Fagin et al. 2001] alternates sorted and random access phases according to a runtime cost parameter $h = cr_i/cs_i$. More specifically, unlike TA, which alternates a sorted access phase and random access phase, CA alternates h sorted access phases and then a random access phase so as to use more sorted accesses (which is h times less expensive) over random accesses. However, such heuristics-based optimization has limited applicability: Algorithm CA assumes h to be the same for all predicates, while such an assumption is unlikely to hold in practice: Across autonomous sources, this ratio may vary (for Figure 1(a)) or can be zero (for 1(b)). Similarly, Algorithms SR-Combine, Quick-Combine, and Stream-Combine use the partial derivative of scoring functions as an indicator to optimize, which restricts their applicability to differentiable ranking functions.

In comparison, by a systematic "cost-based" optimization, our framework complements the current matrix of existing algorithms: (1) By enumerating a comprehensive space of algorithms, our framework not only unifies existing algorithms, but also extends to scenarios which current algorithms have not covered (e.g., "none" cell in Figure 2 and more scenarios not described by the figure); (2) by runtime search over such space, our cost-based optimization systematically optimizes with respect to runtime parameters, unlike existing algorithms that have rather limited and partial adaptation.

Our framework extends and generalizes our preliminary work MPro [Chang and Hwang 2002]. In particular, we extend MPro from focusing only on scheduling random accesses into a complete framework with general applicability to *any* top-*k* scenario. Section 8 will further discuss the implication of such extension.

Meanwhile, we note that Upper [Bruno et al. 2002] has also developed similar heuristics for the same expensive random access scenarios as MPro. However, in addition to the same limited focus on only random accesses, their runtime adaptation heuristics specifically assume *weighted average* scoring functions. In contrast, we propose a more general adaptation framework which enables generalizing Upper not only so as to schedule arbitrary (random and sorted) accesses, but also to support arbitrary monotonic ranking functions.

Finally, ranked queries have also been studied for relational databases: References [Carey and Kossmann 1997, 1998] present optimization techniques for exploiting the limited cardinalities of ranked queries. References [Chaudhuri and Gravano 1999; Donjerkovic and Ramakrishnan 1999] then propose to exploit probabilistic distributions and histograms, respectively, to process rank queries as equivalent Boolean selections.

3. MOTIVATION

While assorted algorithms have been proposed for supporting top-k queries, as summarized by the matrix over various access scenarios (in Figure 2), the current matrix lacks in many aspects: In addition to lacking in terms of generality and adaptivity, as Section 1 discussed, the current matrix lacks conceptual *unification*. While these assorted algorithms, as designed for different scenarios, naturally behave differently, they seem to share some subtle similarities. For example, they keep retrieved objects in order and terminate at a threshold condition. Such resemblance makes us naturally wonder if they can be subsumed by a unified framework: This framework will complement existing works by combining them into a single one-fits-all implementation, while providing insights on how to support the access scenarios yet to be studied.

Our goal is thus to propose a general and unified runtime optimization. We stress that, in contrast to our runtime optimization, most current algorithms are provided with a static (by design) guarantee of *instance optimality* [Fagin et al. 2001]: An algorithm B is optimal over a class of algorithms $\mathbf{A}$ and class of databases $\mathbf{D}$ if for every $A \in \mathbf{A}$ and $D \in \mathbf{D}$, $\mathcal{C}(B, D) = O(\mathcal{C}(A, D))$ for a chosen cost function $\mathcal{C}$. Thus, it allows an *optimality ratio* c such that $\mathcal{C}(B, D) \leq c \cdot \mathcal{C}(A, D) + c'$ as a tolerable cost distance to "absolute optimality." In the lack of runtime optimization, such a static guarantee has prevailed in previous top-k works, as it provides a rather strong optimality guarantee, that is, over any algorithm A and data instance D. However, we stress that the optimality ratio c is not a constant, but varies over problem sizes. For instance, according to Fagin et al. [2001], c can be up to $m(m-1) + m\frac{cr_i}{cs_i}$ for TA, which varies over cost ratio $\frac{cr_i}{cs_i}$ and query size m. As cost ratios and query sizes typically vary in practice, the distance c in many cases is *not* a constant to be ignored in optimization: First, by ignoring access cost *r*atios in optimization, namely, assuming that $\forall i, j : \frac{cs_i}{cs_j} = \frac{cr_i}{cr_j} = 1$, the optimality is meaningful only to limited uniform-cost scenarios. However, actual application scenarios are unlikely to be uniform, especially over autonomous middleware sources (e.g., $\frac{cr_2}{cr_1} = 20$ in Figure 1(a). Second, by ignoring *query* size in optimization, or the number of predicates, the optimality is reduced only with respect to *database* size. For processing a top-k query (as well as traditional Boolean queries), the *problem* size of answering Q over database D is characterized by both query size (i.e., $|Q| = m$) and database size (i.e., $|D| = n$). However, instance optimality assumes m as a constant, which reduces the optimality only with respect to the number of objects evaluated, regardless of the predicates each object evaluates (by contrast, Boolean query optimizers, e.g., Hellerstein and Stonebraker [1993], mainly strive to minimize such predicate evaluation costs). By systematically adapting to all these runtime cost parameters, our framework aims at generally optimizing virtually all access scenarios.

Our goal is thus to develop an optimization framework that conceptually unifies existing algorithms. For such a framework, we take a cost-based optimization approach: Our first task is to define the algorithm space Ω (Eq. (2)) to search over. As motivated in Section 1, such a space must be comprehensive space for top-k queries. Section 4 defines a comprehensive space for top-k queries by abstracting a top-k algorithm as an access scheduling problem. We then study, to define a focused space, how to decompose a top-k query into logical tasks so as to narrow down the space, just as we enumerate a comprehensive and focused space for SQL queries by enumerating all query plans of logical relational operators. Section 5 identifies the required information for answering a top-k query and decomposes it into logical tasks, on the basis of which we define a comprehensive and focused algorithm space for top-k queries in Section 6.

4. DEFINING A COMPREHENSIVE SPACE

This section now tackles the first challenge of defining a comprehensive space that encompasses all possible algorithms. To understand a top-k algorithm constituting such a space, we begin with considering an example algorithm (as Example 2 will show). As our running example query, we will consider Q_1 (from Example 1) for finding the top restaurant, thus setting the retrieval size to $k = 1$. For our illustration, assume that Dataset 1 (Figure 5) represents our example restaurant "objects" (i.e., u_1, u_2, and u_3) and their scores (which can only be known by accessing the sources). Given Q_1 as an input, top-k algorithms will return $\mathcal{K} = \{u_3{:}0.7\}$ as an answer, that is, u_3 is the top-ranked object with score $\mathcal{F}[u_3]$=0.7.

Recall that, as Section 1 mentioned, this article focuses on middleware algorithms. Since a middleware cannot manipulate data directly, it relies on access methods supported by sources: (1) sorted access on predicate p_i, denoted by sa_i; or (2) random access on predicate p_i for object u_j, denoted $ra_i(u_j)$. To contrast, we note that the two types of accesses differ fundamentally in two respects:

—*Side-effects*: Sorted access sa_i has side-effects. To illustrate, in Figure 5, the first sa_1 not only evaluates $p_1[u_3]$=.7 with the highest p_1 score, but also *bounds* the "maximal-possible" score of p_1 for every "unseen" objects, for example, u_1 and u_2, with this *last-seen* score, for example, $p_1[u_1] \leq .7$. In contrast, random access $ra_i(u_j)$ has no effect on objects other than u_j itself.

—*Progressiveness*: Sorted access sa_i is progressive in that repeated accesses give more information. For instance, repeated sa_1 evaluates u_3, u_1, and u_2

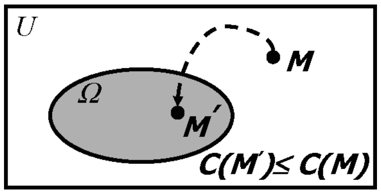


Fig. 4. Cost comprehensiveness.

in turn, as accessing *deeper* into p_1's sorted list. Meanwhile, $ra_i(u_j)$ returns $p_i[u_j]$ every time and thus need not be repeated.

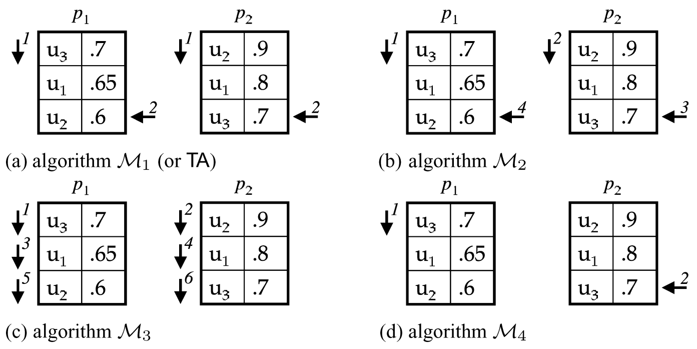
To answer a query, an algorithm performs accesses so as to gather necessary scores. To illustrate how this works, Figure 6 illustrates example algorithms using a sorted list for each predicate. The objects in each list are ordered by corresponding predicate scores, as illustrated in Figure 6(a). On the sorted list, we will use $\downarrow$ and $\leftarrow$ to represent the sorted and random accesses performed, respectively, annotated by their time of access. For instance, $\leftarrow_2$ in Figure 6(a) on p_2 represents that Algorithm $\mathcal{M}_1$ performs a random access at time 2, for p_2 on the pointed object u_3.

Example 2 (*Example Algorithm*). We illustrate, as Figure 6(a) shows, how Algorithm TA [Fagin et al. 2001], a representative algorithm, processes Q_1: At time 1, it performs sorted accesses sa_1 and sa_2 in parallel (as represented by $\downarrow_1$), which evaluate $p_1[u_3]$=.7 and $p_2[u_2]$=.9 with the highest p_1 and p_2 score, respectively. At time 2, it computes the final scores of the objects just seen, namely, u_3 and u_2, by performing random accesses $ra_2(u_3)$ and $ra_1(u_2)$ (as represented by $\leftarrow_2$). The algorithm can now terminate, as the final score of u_3, namely, $\mathcal{F}[u_3] = \min(0.7, 0.7) = 0.7$, is no less than that of the "unseen" object (i.e., u_1): As $p_1[u_1]$ is bounded by the "side-effect," that is, $p_1[u_1] \leq .7$, $\mathcal{F}[u_1] = \min(p_1[u_1], p_2[u_1])$ cannot be higher than $\mathcal{F}[u_3] = .7$.

Example 2 shows one possible algorithm (i.e., $\mathcal{M}_1$ as TA). However, there can be many more algorithms which differ in the accesses they perform. To illustrate, consider the example algorithms answering Q_1 in Figure 6: $\mathcal{M}_2$ performs the same accesses as TA, but one-at-a-time, and $\mathcal{M}_3$ evaluates exhaustively using sorted accesses. Different algorithms, by performing different accesses in various orders, incur different costs. For instance, $\mathcal{M}_4$ can answer the same query, performing only a part of the accesses that Algorithm TA performs (as we will see in Example 6). Our goal is thus to identify the cost-optimal algorithm among many possible algorithms. Hence, we must guarantee that the algorithm space includes the optimal algorithm. For this, we first formalize the notion of *cost comprehensiveness* (as Figure 4 illustrates): While a space Ω may not encompass the "universe" $\mathcal{U}$ of all possible algorithms, it is comprehensive with respect to some cost function if any arbitrary algorithm $\mathcal{M}$ in $\mathcal{U}$ can find its "counterpart" $\mathcal{M}'$ in the Ω that answers the same query with *no more* cost. Such Ω is thus comprehensive enough for optimization; no algorithms outside of the space can be better than those inside, since its counterpart $\mathcal{M}'$ is at least as

OID	p_1	p_2	$\mathcal{F}$
u_1	0.65	0.8	0.65
u_2	0.6	0.9	0.6
u_3	0.7	0.7	0.7

Fig. 5. Dataset 1.



p_1
p_2
u3 .7
u1 .65
u2 .6
u2 .9
u1 .8
u3 .7
1
2
3
4
5
6
(a) algorithm $\mathcal{M}_1$ (or TA)
(b) algorithm $\mathcal{M}_2$
(c) algorithm $\mathcal{M}_3$
(d) algorithm $\mathcal{M}_4$

Fig. 6. Example algorithms.

good as $\mathcal{M}$. We formalize this notion of comprehensiveness next, which provides a foundation for finding such a space.

Definition 1 (*Cost Comprehensiveness*). A space Ω is cost comprehensive with respect to cost function $\mathcal{C}$ if for any arbitrary algorithm $\mathcal{M}$, query $\boldsymbol{Q}$, and dataset $\mathcal{D}$, there exists an algorithm $\mathcal{M}' \in \Omega$ answering $\boldsymbol{Q}$ over $\mathcal{D}$ with no more cost, namely,

$$\mathcal{C}(\mathcal{M}') \leq \mathcal{C}(\mathcal{M}).$$

In summary, while $\mathcal{U}$ is our universe in principle, if some space Ω satisfies cost comprehensiveness (Definition 1), it is sufficient to search in Ω instead of $\mathcal{U}$, that is, $argmin_{\mathcal{M} \in \mathcal{U}} \mathcal{C}(\mathcal{M}) = argmin_{\mathcal{M} \in \Omega} \mathcal{C}(\mathcal{M})$. As a key contribution, we identify such comprehensive space Ω for our objective of minimizing total access costs (Eq. (1)): As a key insight, observe Algorithm $\mathcal{M}_2$ (Figure 6(b)) performing the exact same set of accesses as $\mathcal{M}_1$ (Figure 6(a)) only one-at-a-time: By performing the same set of accesses, the two algorithms answer the same query with the exact same cost with respect to our objective cost function (Eq. (1)), which aggregates the costs of all accesses.

Generalizing the observation, just as $\mathcal{M}_1$ has a sequential counterpart $\mathcal{M}_2$, any $\mathcal{M} \in \mathcal{U}$ has a sequential counterpart $\mathcal{M}'$ which performs the same accesses sequentially such that $\mathcal{C}(\mathcal{M}') \leq \mathcal{C}(\mathcal{M})$. Consequently, a space of all sequential algorithms is, indeed, comprehensive with respect to our objective cost function: No algorithm outside of the space can be better than all algorithms in the space.

This comprehensiveness ensures that we will not miss the optimal algorithm by considering only sequential algorithms.[5] More formally, we model such

[5]While parallelization cannot contribute to our optimization objective of Eq. (1) (or total resource usage), it may benefit elapsed time when sources can handle concurrent accesses, such as web

```
while ($\mathcal{P}$ has not gathered sufficient scoring information
        for determining $\mathcal{K}$):
    select $A$ from any possible accesses $sa_i$ and $ra_i$;
    perform $A$; update $\mathcal{K}$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{A\}$;
```

Fig. 7. Algorithm "skeleton" SEQ.

sequential algorithms by the skeleton SEQ in Figure 7, generating all possible sequential algorithms: At any point during such execution, let $\mathcal{P}$ (the "accesses-so-far") be the accesses performed so far (initially empty). Sequential algorithms continue (in the **while**-loop) to select and perform an access, one at each *iteration*, until $\mathcal{P}$ has gathered sufficient information to determine the top-*k* answers $\mathcal{K}$.

We can now abstract a top-*k* algorithm as an access scheduling problem to minimize access costs. Our goal is thus among a space of possible access scheduling, or a space generated by the skeleton SEQ, which we denote as $\mathcal{G}(\text{SEQ})$, to search for the cost-optimal one $\mathcal{M}_{\text{opt}}$, namely,

$$\mathcal{M}_{\text{opt}} = argmin_{\mathcal{M} \in \mathcal{G}(\text{SEQ})} \mathcal{C}(\mathcal{M}). \tag{3}$$

While we have successfully fulfilled our first objective of achieving comprehensiveness (as Section 3 motivated), we are now facing the second challenge of defining a focused space: Unfortunately, although comprehensive, $\mathcal{G}(\text{SEQ})$ is extremely large, since algorithms vary depending on access A selected at each iteration, which can be *any* among a huge set of supported accesses. To illustrate, for $n = 100,000$ objects with $m = 5$ predicates, at each iteration, there can be as many as $m + m \cdot n = 500,005$ different supported accesses to choose from, as our framework does not make, the "no wild guess" assumption (note that with this assumption, random accesses are restricted to seen objects and thus, the upper bound will be smaller). $\mathcal{G}(\text{SEQ})$ is thus simply too large to identify the optimal algorithm efficiently. Therefore, our next goal is to refine the space so as to be as focused as possible, without compromising the comprehensiveness.

To achieve this goal, we first decompose a top-*k* query into "logical tasks" as building blocks. Section 5 will use such building blocks to define a comprehensive and focused search space in Section 6.

5. DEFINING A FOCUSED SPACE

We now ask a fundamental question: While access methods are the physical means for gathering object scores, what are the logical tasks that a top-*k* query must fulfill? Such logical tasks are determined only by the objective of a query, and this is independent of the physical implementation access methods. In other words, any algorithm, regardless of the access methods used, must successfully carry out these tasks. Such a logical view serves as a critical foundation for systematic algorithm design.

sources typically do. Section 9 will show that such parallelization can successfully build upon our access minimization framework.

5.1 Logical View: Scoring Tasks

Since a top-k query is not explicitly constructed with operators (unlike relational queries), its logical tasks are not clear from the query itself. To identify logical tasks, we take an information-theoretic view and ask: What is the required information for answering a top-k query? Given a database $\mathcal{D} = \{u_1, \ldots, u_n\}$, any algorithm $\mathcal{M}$ must gather certain score information for each object u_j so as to determine the top-k results. We can thus "compose" the work of $\mathcal{M}$ by a set of required *scoring tasks* $\{w_1, \ldots, w_n\}$. To define such tasks, let $\mathcal{K} = \{v_1, \ldots, v_k\}$ be the top-k answers (where each v_i represents some u_j from $\mathcal{D}$). In this article, we assume that applications require top-k answers to be completely evaluated. A task w_j is thus to gather the (exact or partial) scores of object u_j by using relevant accesses in order to either (if $u_j \in \mathcal{K}$) compute u_j's complete score or else prove that it cannot score higher than v_k (the kth answer).

Definition 2 (*Scoring Tasks*). Consider a top-k query $Q = (\mathcal{F}, k)$ with top-k answers $\mathcal{K} = \{v_1, \ldots, v_k\}$. The scoring task w_j for object u_j is:

(1) For $u_j \in \mathcal{K}$: w_j must compute the *final* $\mathcal{F}[u_j]$ score.
(2) For $u_j \notin \mathcal{K}$: w_j must indicate (by some partial scores) the maximal-possible $\mathcal{F}[u_j]$ score such that it is tight enough to support that $\mathcal{F}[u_j] < \mathcal{F}[v_k]$.[6]

As a remark, observe that the definition of scoring tasks depends on $\mathcal{K}$ (the top-k answers) and $\mathcal{F}[v_k]$ (the kth score). These values, unfortunately, will remain undetermined before query processing is fully completed; for this "task view" to be useful, our challenge (as we will discuss) is thus to develop mechanisms for identifying unsatisfied tasks *during* query processing, before $\mathcal{K}$ and $\mathcal{F}[v_k]$ are known.

Example 3 (*Scoring Tasks*). Consider our running example Q_1 over $\mathcal{D}_1 = \{u_1, u_2, u_3\}$ (Figure 5), For $k = 1$, the answer is $\mathcal{K} = \{u_3\}$ with $\mathcal{F}[u_3]$=.7 (these values are not known until Q_1 is processed). We can specify the scoring tasks $\{w_1, w_2, w_3\}$ for the three objects as follows.

Consider task w_3: Since $u_3 \in \mathcal{K}$, w_3 must gather all predicate scores, namely, $p_1[u_3]$ and $p_2[u_3]$, for computing $\mathcal{F}[u_3]$. Note that w_3 can do so in various ways, for example, by one sorted access sa_1 into p_1 (which hits u_3 and returns $p_1[u_3] = .7$) and a random access $ra_2(u_3)$ (returning $p_2[u_3] = .7$).

By contrast, task w_2 for u_2 (and similarly w_1 for u_1) only needs to prove, by gathering some partial scores, that $\mathcal{F}[u_2] < \mathcal{F}[u_3] = .7$. To do so, w_2 can use, say, two sorted accesses sa_1 into p_1, which return first $p_1[u_3] = .7$ and then $p_1[u_1]$=.65. Now, since u_2 is still unseen from the sorted list of p_1, it is bounded by the last-seen score, that is, $p_1[u_2] \leq .65$. As $\mathcal{F}[u_2] = \min(p_1[u_2], p_2[u_2])$, $\mathcal{F}[u_2]$ cannot be higher than $p_1[u_2]$, namely, $\mathcal{F}[u_2] \leq .65 < .7$.

[6]To give deterministic semantics, we assume that there are no ties in $\mathcal{F}$ scores otherwise, a *deterministic* tie-breaker function can be used to determine an order, for example, by unique object IDs. Such enforcement of certain tie-breakers enables optimization to consider comparable algorithms that return the exact same set of top-k results.

We stress that these scoring tasks are both necessary and atomic. First, each w_j is necessary: (1) If u_j is a top-k answer, $\mathcal{M}$ cannot return its final score without w_j; (2) otherwise, without w_j proving $\mathcal{F}[u_j] < \mathcal{F}[v_k]$, $\mathcal{M}$ cannot safely exclude u_j from the top-k. Furthermore, each w_j, as a per-object task, is *atomic*: For arbitrary $\mathcal{F}$, w_j cannot generally be decomposed into smaller required subtasks. For case (1) of Definition 2, that is, when $u_j \in \mathcal{K}$, obviously, all predicate scores are required. For case (2), *no* subsets of u_j's predicate scores are absolutely required, as long as the upper-bound inequality can be proved.

In summary, we now view query processing as equivalent to fulfilling a set of (necessary and atomic) tasks $\{w_1, \ldots, w_n\}$: Each task w_j, for object u_j, gathers the required per-object information. Only when (and clearly when) all the tasks are fulfilled can the query be answered.

5.2 Identifying Unsatisfied Tasks

At any point during processing, some scoring task (defined in Definition 2) is *unsatisfied*. Formally, at any point, scoring task w_j is unsatisfied, with respect to the accesses performed so far, if the scoring task in Definition 2 is yet to be fulfilled. For example, when object u_i is one of the top-k results and only partially evaluated at the point, its scoring task w_i is considered to be unsatisfied and u_i still needs to be further evaluated. To focus query processing, it is critical to identify the unsatisfied tasks and complete such tasks first. However, *during* query processing, it is challenging to judge whether a task is satisfied, since $\mathcal{K} = \{v_1, \ldots, v_k\}$, which our task specification (Definition 2) requires, is not determined until the very end.

In fact, for our purpose, we can address a slightly different problem: Given a set of accesses-so-far $\mathcal{P}$ that has been performed, can we find *any* unsatisfied task? Instead of identifying *all*, for query processing to move on, it is sufficient to find just one (note that any unsatisfied task must eventually be fulfilled). Our insight is that by comparing the "score state" of objects, we can always reason some tasks to be clearly unsatisfied, regardless of the eventual result $\mathcal{K}$.

Example 4 (*Unsatisfied Tasks*). Consider Q_1 over $\mathcal{D}_1$, suppose that at some point, we have performed $\mathcal{P} = \{sa_1, sa_1, sa_2, ra_1(u_2)\}$. Referring to Figure 5, these accesses will gather the following score information:

—The two sorted accesses sa_1 on p_1 will hit $p_1[u_3] = .7$ and $p_1[u_1] = .65$. Due to a side-effect (Section 4), the "unseen" objects (i.e., u_2) will be bounded by the last-seen score, namely, $p_1[u_2] \leq .65$.

—The sorted access sa_2 on p_2 will return $p_2[u_2] = .9$, and set upper bounds $p_2[u_1] \leq .9$ and $p_2[u_3] \leq .9$.

—The random access $ra_1(u_2)$ returns $p_1[u_2] = .6$.

Putting all of this together, Figure 8 summarizes the current "score state." For u_1, the aforementioned accesses gathered $p_1[u_1] = .65$ and $p_2[u_1] \leq .9$, and thus $\mathcal{F}[u_1] \leq \min(.65, .9) = .65$. Similarly, $\mathcal{F}[u_2] = .6$ and $\mathcal{F}[u_3] \leq .7$

OID	p_1	p_2	$\mathcal{F}$
u_1	.65	$\leq .9$	$\leq .65$
u_2	.6	.9	.6
u_3	.7	$\leq .9$	$\leq .7$

Fig. 8. The score state of Example 4.

At this point, while we do not know what $\mathcal{K}$ will be (as Definition 2 requires), we can identify at least the scoring task w_3 for u_3 as unsatisfied, *no matter* what $\mathcal{K}$ is:

—if $u_3 \in \mathcal{K}$ (i.e., u_3 will eventually be the top-1): Here, w_3 needs to gather exact $p_2[u_3]$ to compute the $\mathcal{F}$ score.

—if $u_3 \notin \mathcal{K}$: In this case, the top-1 is u_1 or u_2, with $\mathcal{F}$ scores of (at most) .65 and .6, respectively (Figure 8). Thus, the top-1 score (i.e., $\mathcal{F}[v_k]$ in Definition 2) is at most .65. Clearly, w_3 has *not* proved that $\mathcal{F}[u_3] \leq .65$, since u_3 can score as high as .7.

As Example 4 implies, task w_j is unsatisfied if u_j has "potential" to be in the top-*k* results $\mathcal{K}$. For such u_j (e.g., u_3), regardless of what $\mathcal{K}$ will be, we must know more about its scores in order to declare it as either top-*k* or not. We thus identify whether w_j is unsatisfied as follows: We quantify the current "potential" of u_j (with respect to $\mathcal{P}$), and determine if this potential is high enough to make the top-*k* results.

To begin with, we measure the *current potential* of an object by its maximal-possible score. Define $\overline{\mathcal{F}}_{\mathcal{P}}[u_j]$ as the maximal score that u_j may possibly achieve, given the partial scores that accesses-so-far $\mathcal{P}$ has gathered. As a standard assumption, $\mathcal{F}$ is monotonic, namely, $\mathcal{F}(x_1, \ldots, x_m) \geq \mathcal{F}(y_1, \ldots, y_m)$ when $\forall i : x_i \geq y_i$. We thus can compute $\overline{\mathcal{F}}_{\mathcal{P}}[u_j]$ by substituting unevaluated predicates with their maximal-possible scores. Note that p_i is bounded by the last-seen score from its sorted accesses, denoted as $\overline{p_i}$ (Section 4 discussed such side-effects of sorted accesses). For instance, as Figure 8 shows, $\overline{\mathcal{F}}_{\mathcal{P}}(p_1, p_2)[u_1] = \min(p_1[u_1] = .65, \overline{p_2} = .9) = .65$. Thus, formally, $\overline{\mathcal{F}}_{\mathcal{P}}(p_1, \ldots, p_m)[u_j] =$

$$\mathcal{F}\left(\begin{array}{ll} p_i = p_i[u_j] & \text{if } \mathcal{P} \text{ has determined } p_i[u_j] \\ p_i = \overline{p_i} & \text{otherwise.} \end{array} \forall i\right) \tag{4}$$

Further, we focus on the current top-*k* objects by their potentials. Let $\mathcal{K}_{\mathcal{P}} = \{v_1, \ldots, v_k\}$ be these current top objects ranked by their $\overline{\mathcal{F}}_{\mathcal{P}}$ scores (to illustrate, in Example 4, $\mathcal{K}_{\mathcal{P}} = \{u_3\}$). There are two situations, depending on whether the current top objects are incomplete.

First, if $\mathcal{K}_{\mathcal{P}}$ contains any *incomplete* object v_j with only partial scores, then as Example 4 argued for u_3 (an incomplete top-1), such v_j needs further accesses either way. This is by Definition 2: (1) If v_j is indeed the final top-*k*, it needs complete evaluation. (2) else, it needs further accesses to lower its maximal-possible score so as to be safely excluded from top-*k*. Thus, task w_j for such incomplete v_j is clearly unsatisfied.

Second, if all objects $v_1, \ldots, v_k$ in $\mathcal{K}_{\mathcal{P}}$ are complete, then these current top-*k* with respect to $\mathcal{P}$ are now indeed the *final* top-*k* (i.e., $\mathcal{K}_{\mathcal{P}} = \mathcal{K}$) (and the query can halt with these answers). To see why, we make two observations: (1) Every $v_j \in \mathcal{K}_\mathcal{P}$ is complete and thus has its exact score, that is, $\mathcal{F}[v_j] = \overline{\mathcal{F}}_\mathcal{P}[v_j]$. (2) Every object $u_i \notin \mathcal{K}_\mathcal{P}$ with the current ranking has a maximal-possible score lower than the aforementioned exact scores, namely, $\forall v_j \in \mathcal{K} : \overline{\mathcal{F}}_\mathcal{P}[u_i] \leq \mathcal{F}[v_j]$. It follows that these v_j are the fully evaluated top-k answers. With these two observations, Definition 2 will declare that all scoring tasks are satisfied and thus, query processing can indeed halt.

Theorem 1 states our results on identifying unsatisfied tasks.

Theorem 1 (Unsatisfied Scoring Tasks). *Consider a top-k query $Q = (\mathcal{F}, k)$ over $\mathcal{D} = \{u_1, \ldots, u_n\}$. With respect to a set $\mathcal{P}$ of performed accesses, let $\mathcal{K}_\mathcal{P} = \{v_1, \ldots, v_k\}$ be the current top-k objects ranked by $\overline{\mathcal{F}}_\mathcal{P}[\cdot]$.*

(1) *For all $v_j \in \mathcal{K}_\mathcal{P}$ such that v_j has not been completely evaluated, its scoring task w_j is unsatisfied.*

(2) *If all v_j's are complete, then every scoring task w_j, $\forall u_j \in \mathcal{D}$, is satisfied and $\mathcal{K}_\mathcal{P}$ is the top-k results.*

Proof. (1) If $v_j \in \mathcal{K}_\mathcal{P}$ has not been completely evaluated, its scoring task w_j is unsatisfied; no matter what $\mathcal{K}$ will eventually be, there are the following two possible situations.

—if $v_j \in \mathcal{K}$: As its scoring task w_j must compute $\mathcal{F}[v_j]$, the task is not complete until we gather $p_i[v_j]$ for every unevaluated predicate p_i of v_j. Since v_j has not been completely evaluated, such p_i must exist and thus w_j is still unsatisfied (by Definition 2, case 1).

—if $v_j \notin \mathcal{K}$: Supposing that its scoring task w_j is satisfied that will indicate that there are *at least k* objects u (e.g., those in $\mathcal{K}$) satisfying $\overline{\mathcal{F}}_\mathcal{P}[v_j] < \mathcal{F}[u]$, which in turn satisfy $\overline{\mathcal{F}}_\mathcal{P}[v_j] < \overline{\mathcal{F}}_\mathcal{P}[u]$, as $\mathcal{F}[u] \leq \overline{\mathcal{F}}_\mathcal{P}[u]$. Meanwhile, as $v_j \in \mathcal{K}_\mathcal{P}$, there are *at most k* $- 1$ objects u $\overline{\mathcal{F}}_\mathcal{P}[v_j] < \overline{\mathcal{F}}_\mathcal{P}[u]$, a contradiction.

(2) If all v_j's are complete, $\forall\, u \notin \mathcal{K}_\mathcal{P} : \mathcal{F}[v_j] = \overline{\mathcal{F}}_\mathcal{P}[v_j] > \overline{\mathcal{F}}_\mathcal{P}[u] \geq \mathcal{F}[u]$, $\forall\, u \notin \mathcal{K}_\mathcal{P}$ holds and therefore $\mathcal{K}_\mathcal{P} = \mathcal{K}$ holds. With this, we can show that scoring task w_i is satisfied for every $u_i \in \mathcal{D}$.

—$\forall u_i \in \mathcal{K}_\mathcal{P} = \mathcal{K}$: As every $u_i \in \mathcal{K}$ has been completely evaluated, w_i is satisfied (by Definition 2, case 1).

—$\forall u_i \notin \mathcal{K}_\mathcal{P} = \mathcal{K}$: As $\forall v_j \in \mathcal{K} : \mathcal{F}[v_j] > \mathcal{F}[u_i]$ holds, w_i is satisfied (by Definition 2, case 2). □

In summary, Theorem 1 states that at any point, top-k objects need to be further evaluated, that is, at least one either sorted or random access on its unevaluated predicates is necessary. However, note that this does not necessarily mean such objects need to be evaluated completely.

Theorem 1 thus provides an important basis for constructing a focused space by guaranteeing to identify unsatisfied tasks, if any: Condition 2 gives a precise way to determine if there still exist any unsatisfied tasks, while Condition 1 will identify at least some of them (i.e., those incomplete v_j). Meanwhile, note that this makes no assumptions on particular physical accesses—we can thus generally use arbitrary top-k accesses, not only random accesses, as in Chang and Hwang [2002], but also sorted accesses with progressiveness and side-effects, as well (Section 4).

6. PUTTING TOGETHER FRAMEWORK NC

This section develops a space that is both comprehensive and focused. Built upon our algorithm abstraction (Section 4) and task decomposition (Section 5), Section 6.1 first develops a framework, NC, which induces such an algorithm space. Section 6.2 then shows that the space induced is both comprehensive and focused.

6.1 The Framework

Recall that in relational queries, this space is induced by an algebraic framework: As a query is composed of relational operators, an algorithm space simply enumerates all algebraically equivalent query plans. In other words, the algebraic framework of relational queries induces a space of query plans, each as a different schedule. Optimization consists of finding a good schedule of operations conforming to the framework.

Building on this insight, we develop a framework that, by scheduling and performing an access one-by-one at each iteration, generates a space of algorithms. For instance, a framework where *any* supported access is scheduled at iteration, namely, SEQ (Section 4), is essentially a space of all sequential algorithms . In contrast, in this section, in order to render a more focused space, we develop a framework that hinges on the insight that query processing can focus *only* on unsatisfied tasks, without compromising optimality. In other words, our framework will first identify some unsatisfied task w_j and then focus selection on *only* those accesses for fulfilling w_j.

This insight is built upon task decomposition (Section 5); that top-k query processing is equivalent to fulfilling a set of (necessary and atomic) tasks $\{w_1, \ldots, w_n\}$. With this task view, during processing, when a set of accesses $\mathcal{P}$ has been performed, we can identify unsatisfied tasks by Theorem 1 (when all tasks are satisfied, query processing can halt, as Theorem 1 also asserts). For any unsatisfied w_j, we can construct a set of accesses N_j, *specifically* for satisfying w_j, by collecting all and only those accesses that can further process w_j. These accesses constitute the necessary choices for fulfilling w_j. More precisely, N_j will consist of any (random or sorted) accesses that can return (exact or bounding) scores about u_j's unevaluated predicates (as Theorem 1 states, for such unsatisfied w_j, its object u_j must still be incomplete).

Example 5 (*Necessary Choices*). Continue our running example. Example 4 identified that task w_3 is unsatisfied for object u_3, with a score state ($p_1 = .7$, $p_2 \leq .9 \rightarrow \mathcal{F} \leq .7$), as Figure 8 shows. Note that w_3 is unsatisfied, since the accesses-so-far $\mathcal{P}$ has not gathered sufficient information for u_3 (for either case of Definition 2). To satisfy w_3, we must know more about u_3, especially about the predicate p_2 with unknown score, using some of the following accesses:

—sorted accesses on p_2: Performing sa_2 can lower the upper bound of $p_2[u_3]$. In other words, as $\mathcal{P}$ (Example 4) already has one sa_2, the next sa_2 will return

```
**Framework** NC($Q$, $\mathcal{D}$): Necessary Choices
**Input:** query $Q = (\mathcal{F}(p_1, \ldots, p_m), k)$,
           database $\mathcal{D}= \{u_1, \ldots, u_n\}$
**Output:** $\mathcal{K}$, top-$k$ objects from $\mathcal{D}$ w.r.t. to $\mathcal{F}$
(1) $\mathcal{P} \leftarrow \phi$; // accesses-so-far
(2) $\mathcal{K}_\mathcal{P} \leftarrow \{v_1, \ldots, v_k \mid$ top-$k$ from $\mathcal{D}$ ranked by $\overline{\mathcal{F}}_\mathcal{P}[\cdot]\}$;
(3) while (true)
(4)     $U \leftarrow \{v_j | v_j \in \mathcal{K}_\mathcal{P}; v_j$ is incomplete$\}$)
(5)     if ($U$=={}) break;
(6)     $v_j \leftarrow$ any object in $U$; // e.g., the highest-ranked
(7)     $N_j \leftarrow \{sa_i, ra_i(v_j) | \; p_i[v_j]$ is undetermined by $\mathcal{P}\}$;
(8)     alternatives $\leftarrow N_j$;
(9)     Select access $A$ from alternatives;// access selection.
(10)    perform $A$; update $\mathcal{K}_\mathcal{P}$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{A\}$;
(11)    return $\mathcal{K}= \mathcal{K}_\mathcal{P}$;
```

Fig. 9. Framework NC.

u_1 with score .8 (Figure 5). This new last-seen score by sa_2 will give u_3 a "tighter" bound for p_2 (from $\leq .9$ to $\leq .8$).

—random access on p_2: Performing $ra_2(u_3)$ will return the exact score of u_3 for p_2, thus rendering u_3 completely evaluated, with score state ($p_1 = .7$, $p_2 = .7 \rightarrow \mathcal{F} = .7$). In fact, w_3 is now satisfied.

Putting this all together, N_3 is thus $\{sa_2, ra_2(u_3)\}$.

THEOREM 2 (NECESSARY CHOICES). *Given a set of performed accesses $\mathcal{P}$, let w_j be an unsatisfied scoring task for object u_j. The* necessary choices *for w_j with respect to $\mathcal{P}$ is $N_j = \{sa_i, ra_i(u_j) \mid p_i[u_j]$, which is undetermined by $\mathcal{P}\}$. Without performing at least one of N_j, w_j remains unsatisfied.*

PROOF. If $v_j \in \mathcal{K}$: As its scoring task w_j must compute $\mathcal{F}[v_j]$, w_j remains unsatisfied until we gather $p_i[v_j]$ for every unevaluated predicate p_i of v_j, either by $ra_i(v_j)$ or by sa_i accessing v_j, for all unevaluated predicates p_i.

If $v_j \notin \mathcal{K}$: As its scoring task w_j requires to lower the upper bound of v_j below the top-k results, w_j remains unsatisfied until we lower the upper bound of v_j, either by evaluating unevaluated predicates by $ra_i(v_j)$ or by lowering the upper bounds by sa_i for all unevaluated predicates p_i. □

Figure 9 illustrates our framework NC. At each iteration, it identifies necessary choices, with Theorem 1 to guide this process. At any point, NC maintains $\mathcal{K}_\mathcal{P}$, the current top-k objects with respect to accesses-so-far $\mathcal{P}$ ranked by maximal-possible scores $\overline{\mathcal{F}}_\mathcal{P}[\cdot]$. Some objects in $\mathcal{K}_\mathcal{P}$ may still be incomplete, as represented as U in the figure. As Theorem 1 specifies, there are two situations:

(1) if $U = \phi$: As all top-k objects are complete, Theorem 1 asserts no more unsatisfied tasks, which is thus the termination condition of NC: NC will break the **while**-loop (since $U = \phi$), and return $\mathcal{K}_\mathcal{P}$.
(2) otherwise: Since $U \neq \phi$, there are incomplete top-k objects. Any such object v_j corresponds to an unsatisfied task w_j by Theorem 1. NC arbitrarily picks

step	$\overline{p_1}$	$\overline{p_2}$	$\mathcal{K}_\mathcal{P}$	alternatives	*Select*
1.	1	1	$\{u_3\}$	$N_3{=}\{sa_1, sa_2, ra_1(u_3), ra_2(u_3)\}$	sa_1
2.	0.7	1	$\{u_3\}$	$N_3{=}\{sa_2, ra_2(u_3)\}$	$ra_2(u_3)$

Fig. 10. Illustration of NC.

any such v_j (say, the highest-ranked) without compromising optimality, and constructs the necessary choices N_j (by Theorem 2) as alternatives. As each unsatisfied task remains unsatisfied until at least one among its necessary choices is performed, arbitrarily picking one such unsatisfied task does not compromise optimality.

Note that NC essentially relies on Theorem 1 to isolate a set of necessary choices. Theorem 1 enables an effective way to search for necessary choices by maintaining the current top-*k* objects $\mathcal{K}_\mathcal{P}$. Thus, a search mechanism for finding unsatisfied tasks should return top-*k* objects when requested, for example, using a *priority queue* that orders objects by maximal-possible scores as priorities. Note that in the beginning, all objects have the same maximal-possible score, namely, a perfect 1.0. This initial condition is simply a special case of ties: In principle, NC will initialize (in Step 2) $\mathcal{K}_\mathcal{P}$ with some deterministic tie-breaking order. While any tie-breaker can be used, for the sake of presentation, our examples will use OID to break ties, for example, when u_i and u_j tie and $i > j$, then we consider that u_i outranks u_j.

Observe that when $k > 1$, there may be multiple incomplete v_j in $\mathcal{K}_\mathcal{P}$ at each iteration. We stress that NC can simply choose *any* such v_j to proceed, such as that with the highest partial score, and still ensure comprehensiveness. The reason is that each such v_j designates an unsatisfied task w_j, which remains unsatisfied until some access is performed for the task. Every w_j is thus "equally" necessary, as we formally discuss in Theorem 3. Example 6 illustrates how NC works.

Example 6 (*Framework NC*). Figure 10 shows the execution of the algorithm $\mathcal{M}_4$ (Figure 6(d)) that NC can generate. Initially, at Step 1 (Figure 10), since all the maximal-possible scores tie at 1.0, $\mathcal{K}_\mathcal{P}$ is set to $\{u_3\}$ (by the highest OID, according to our tie-breaker). According to NC, $\mathcal{M}_4$ can then *Select* an access from alternatives $= N_3$, such as sa_1 in this case, which returns $p_1[u_3] = .7$ (see Figure 5) and lowers $\overline{p_1}$ to .7.

At Step 2, as all maximal-possible scores tie at .7, u_3 remains as the top in $\mathcal{K}_\mathcal{P}$. However, u_3 now induces a smaller N_3, with accesses only for its unevaluated predicate p_2. $\mathcal{M}_4$ can then *Select* $ra_2(u_3)$, which returns $p_2[u_3] = .7$ and completes u_3 with $\mathcal{F}[u_3]{=}.7$. Since $\mathcal{K}_\mathcal{P}$ with u_3 as the top-1 is now fully complete, according to NC, $\mathcal{M}_4$ will halt, with total accesses $\mathcal{P}(\mathcal{M}_4) = \{sa_1, ra_2(u_3)\}$.

6.2 Comprehensive and Focused Space

This section shows that the space framework which NC renders is not only far more focused than that rendered by the space SEQ, but also sufficiently comprehensive. First, we note that NC, by focusing on only necessary choices, that is, $|$ alternatives$| = 2 \cdot m$, is clearly more focused than SEQ in selecting an access from any supported accesses, namely, $|\,\mathsf{alternatives}| = m + m \cdot n$. Further, we stress that although more focused, NC is still comprehensive enough for optimization. This comprehensiveness results from the "completeness" property of necessary choices, which NC uses as alternatives, as we formally state to follow.

THEOREM 3 (N_j COMPLETENESS). *A set of necessary choices N_j for every j, identified by* NC *for an unsatisfied task w_j, is* complete *with respect to accesses-so-far $\mathcal{P}$ such that any algorithm having completed $\mathcal{P}$ must continue with at least one of N_j.*

PROOF. N_j, by Theorem 2, contains all accesses that can contribute to the unsatisfied task w_j. Since w_j is necessary (Section 5.1), *at least one* access in N_j must be further executed, or w_j cannot be satisfied nor the query answered. Thus, N_j is complete with respect to accesses-so-far $\mathcal{P}$. □

This completeness property ensures that the space of algorithms generated by framework NC, denoted as $\mathcal{G}$(NC), is comprehensive for optimization (i.e., cost comprehensiveness in Definition 1), as Theorem 4 next states. With this guarantee, it is sufficient to search only within NC for an optimal algorithm.

THEOREM 4 (NC COMPREHENSIVENESS). *For any algorithm $\mathcal{M}_1$ with an access cost C_1 with respect to the cost model $\mathcal{C}$ (Eq. (1)), there exists an algorithm $\mathcal{M}_2$ in $\mathcal{G}$(NC) with cost C_2 such that $C_2 \leq C_1$.*

PROOF. Consider any query processing by $\mathcal{M}_1$ (for some query Q over database $\mathcal{D}$). We will show the generality of NC by constructing an algorithm $\mathcal{M}_2$ in framework NC for the same processing, such that $\mathcal{M}_2$ costs no more than $\mathcal{M}_1$.

Let $\mathcal{P}_1$ be the total accesses that $\mathcal{M}_1$ has performed, namely, $\mathcal{P}(\mathcal{M}_1) = \mathcal{P}_1$. Since $\mathcal{M}_2$ follows the iterative framework (Figure 10), let $\mathcal{P}_2^j$ be the accesses of $\mathcal{M}_2$ before the jth iteration; initially, $\mathcal{P}_2^1 = \phi$. Similarly, let $\mathsf{alternatives}^j$ be alternatives of $\mathcal{M}_2$ at the jth iteration.

Our proof is based on the following two lemmas (L_1 and L_2) for every iteration j, which we prove later.

—L_1: $\mathsf{alternatives}^j \cap (\mathcal{P}_1 - \mathcal{P}_2^j) \neq \phi, \forall j$.

—L_2: $\mathcal{P}_2^j \subseteq \mathcal{P}_1, \forall j$.

Note that by proving L_1, we show NC can construct algorithm $\mathcal{M}_2$ so as to follow the access of $\mathcal{M}_1$ at each iteration. More specifically, for every iteration j, $\mathcal{M}_2$ selects one access from $\mathsf{alternatives}^j$ that is performed by $\mathcal{M}_1$, but not yet by $\mathcal{M}_2$, namely, $\mathcal{P}_1 - \mathcal{P}_2^j$, which is possible if $\mathcal{P}_2^j \subseteq \mathcal{P}_1, \forall j$. We can then show, by proving L_2, that such algorithm $\mathcal{M}_2$ halting at iteration j (denoted as $\mathcal{M}_2^j$) incurs no more access than $\mathcal{M}_1$ at any iteration, that is, $\forall j : \mathcal{P}_2^j \subseteq \mathcal{P}_1$. Note that this immediately implies that $\mathcal{C}(\mathcal{M}_2^j) \leq \mathcal{C}(\mathcal{M}_1)$ as well, because our cost function (Eq. (1)) is monotonic to the accesses performed. In other words, in our cost model, if $\mathcal{M}_1$ performs more accesses than $\mathcal{M}_2^j$ in every access type, then $\mathcal{M}_1$ is guaranteed to have an overall higher cost, namely, $\mathcal{P}(\mathcal{M}_2^j) \subseteq \mathcal{P}(\mathcal{M}_1) \Longrightarrow \mathcal{C}(\mathcal{M}_2^j) \leq \mathcal{C}(\mathcal{M}_1)$. To complete the proof, we now show by induction that L_1 and

L_2 hold; we will also specify the behavior of $\mathcal{M}_2$ for each iteration so as to show how it can be constructed in the NC framework.

—$j = 1$: Consider L_1. We note that by definition of the framework NC, $\mathsf{alternatives}^j$ is complete in that any algorithm (like $\mathcal{M}_1$) which has performed $\mathcal{P}_2^j$ must have performed some access A among $\mathsf{alternatives}^j$. Thus, as $\mathcal{M}_1$ has performed $\mathcal{P}_2^1$ (trivially, since $\mathcal{P}_2^1 = \phi$), by the completeness, it must have performed access $A \in \mathsf{alternatives}^1$ in addition to $\mathcal{P}_2^1$. In other words, A is in both $\mathsf{alternatives}^1$ and $\mathcal{P}_1 - \mathcal{P}_2^1$, and thus L_2 holds. Moreover L_2 is trivial, since initially $\mathcal{P}_2^1 = \phi$.

—$j = k$: As the induction hypothesis, assume for $j = k$ that the lemmas hold.
What should algorithm $\mathcal{M}_2$ do in each iteration? We now construct $\mathcal{M}_2$ for iteration k: If $\mathcal{M}_2$ exhausts $\mathcal{P}_1$, which provides enough information to answer Q, $\mathcal{M}_2$ halts right before this iteration. Otherwise, NC requires $\mathcal{M}_2$ to select one access from $\mathsf{alternatives}^k$ to continue. We will let $\mathcal{M}_2$ choose an access A^k which is also in $\mathcal{P}_1 - \mathcal{P}_2^k$. Such A^k must exist by L_2, that is, $A^k \in \mathsf{alternatives}^k \cap (\mathcal{P}_1 - \mathcal{P}_2^k)$.

—$j = k + 1$: First, L_1 holds; by L_1 (just prove on), $\mathcal{P}_2^{k+1} \subseteq \mathcal{P}_1$, thus $\mathcal{M}_1$ has performed $\mathcal{P}_2^{k+1}$. By the completeness of $\mathsf{alternatives}^{k+1}$, $\mathcal{M}_1$ must have performed, *in addition* to $\mathcal{P}_2^{k+1}$, some access $A \in \mathsf{alternatives}^{k+1}$. In other words, A is in both $\mathsf{alternatives}^{k+1}$ and $\mathcal{P}_1 - \mathcal{P}_2^{k+1}$, and thus L_2 holds.
Second, L_1 holds: Note that $\mathcal{P}_2^{k+1} = \mathcal{P}_2^k \cup \{A^k\}$. Since $\mathcal{P}_2^k \subseteq \mathcal{P}_1$ (by the induction hypothesis on L_1) and $A^k \in \mathcal{P}_1 - \mathcal{P}_2^k$ (by the construction of $\mathcal{M}_2$), it follows that $\mathcal{P}_2^{k+1} \subseteq \mathcal{P}_1$ holds. □

In summary, we stress that NC, as an algorithm-generating framework, defines an optimization space which is comprehensive and focused. Our goal, in principle, is thus to "instantiate" an optimal algorithm $\mathcal{M}_{\mathsf{opt}}$ in $\mathcal{G}(\mathsf{NC})$, which depends on query- and data-specific factors. Section 7 will discuss optimization techniques for finding $\mathcal{M}_{\mathsf{opt}}$, further refining Eq. (3), as follows:

$$\mathcal{M}_{\mathsf{opt}} = argmin_{\mathcal{M} \in \mathcal{G}(\mathsf{NC})} \mathcal{C}(\mathcal{M}). \tag{5}$$

7. SEARCH: DYNAMIC OPTIMIZATION

In this section, we discuss how to actually optimize top-*k* queries by using framework NC, detailed in Section 6. As briefly discussed, with optimization space $\mathcal{G}(\mathsf{NC})$ defined, the query optimization problem is essentially that of identifying the cost-optimal algorithm $\mathcal{M}_{opt}$ in Eq. (5). For systematic optimization, we must address the following three tasks, each corresponding to its counterpart in Boolean query optimization:

(1) *Space reduction*: Although $\mathcal{G}(\mathsf{NC})$ is already much focused, it is still too large for exhaustive search. We thus design a suite of systematic techniques to reduce the space, for which we can argue how they retain promising algorithms in the space.

(2) *Cost estimation*: As a ground to compare algorithms in the space, the optimizer must be able to estimate their cost. Our cost estimation extends the insight of its Boolean counterpart, as we will discuss in Section 7.2.
(3) *Search*: Within the space of algorithms with their estimated costs, we design effective optimization schemes to prioritize search. Similarly, Boolean optimization enumerates plans in particular ways, for example, dynamic programming.

7.1 Space Reduction

While already much focused, $\mathcal{G}(\mathsf{NC})$ is still too large for exhaustive search. At each iteration, NC may *Select* any type of access on any unevaluated predicates of top-k objects. We thus need to further focus within NC with some systematic reduction techniques. These techniques contribute in two ways: First, they reduce the space significantly, while we can argue how they retain the promising algorithms for consideration. Second, they give "orders" to the reduced space so that algorithm can be systematically enumerated by a few configuration parameters. In particular, we use the following techniques for optimization:

First, we choose to focus on *SR* (sorted-then-random) algorithms, which perform all sa_i on predicate p_i, if such exists, before any $ra_i(\cdot)$. We argue that focusing on such algorithms allows us to reduce our plan space with no loss of optimality– Lemma 1 states that, for any top-k algorithm, we have its *SR*-counterpart gathering the same score information, with no more cost.

LEMMA 1 (*SR*-COUNTERPART). *For any algorithm $\mathcal{M}_1 \in \mathcal{G}(\mathsf{NC})$, there exists its SR-counterpart $\mathcal{M}_2$ with no more cost, namely, $\mathcal{C}(\mathcal{M}_2) \leq \mathcal{C}(\mathcal{M}_1)$.*

PROOF. We prove by constructing the *SR*-counterpart $\mathcal{M}_2$ of $\mathcal{M}_1$ with no more cost. Let $\mathcal{P}_1^i$ be the total accesses that $\mathcal{M}_1$ has performed on p_i, that is, $\mathcal{P}(\mathcal{M}_1) = \sum_i \mathcal{P}_1^i$. In other words, $\mathcal{P}_1^i$ should be sufficient for collecting the same information as $\mathcal{M}_1$ on predicate p_i. We thus construct $\mathcal{M}_2$ to perform the same accesses in $\mathcal{P}_1^i$ for every p_i, but in a sorted-then-random manner, namely, $\mathcal{P}_2^i$ first chooses every sorted access in $\mathcal{P}_1^i$ and then every random access in $\mathcal{P}_1^i$. However, note that some $ra_i(o) \in \mathcal{P}_1^i$ will be redundant in $\mathcal{P}_2^i$ if $p_i[o]$ has been already evaluated by preceding sorted access. Consequently, $\forall i : \mathcal{P}_2^i \subseteq \mathcal{P}_1^i$, and thus $\mathcal{M}_2$ terminates as early as $\mathcal{M}_1$, if not earlier, that is, $\mathcal{C}(\mathcal{M}_2) \leq \mathcal{C}(\mathcal{M}_1)$. □

Second, we assume that random access on every object follows the same "global" order $\mathcal{H}$. Specially, when multiple random accesses exist in alternatives, we follow some particular order $\mathcal{H}$ (given by the optimizer; see Section 7.3) so as to choose which to perform. To illustrate, supposing necessary choices are alternatives $= \{ra_i(u_1), ra_j(u_1)\}$ given $\mathcal{H} = (p_i, p_j)$, we pick $ra_i(u_1)$ first, since the next unevaluated predicate of u_1 is p_i, according to $\mathcal{H}$, which we denote as $next(u_1, \mathcal{H}) = p_i$. According to our preliminary study [Chang and Hwang 2002], global scheduling is as effective as local scheduling (thus hardly compromising comprehensiveness), while significantly reducing the per-object scheduling overhead.

By focusing on the preceding two techniques, we propose framework NC with SR/G (SR-subset and global) scheduling techniques, trading high efficiency over

step	$\overline{p_1}$	$\overline{p_2}$	$\mathcal{K}$	alternatives	*Select*
1.	1	1	$\{u_3\}$	$N_3 = \{sa_1, sa_2, ra_1(u_3), ra_2(u_3)\}$	sa_1
2.	0.7	1	$\{u_3\}$	$N_3 = \{sa_2, ra_2(u_3)\}$	sa_2
3.	0.7	0.9	$\{u_3\}$	$N_3 = \{sa_2, ra_2(u_3)\}$	sa_2
4.	0.7	0.8	$\{u_3\}$	$N_3 = \{sa_2, ra_2(u_3)\}$	$ra_2(u_3)$

Fig. 11. Illustration of SR/G techniques.

```
Procedure Select (alternatives, $\Delta, \mathcal{H}$):
if $\exists sa_i \in$ alternatives such that $\overline{p_i} > \delta_i$:
    $A \leftarrow sa_i$;
else if $\exists ra_i(u) \in$ alternatives such that $p_i = next(u, \mathcal{H})$:
    $A \leftarrow ra_i(u)$;
```

Fig. 12. *Select* with SR/G techniques.

a slight compromise of comprehensiveness. These techniques customize the *Select* routine of NC, as Figure 12 shows: Now the selection is more focused, guided by the two parameters Δ and $\mathcal{H}$. Here, $\Delta = \{\delta_1, \ldots, \delta_m\}$ represents the suggested depth of sorted access δ_i for each predicate p_i, while $\mathcal{H}$ determines the ordering of predicate evaluation, which will be determined by the optimizer, as Section 7.3 will discuss. In essence, *Select* chooses sorted access whenever there exists sa_i which has not reached the suggested depth δ_i for predicate p_i, that is, $\overline{p_i} > \delta_i$.[7] Otherwise, it performs random access in alternatives by picking the next unevaluated predicate (according to $\mathcal{H}$). Example 7 illustrates how these techniques actually work with our running example (for the sake of presentation, NC from here on refers to the framework with SR/G techniques).

Example 7 (*SR/G Techniques*). Consider our running example Q_1 on Dataset 1. Figure 11 illustrates how SR/G techniques guide the access selection of NC when $\Delta = (0.8, 0.8)$ and $\mathcal{H} = (p_1, p_2)$.

At Step 1, among necessary choices alternatives $= N_3$, *Select* focuses on sa_1 and sa_2, since the suggested sorted access depths have not yet been reached, that is, $\overline{p_1} > \delta_1 = 0.8$ and $\overline{p_2} > \delta_2 = 0.8$ (we arbitrarily pick one, e.g., sa_1.) Similarly, at Steps 2 and 3, *Select* chooses sa_2, until it lowers $\overline{p_2}$ below the suggested depth δ_2 after Step 3. Then, at Step 4, we perform $ra_2(u_3)$, which completes the evaluation on u_3. Hence, NC can return u_3 as the top-1 answer with four accesses $\mathcal{P} = \{sa_1, sa_2, sa_2, ra_2(u_3)\}$, as $\mathcal{F}[u_3]$ is higher than the maximal-possible scores of the rest.

In addition to reducing the search space, SR/G techniques enable enumerating algorithms by parameters Δ and $\mathcal{H}$. In other words, every SR algorithm can be identified by a $(\Delta,\mathcal{H})$ pair. Consequently, our optimization problem can now be restated as identifying the minimal-cost algorithm $(\Delta_{opt}, \mathcal{H}_{opt})$ such that $(\Delta_{opt}, \mathcal{H}_{opt}) = argmin_{\Delta,\mathcal{H}} \mathcal{C}((\Delta, \mathcal{H}))$.

[7]While rare in practice, there can be an extreme case where the suggested depth is too shallow, so that all objects seen from the sorted access are fully evaluated before identifying the top-*k* results. In such a case, NC can incrementally increase the depth proportionally to the suggested depth for each predicate.

7.2 Cost Estimation

As a prerequisite to identifying the cost-optimal algorithm $(\Delta_{opt}, \mathcal{H}_{opt})$, we need to develop a means to estimate the cost of an SR/G top-k algorithm $(\Delta, \mathcal{H})$. To motivate, recall the cost estimation for Boolean query plans. The cost of Boolean query essentially sums up the cost of processing each predicate p_i for the cardinality N_i of the objects that evaluate p_i. For Boolean queries, such cardinality can be estimated by Boolean selectivity, that is, the ratio of the number of data objects which evaluate the given predicate to be true, obtained from some statistical samples, such as histograms. For instance, in a simple conjunctive query, N_i is simply the product of the predicate selectivities of those evaluated prior to p_i, multiplied by the database size N, assuming predicate independence.

Similarly, for top-k algorithms, we can estimate the cost based on our selectivity estimation from statistical samples. However, unlike Boolean queries composed of relational operators, the aggregate effect cannot be computed analytically, as predicates are aggregated by arbitrary function $\mathcal{F}$. To estimate this arbitrary aggregation, we generalize Boolean selectivity into the notion of *aggregate selectivity* of a set of evaluated predicates $\mathcal{P}$, which is the ratio of the number of objects whose aggregate scores that still make to the top-k answers. More formally, let θ be the lowest score of the top-k results (which we will not know a priori until the query is fully evaluated). Observe that $\overline{\mathcal{F}}_{\mathcal{P}}[u]$ will *eventually* be on the top-k if $\overline{\mathcal{F}}_{\mathcal{P}}[u] \geq \theta$ (since only the final answers will surface to and remain on the top). We thus define the aggregate selectivity $\mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}, \Delta)$ for a set of accesses $\mathcal{P}$ as the ratio of the number of database objects u that "pass" $\overline{\mathcal{F}}_{\mathcal{P}}[u] \geq \theta$ after sorted access up to depth Δ (this selectivity notion, unlike its Boolean-predicate counterpart, depends on the aggregate filtering effect of all the predicates evaluated).

With this notion, we can estimate the cost of random accesses after the preceding sorted access phase (we will later discuss how to estimate the cost of sorted accesses up to Δ). In other words, given $\mathcal{H} = (p_1, \ldots, p_m)$, the aggregated selectivity $\mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta)$ of the first $i-1$ predicates in $\mathcal{H}$, denoted as subschedule $\mathcal{H}_{i-1} = (p_1, \ldots, p_{i-1})$, is $N \cdot \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta)$ and so is the number of random accesses on p_i. The cost of the random access phase is thus

$$\sum_{i=1}^{m} n \cdot \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta) \cdot cr_i = n \cdot \sum_{i=1}^{m} \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta) \cdot cr_i.$$

We can now formulate the overall cost, adding the cost of sorted access phase. More specifically, denoting the number of objects with p_i score of no less than δ_i as $n(p_i, \delta_i)$, the overall cost $\mathcal{C}((\Delta, \mathcal{H}))$ can be formulated as follows:

$$\mathcal{C}((\Delta, \mathcal{H})) = \sum_{i=1}^{m} n(p_i, \delta_i) \cdot cs_i + n \cdot \sum_{i=1}^{m} \mathcal{S}^{\theta}_{\mathcal{F}}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta) \cdot cr_i.$$

Such cost can be estimated by "simulating" the aggregate effect on the statistical samples, or mimicking the actual execution with the retrieval size k' proportional to the sample size s, namely, $k' = \lceil k \cdot \frac{s}{n} \rceil$. We first mimic the sorted access phase by performing sorted accesses on these samples (we discuss how to get such samples later) up to the depth Δ so as to get the estimated cost of sorted accesses.[8] We then use the samples to estimate the aggregate selectivity and thus random access costs. In principle, such samples can be obtained from online sampling (i.e, randomly sample the database at runtime), or statistics-based synthesis. In the worst case, when online sampling is unavailable or too costly, or when a priori statistics are not available, we can still generate "dummy" synthesis based on some assumed distribution (e.g., uniform) as a crude approximation. Though such samples cannot well represent actual score distributions, they help optimize for other important aspects, as we empirically show in Section 9. While our optimizer will certainly benefit from more accurate samples and statistics, Section 9 will implement our optimization framework using dummy synthesis so as to validate our framework in the worst-case scenario.

7.3 Search

Toward our goal of identifying the optimal algorithm $(\Delta_{opt}, \mathcal{H}_{opt})$, we decompose the problem into the two subtasks of identifying Δ_{opt} first and $\mathcal{H}_{opt}$. However, as Δ-optimization and $\mathcal{H}$-optimization are mutually dependent, the process is iterative. Specfically, we can identify the optimal Δ with respect to some random access scheduling $\mathcal{H}_0$ to start with. Denoting this identified optimal depth as Δ_0, we then find the optimal random access scheduling $\mathcal{H}_1$ with respect to Δ_0, which may not be identical to $\mathcal{H}_0$. In other words, due to the mutual recursiveness in Δ- and $\mathcal{H}$-optimization, finding an optimal pair of $(\Delta_{opt}, \mathcal{H}_{opt})$ is essentially continuing the aforementioned iterations until reaching the minimal-cost pair, that is, hill climbing (note that this search can lead to a local minimum when there are multiple local minima). However, for simplicity, our two-phase approach was designed to perform a one-iteration approximation of the aforementioned iterative processes. In other words, we find Δ_{opt} with respect to some initial random access scheduling $\mathcal{H}_0$ and find $\mathcal{H}_{opt}$ with respect to Δ_{opt} identified in the first phase. We believe this approximation gives an already good solution, considering both the cost of optimization and benefit gained, as we empirically validate in Section 9.

— Δ-optimization: We first identify the optimal depth Δ_{opt} with respect to some initial schedule $\mathcal{H}_0$, namely, $\Delta_{opt} = argmin_{\Delta}\mathcal{C}((\Delta, \mathcal{H}_0))$.
— $\mathcal{H}$-optimization: We then identify the optimal scheduling $\mathcal{H}_{opt}$ with respect to Δ_{opt} identified.

For $\mathcal{H}$-optimization, we adopt the global predicate scheduling proposed in our preliminary study [Chang and Hwang 2002] that uses online sampling to identify a predicate scheduling with the highest aggregated filtering effect (discussed in Section 7.2). For Δ-optimization, we first study how it is specific to runtime factors, for example, score functions, predicate score distributions, and cost scenarios, as Example 8 will illustrate.

[8]Here, Δ corresponds to a set of threshold scores to reach these scores will stay the same for samples of any size, as long as they preserve the statistical properties of the dataset.

Example 8 (*Δ-Optimization Possibilities*). To illustrate, we continue Example 7 with a different depth configuration $\Delta_2 = (0.8, 1)$. In fact, Δ_2 generates the algorithm illustrated in Figure 10: It starts with sa_1 as $\overline{p_1} > \delta_1$, but chooses $ra_2(u_3)$ next as $\overline{p_2} \leq \delta_2$.

Observe from this example that different configurations imply different access costs: While a *parallel* configuration of $\Delta_1 = (0.8, 0.8)$ requires four accesses to answer Q_1 (Figure 11), a focused configuration $\Delta_2 = (0.8, 1)$ requires only two accesses (Figure 10). However, note that this finding is only specific to Q_1. For instance, when scoring function $\mathcal{F}$ is *avg* (the average function) for the same query Q_1, Δ_1 requires less accesses (four) than Δ_2 (six).

Consequently, we need search schemes that systematically adapt to the given query for exploring the Δ-space, namely, the m-dimensional space of $\delta_1 \times \ldots \times \delta_m = [0:1]^m$. In particular, we propose three search schemes. First, we implement an approximate exhaustive search Naive over this Δ-space by discretizing each δ_i into a set of x finite values in the range [0:1] and estimating the cost for all possible x^m combinations. Second, we then develop two informed search schemes, Strategies and HClimb, which guide the search by query-driven or generic hill-climbing strategies, respectively. Among these three schemes, we focus on HClimb in particular, which is general to any query, yet evaluated to be the most efficient and effective from our unreported evaluation. From a random starting point, HClimb simply searches toward neighbors with less estimated cost (see Section 7.2 for cost estimation), until it reaches the cost-minimal configuration. The scheme is typically enhanced with multiple random starting points so as to avoid local minimum.

8. UNIFICATION AND CONTRAST

Framework NC,[9] in addition to being a general and adaptive optimization framework, enables the conceptual unification of existing algorithms. It complements existing algorithms by: (1) generating similar behaviors when their behaviors are desirable, while (2) optimizing beyond their "static" behaviors when not desirable. In particular, we first discuss how this unifies and contrasts with TA [Fagin et al. 2001] (Section 8.1). We then discuss how it is based on and extends our preliminary work MPro [Chang and Hwang 2002] (Section 8.2).

Since most existing algorithms (as originated in a middleware context) assume no wild guesses [Fagin et al. 2001], in order to be more comparable, we transform NC to handle this restriction (while NC can generally work with or without such an assumption). In such settings, an algorithm cannot refer to an object u (for random access) before "knowing" it from some sorted access. Thus NC must distinguish between seen and unseen objects. In other words, each object u is marked as *unseen* until hit by some sorted access and becoming *seen*. In particular, we implement this distinction by introducing a *virtual* object unseen to represent all unseen objects, as all unseen share the same maximal-possible score $\overline{\mathcal{F}}[\text{unseen}] = \mathcal{F}(\overline{p_1}, \ldots, \overline{p_m})$. This virtual object needs

[9]For notational simplicity, we use NC interchangeably as both an abstract framework and the optimal algorithm generated.

step	$\overline{p_1}$	$\overline{p_2}$	$\mathcal{K}_p$	alternatives	*Select*
1.	1	1	unseen	$N_{\text{unseen}} = \{sa_1, sa_2\}$	sa_1
2.	0.7	1	u_3	$N_3 = \{sa_2, ra_2(u_3)\}$	$ra_2(u_3)$

Fig. 13. NC with no wild guess.

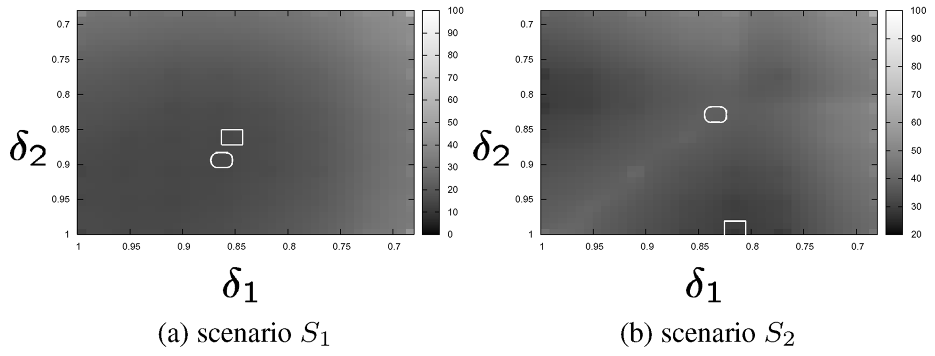


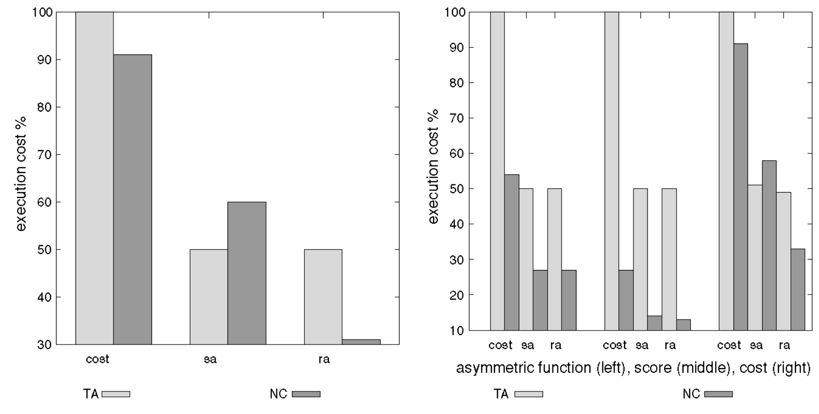
Fig. 14. Illustrations of TA and NC.

special handling, as Figure 13 shows with query Q_1: Initially, all objects are unseen, hence NC initializes $\mathcal{K}_{\mathcal{P}}$ with only the unseen. Then, when this unseen is at the top (e.g., Step 1), its induced choices N_{unseen} will contain only sorted accesses, since random access is not allowed for an unseen object, by the no wild guesses assumption. Finally, objects hit by some sorted access will become seen (e.g., u_3 seen by sa_1 at Step 1), and once seen, are treated the same as in NC (without the no wild guesses assumption) and may surface to $\mathcal{K}_{\mathcal{P}}$ (e.g., u_3 at Step 2).

8.1 Algorithm TA

We now observe how NC unifies and contrasts with TA, which is an early and probably the most representative existing top-*k* algorithm of all. As Figure 2 lists, TA is designed for access scenarios where both sorted and random access have uniform unit costs, namely, $\frac{cr_i}{cs_i} \approx 1$. Let's call it the uniform scenario. As illustrated in Example 2, the behaviors of TA can be characterized as follows: (1) *Equal-depth-sorted access*— At each iteration, it performs sorted access to all predicates; (2) *exhaustive-random access*— it then does exhaustive random access on every seen object; and (3) *early-stop*— it terminates as soon as *k* evaluated objects score no less than the upper bound score of unseen objects. We now ask: When such behaviors are desirable, would NC generate similar behaviors (unification)? When not, would NC optimize beyond such static behaviors (contrast)?

Unification. In "symmetric" cases, where each predicate contributes rather equally to both overall score and cost, such as, $\mathcal{F} = avg$ with equal predicate access costs, NC will indeed generate a TA which is built for such a scenario in mind. We illustrate this through a scenario S_1 with scoring function $\mathcal{F} = avg\ (p_1, p_2)$, in which the scores of p_1 and p_2 are uniformly distributed over



(a) scenario S_1 **(b)** over asymmetric scenarios

Fig. 15. Comparison of TA and NC when $sa = \sum_i S_i \cdot cs_i$ and $ra = \sum_i R_i \cdot cr_i$, namely, $cost = sa + ra$.

[0 : 1] and $\forall i : cs_i = cr_i = 1$. To observe how NC adapts to S_1, Figure 14(a) shows a contour plot of $\mathcal{C}(\Delta, \mathcal{H}_0)$ with respect to $\Delta = (\delta_1, \delta_2)$. NC identifies the minimal-cost Δ_{opt}, or the darkest cell marked by a rectangle, at around (.85, .85). For comparison, the figure also marks the depth TA reaches by an oval, at around (.87, .87).

Observe that the two algorithms are indeed almost identical: (1) Both perform *equal-depth-sorted access* up to similar depths; (2) by accessing the same depths, they will both see the same set of objects. However, since NC does not use exhaustive random access, it will perform less random accesses than TA. We thus expect NC to be slightly better overall; and (3) the output $\mathcal{K}$ of NC shares the same *early-stop* condition as TA: It terminates when k evaluated objects score no less than the upper-bound score of unseen objects.

Contrast. However, NC contrasts with TA by being able to adapt. Even among uniform scenarios, in the asymmetric cases, TA's characteristic behaviors cannot adapt well. To illustrate such cases, Figure 14(b) shows scenario S_2, which replaces the scoring function of S_1 to $\mathcal{F} = min$ (and otherwise, the same as S_1). In contrast to *avg*, where every predicate score symmetrically contributes to the overall score, *min* is asymmetric in the sense that only one predicate with the minimum score determines the overall score of each object. Observe how NC adapts beyond TA and thus generates a rather different algorithm with less cost (thus a darker cell) in this scenario: NC focuses sorted access on p_1 with $\Delta_{opt} = (.81, 1)$, while TA performs equal-sorted access up to (.83, .83).

For a closer observation, Figure 15 compares the relative access costs of TA and NC (normalized to the total cost of TA as 100%) in various scenarios: As symmetric cases, Figure 15(a) first considers scenario S_1, which is rather favorable to TA (as explained earlier). In such a case, both algorithms behave similarly (except that NC slightly outperforms TA by going deeper in sorted accesses to trade future random accesses). By contrast, Figure 15(b) considers scenarios that introduce asymmetry, one-at-a-time, upon S_1, as follows:

—*Asymmetric function:* Unlike for a symmetric function like $\mathcal{F} = avg$ in S_1, where each predicate equally contributes $\mathcal{F}$, the optimal configuration for an asymmetric function tends not to be equal-depth; when $\mathcal{F} = min$, NC adapts to focus sorted access on one predicate.

—*Asymmetric scores:* Unlike in scenario S_1, predicate score distributions may differ significantly. When the distribution of p_2 is replaced by normal distribution with mean .2 and variance .1, NC adapts to perform more sorted accesses on p_2 (which is more selective to distinguish objects by scores).

—*Asymmetric costs:* When certain predicates are more expensive (e.g., web-accessible predicate), NC adapts to replace such expensive accesses by more economic alternatives: When p_1 is three times more expensive (for both sorted and random access) than p_2, NC outperforms TA by favoring accesses on p_2.

8.2 Algorithm MPro

We next discuss how NC is based on and extends our preliminary work MPro [Chang and Hwang 2002] for a simpler random-only scenario (similar to Upper [Bruno et al. 2002] and TA_Z [Fagin et al. 2001], as we will discuss later). Consider $\mathcal{F}(p_1, \ldots, p_m, p_{m+1}, \ldots, p_n)$. MPro distinguishes two types of predicates: While $p_{m+1}, \ldots, p_n$ are simply *ordinary* (or *indexed*) predicates, the other group $p_1, \ldots, p_m$ are *expensive* predicates, and either *probe-only* or *random-only*.

—$\forall p_i \in \{p_1, \ldots, p_m\}$: p_i supports only random access with unit cost cr_i; thus $cs_i = \infty$.

—$\forall p_i \in \{p_{m+1}, \ldots, p_n\}$: p_i supports both random access and sorted access, with unit cost cr_i and cs_i, respectively.

MPro

aims at minimizing random accesses, or per-object probes on expensive predicates. In brief, this works as follows:

(1) Merge the ordinary predicates $p_{m+1}, \ldots, p_n$ into one single list x (or a conceptual predicate), using TA. By this merging, we view the scoring function as $\mathcal{F}(p_1, \ldots, p_m, x)$.

(2) Sort all objects u by their maximal-possible score $\overline{\mathcal{F}}[u]$ (with respect to its evaluated predicates). Let $\mathcal{K}_\mathcal{P}$ be the current top-*k* objects.

(3) At each iteration, pick an incomplete object v_j from $\mathcal{K}_\mathcal{P}$. Let $P_j = \{ra_i(v_j) | p_i[v_j]$ be unevaluated so far$\}$. Pick a probe $ra_i(j)$ from P_j, according to some predicate schedule $\mathcal{H}$, and execute this. Terminate and return $\mathcal{K}_\mathcal{P}$ when all top-*k* objects are complete.

In essence, MPro has the following characteristic behaviors: (1) *x-separation*: It separates the sorted access-capable predicates, namely, $p_{m+1}, \ldots, p_n$, from the rest, namely, $p_1, \ldots, p_m$, by isolating the former and merging them into x by TA. (2) *x-stop*: It will retrieve from the merged x-list in the sorted order and stop as soon as $\forall v \in \mathcal{K}_\mathcal{P} : \mathcal{F}[v] \geq \mathcal{F}(1, \ldots, 1, \overline{p_{m+1}}, \ldots, \overline{p_n})$, where $\mathcal{K}_\mathcal{P}$ is the final top-*k* answers, and the depths $\overline{p_{m+1}}, \ldots, \overline{p_n}$ are determined by the merging algorithm TA. (3) *p-minimization*: For these retrieved objects, MPro will minimize probe cost by finding an optimal $\mathcal{H}$.

Unification. As MPro aims at minimizing random accesses, we can see that NC can generate MPro if "projected" *only* to expensive predicates (with this projection, we ignore x-separation at this point, which we will revisit later.)

First, NC will satisfy the same x-stop condition: Note that the unseen object from the x-list has $\overline{\mathcal{F}}$[unseen] = $\mathcal{F}(1, \ldots, 1, \overline{p_{m+1}}, \ldots, \overline{p_n})$, and thus the same stop-condition holds.

Second, NC will naturally perform the same p-minimization: As outlined before, for probe-only predicates, MPro essentially operates on the same machinery as NC, that is, sorting by maximal-possible scores, further probing on some incomplete v_j in $\mathcal{K}_\mathcal{P}$, and stopping when $\mathcal{K}_\mathcal{P}$ completes. For such incomplete v_j, MPro constructs a set of P_j of random-only accesses for further probing, corresponding to alternatives of NC. Meanwhile, NC as a general mechanism, constructs N_j for incomplete object v_j, including *both* sorted and random accesses (Line 5, Figure 9). However, as these probe-only predicates do not support sorted accesses, namely, $cs_i = \infty$, the optimizer (Section 7) will then algorithmically "ignore" the sorted accesses, and focus only random accesses as MPro does, by configuring the sorted access depths as Δ: ($\delta_1 = 1, \ldots, \delta_m = 1$), that is, no sorted access at all. In summary, NC adapts to achieve the same p-minimization, while using a general-purpose mechanism.

Contrast. Although NC can generate MPro, the two algorithms differ fundamentally for ordinary predicates supporting both sorted and random accesses. While NC integrates such predicates in their optimization, MPro isolates (and thus ignores) such predicates by x-separation. By using TA as a black box for merging $p_{m+1}, \ldots, p_n$, MPro will suffer the restrictions of TA over NC, just as discussed in Section 8.1.

Remark. To summarize, NC generalizes beyond MPro significantly by generally handling both sorted and random accesses. Such a generalization is nontrivial. Essentially, as Section 4 identified, sorted access is fundamentally different with its progressiveness and side-effects. By focusing only on random accesses, MPro does not deal with defining a complete framework. To illustrate, consider a random-only setting, when some $v_j \in \mathcal{K}_\mathcal{P}$ is incomplete with possible future accesses identified as $P_j = \{ra_1(v_j), ra_2(v_j)\}$. By contrast, in its sorted-also counterpart (by adding sorted accesses) the possible future accesses are $N_j = \{sa_1, ra_1(v_j), sa_2, ra_2(v_j)\}$.

To contrast the two, we identify the following challenges in defining a complete framework:

(1) *Sorted access side-effects*. In random-only, such v_j can be *univocally* identified as *required* for further processing. If not picked, v_j will remain necessary forever. However, in sorted-also, v_j may become unnecessary (by retiring from the current top-k), simply by the side-effects from accessing *others*.

(2) *Sorted access progressiveness*. In random-only, for v_j just picked, with respect to a given schedule (e.g., $\mathcal{H} = (ra_2(v_j), ra_1(v_j)))$, the next probe (e.g., $ra_2(v_j)$ in P_j) can be univocally determined to be required. However, in sorted-also, it is not clear exactly what to schedule. To illustrate, as every

sorted access can repeat for progressive accesses, there are generally an infinite number of possible schedules; for example, for N_j: (sa_1, sa_2, $ra_2(v_j)$, $ra_1(v_j)$), (sa_1, sa_1, sa_2, $ra_2(v_j)$, $ra_1(v_j)$), etc.

In summary, NC generalizes MPro, which focuses only on random-only and thus reduces the optimization to a "bare-bone" of finding the optimal predicate scheduling $\mathcal{H}$. In these limited scenarios, the aforementioned Properties 1 and 2 together univocally determine a required probe on a particular v_j for a particular p_i, namely, *the necessary-probe principle* [Chang and Hwang 2002]. Targeting the same scenarios, Upper applies the same principle of evaluating that object with the highest maximal score. However, runtime adaptation heuristics further restrict its applicability to weighted average scoring functions, in addition to its limitation to random access optimization. TA_Z similarly evaluates the object with the highest maximal score, but lacks predicate scheduling and thus always evaluates objects completely, which explains consistently worse performances compared to Upper (as reported in Bruno et al. [2002]).

In contrast to MPro, Upper, and TA_Z, the notion of necessary choices of our NC framework enables handling both random and sorted accessby identifying the necessary tasks that must be satisfied (Theorem 1) and scheduling only such tasks (Theorem 4).

9. EXPERIMENTS

This section reports our experiments. Our goal is two-fold: First, to validate the adaptivity scalability, generality, and of NC, Section 9.1 studies its performance over a wide range of middleware settings by simulating extensive synthetic scenarios. Second, to validate practicality, Section 9.2 quantifies the absolute performance of NC over real web sources. Our experiments were conducted with a Pentium III 933MHz machine with 256M RAM, using our implementation of NC in Python. Note that for runtime search, we use "dummy" synthesis, assuming uniform distributions for all predicates (as discussed in Section 7.2). While our optimizer will certainly benefit from more accurate sampling, we implement using dummy synthesis (unless noted otherwise) to validate our framework in the worst-case scenario, where sampling is expensive or even infeasible.

9.1 Synthetic Middleware Scenarios

In this section, we perform extensive experiments to study the adaptivity, scalability, and generality of NC over various performance factors. To isolate and control these factors, our experiments in this section used synthetic datasets. In particular, we synthesize our scenarios varying the following performance factors: (1) unit costs cs_i and cr_i, (2) the scoring function $\mathcal{F}$, and (3) score distribution D_i, of each predicate p_i.

Over varying performance factors, our goal is to compare NC with the existing algorithms listed in the matrix of Figure 2. In particular, we compare to TA, CA, and NRA, as the rest of algorithms in the matrix are either not applicable to the scenarios or subsumed by the algorithms considered: First, although

retrieval size (k)	100
number of predicates (m)	2
predicate score distribution	uniform
database size (n)	10,000
scoring function ($\mathcal{F}$)	$avg(p_1, \ldots, p_m)$
unit costs	$\forall i : cs_i = cr_i = 1$
score distributions	$\forall i : D_i = unif$

Fig. 16. Default setting.

$(h_1 = \frac{cr_1}{cs_1}, h_2 = \frac{cr_2}{cs_2})$	$cs_2 > cr_2$	$cs_2 = cr_2$	$cs_2 < cr_2$
$cs_1 > cr_1$	$(\frac{1}{r}, \frac{1}{r})$	$(\frac{1}{r}, 1)$	$(\frac{1}{r}, r)$
$cs_1 = cr_1$	$(1, \frac{1}{r})$	$(1, 1)$	$(1, r)$
$cs_1 < cr_1$	$(r, \frac{1}{r})$	$(r, 1)$	(r, r)

Fig. 17. Unit cost scenarios.

Algorithm Quick-Combine, SR-Combine, and Stream-Combine are designed for the same scenarios, their limited runtime optimization heuristics (i.e., using the partial derivative of $\mathcal{F}$) restrict their applicability over our synthesized scenarios, including nondifferentiable functions as well, for example, $\mathcal{F} = min$ in Q_1. Second, the rest of algorithms can be considered as special cases of NC: As we explained earlier in Section 8.2, NC unifies MPro, Upper, and TA_Z, designed specifically for simpler probe-only scenarios. As NC generalizes MPro and therefore performs identically, we do not compare MPro with NC.

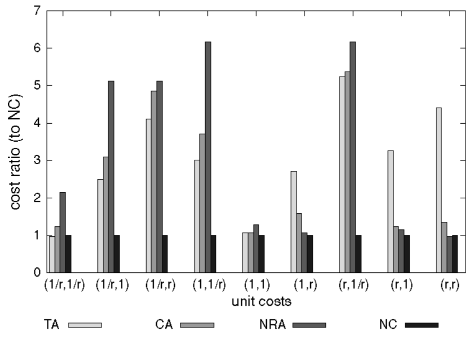
Adaptivity of NC. To validate the adaptivity of NC over existing algorithms, we first compare the performance of the four algorithms, varying one parameter at a time, as follows over the default setting described in Figure 16:

—*Unit costs*. To understand the impact of varying unit costs, we categorize cost scenarios into $cs_i > cr_i$, $cs_i = cr_i$, and $cs_i < cr_i$ for each predicate p_i. In particular, for $m = 2$, Figure 17 shows how we synthesize such scenarios by varying $h_i = \frac{cr_i}{cs_i}$ to $\frac{1}{r}$, 1, and r.

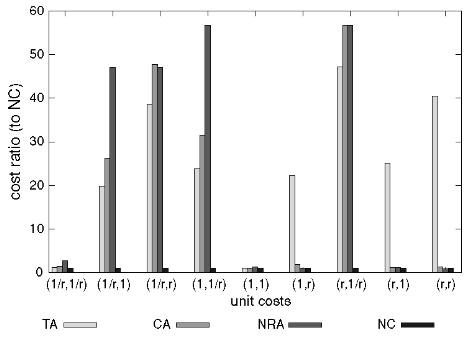
—*Scoring function.* To understand the adaptivity over various scoring functions, we evaluate over $\mathcal{F}$: *min*, *wavg* (weighted average), and *gavg* (geometric average). For weighted average $wavg_c = w_1 \cdot\ p_1 + w_2 \cdot p_2$, we vary $c = \frac{w_2}{w_1}$ to 1 and 10 when $\sum_i w_i = 1$. Similarly, we vary c to 1 and 10 for geometric average $gavg_c = p_1^{w_1} \cdot p_2^{w_2}$, when $\sum_i w_i = 1$.

—*Score distribution.* To understand the impact over different distributions, we change D_2 to normal distribution and vary its mean to .2, .5, and .8, with the variance as .16.

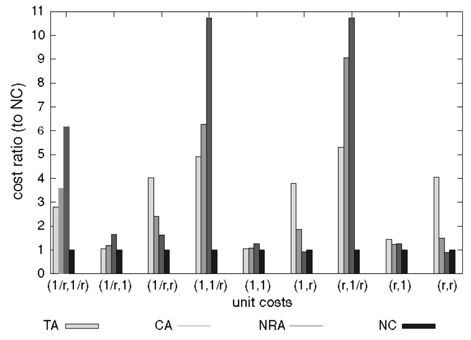
Figure 18 reports our evaluation results over various cost scenarios (as enumerated in Figure 17) when $cs_1 = 10$ and $cs_2 = x \cdot 10$, while cr_1 and cr_2 are determined by each scenario as $h_1 \cdot cs_1$ and $h_2 \cdot cs_2$, respectively. First, Figure 18(a) compares the average total access cost (relative to the cost of NC) when $x = 1$ (i.e., $cs_1 = cs_2$) and $r = 10$. Observe that NC is robust across all scenarios and significantly outperforms existing algorithms in most scenarios by generally adapting to various cost situations. For instance, when $(h_1, h_2) = (1, \frac{1}{10})$, NC saves 67%, 73%, and 84% from the cost of TA, CA, and NRA, respectively. In



(a) $x = 1, r = 10$



(b) $x = 1, r = 100$

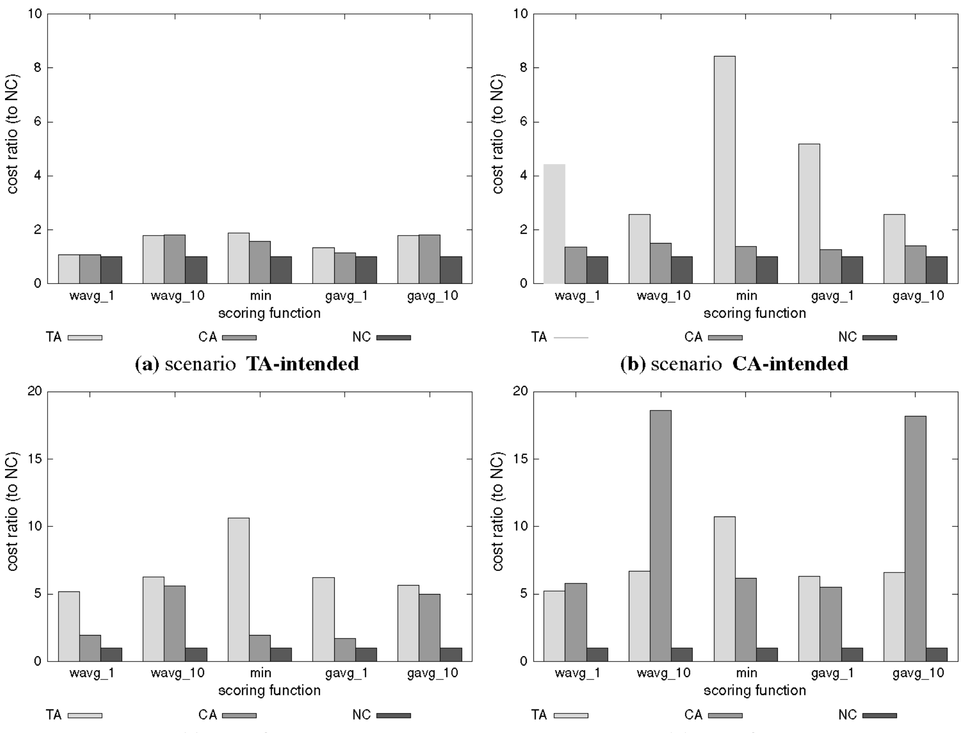


(c) $x = 10, r = 10$

Fig. 18. Adaptivity of NC over unit costs.

addition to the robust performances of NC, it is also interesting to observe how NC unifies existing algorithms. For instance, in uniform cost scenarios, for example, $(h_1, h_2) = (1, 1)$, NC will similarly perform equal-depth sorted accesses (Section 8) and perform comparably to TA, which was specifically designed for such a scenario. For scenarios where random access is expensive, which are ideal for CA, for example, $(h_1, h_2) = (10, 10)$, NC unifies the ideal existing algorithms CA and NRA and performs comparably by similarly trading some expensive random accesses with more economical sorted accesses.

However, in other scenarios, NC significantly outperforms existing algorithms by orders of magnitude. To illustrate, Figures 18(b) and (c) repeat the same sets of experiments, changing to $r = 100$ and $x = 10$, respectively. Figure 18(b) reports results when the asymmetry across the access costs of sorted and random accesses are more significant, namely, when $r = 100$. Observe that the cost savings of NC is even more significant in these scenarios: When $(h_1, h_2) = (1, \frac{1}{100})$, NC saves 96%, 97%, and 98% from the cost of TA, CA, and NRA, respectively, as these existing algorithms cannot adapt to such asymmetry in costs. Similarly, compared to Figure 18(a), the cost savings of NC is more significant in Figure 18(c), where the asymmetry of access costs across predicates is more significant, that is, $x = \frac{cs_2}{cs_1} = 10$. Note that when $(h_1, h_2) = (1, \frac{1}{10})$, NC saves 75%, 85%, and 91% from the cost of TA, CA, and NRA, respectively, by adapting to cost asymmetry, as similarly observed in previous evaluations. In summary, NC performs robustly across wide ranges of scenarios and



(a) scenario **TA-intended**

(b) scenario **CA-intended**

(c) scenario **A1**

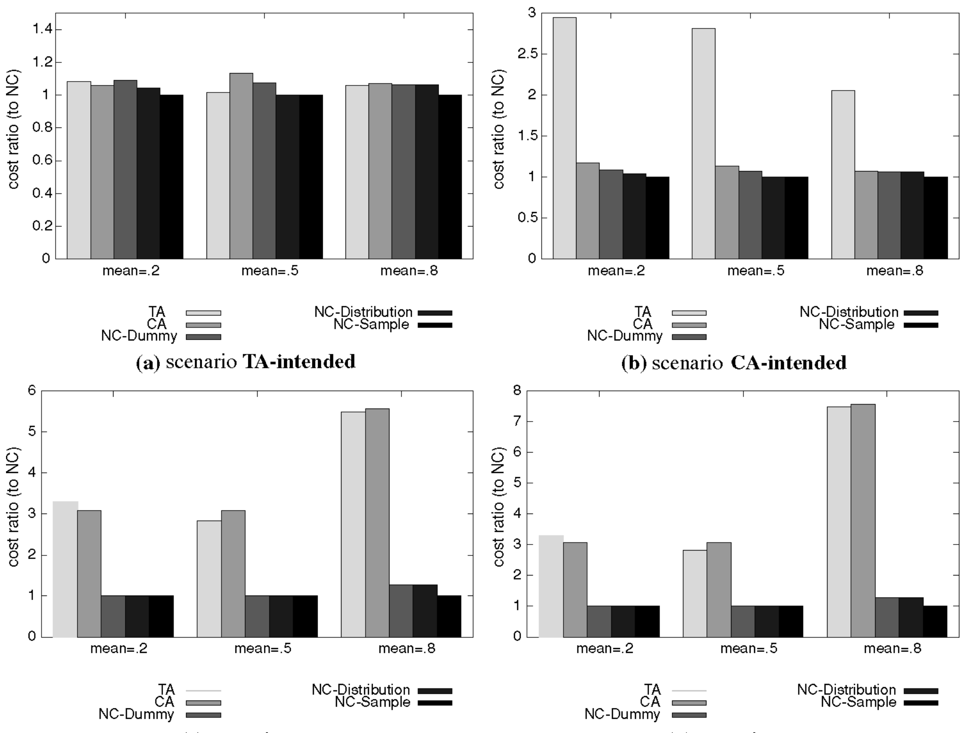
(d) scenario **A2**

Fig. 19. Adaptivity of NC over scoring functions.

significantly outperforms existing algorithms in asymmetric scenarios, as consistently observed in Section 8.

We now study how NC adapts to other cost factors, namely, scoring functions and score distributions. In particular, we vary such factors in four representative cost scenarios that best depict the unification and contrasting behaviors of NC to existing algorithms. First, to show unification behaviors, we pick the intended scenarios for TA and CA, that is, $(h_1, h_2) = (1, 1)$ and (r, r), respectively, which we denote as scenarios **TA-intended** and **CA-intended**. Second, to show our contrasting behaviors, we pick asymmetric scenarios $(h_1, h_2) = (r, \frac{1}{r})$ when $cs_1 = cs_2$ (denoted as scenario **A1**) and $cs_1 = 10 \cdot cs_2$ (denoted scenario **A2**). For the sake of presentation, we drop the comparison with NRA, which has shown mostly worse performance to CA in the previous evaluations reported in Figure 18.

Figure 19 evaluates the adaptivity of NC over varying scoring functions in the default setting with $x = 1$ and $r = 10$ (as in Figure 18(a)). As Section 8 discussed and also observed here, when the function is symmetric, such as when, $\mathcal{F} = avg$, NC unifies the behavior of TA, which is specifically designed for such scenarios. However, in all other functions with asymmetry, for example, $\mathcal{F} = wavg$ or $\mathcal{F} = gavg$, NC outperforms existing algorithms. Observe that cost differences grow significantly as such the asymmetry of the weighted and geometric average increases. For instance, the weight ratio $c = \frac{w_2}{w_1}$ increases



(a) scenario **TA-intended**

(b) scenario **CA-intended**

(c) scenario **A1**

(d) scenario **A2**

Fig. 20. Adaptivity of NC over score distributions.

from 1 to 10 for both $wavg_c = w_1 \cdot p_1 + w_2 \cdot p_2$ and $gavg_c = p_1^{w_1} \cdot p_2^{w_2}$ increases when $\sum_i w_i = 1$. The same observation holds across different cost scenarios, and the cost savings are more significant when there exists additional asymmetry in costs, either across different access types (e.g., **CA-intended**) or different predicates (e.g., **A1** and **A2**).

Figure 20 studies the adaptivity of NC over varying score distributions in the default setting with $x = 1$ and $r = 10$ (as in Figure 18(a)). Recall that in all our evaluations, we employed NC using dummy synthesis (Section 7.2). Meanwhile, in this experiment, to observe the impact of sampling accuracy, we also implement two versions of NC with more representative samples, namely: (1) NC-Sample, which uses actual samples of 0.1%, in data size, and (2) NC-Distribution, which uses synthetic "approximate" samples generated using a priori knowledge of the actual distribution (e.g., normal distribution with mean as 0.8 and variance as 0.16). Figure 20(a) studies the performance of NC when $\mathcal{F} = avg$. Observe that more accurate samples are indeed helpful to adapt more closely on the score distributions. For instance, when the mean is 0.8, in all cost scenarios, NC-Sample and NC-Distribution can adapt better than NC-Dummy by performing more sorted accesses on p_1, which is more selective in distinguishing objects by scores. Meanwhile, note that dummy synthesis is equally effective in optimizing for *other* important cost factors. Observe that in scenarios **A1** and **A2**, while NC-Dummy does not know the actual distribution,

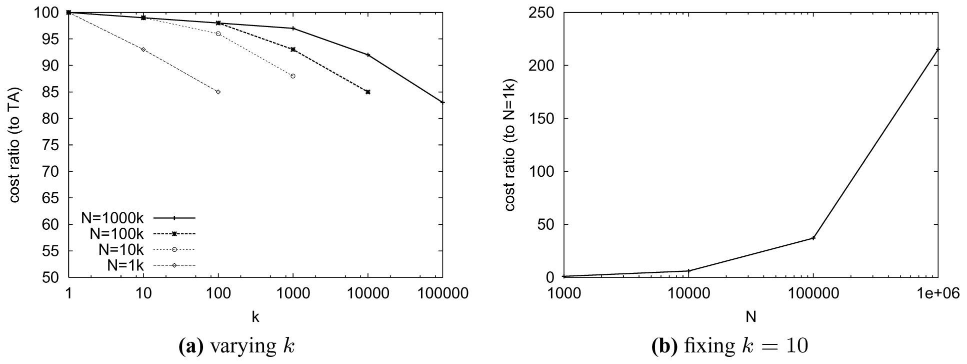


Fig. 21. Scalability of NC over N.

it significantly outperforms existing algorithms by optimizing to other runtime factors and performs very closely to NC with more accurate statistics.

Scalability of NC: Figure 21 studies the scalability of NC varying N and k from the default setting in Figure 16. In particular, we choose to focus on scenario **TA-intended**, the most difficult scenario, as the cost savings of NC was minimal in the previous experiments. Figure 21(a) reports the cost ratio of NC with respect to TA, varying $N = 1k, 10k, 100k$, and $1,000k$ in the default setting with $x = 1$ and $r = 10$. It is interesting to observe that, the cost ratios are approximately the same when the relative retrieval size is the same, regardless of database size. For instance, for $k = N \cdot 1\%$ (i.e., $k = 10, 100, 1,000$, and $10,000$ for $N = 1k, 10k, 100k$, and $1,000k$), the cost ratios are all around 95%. This observation shows that NC has good *data scalability*, that is, the cost ratio for finding k answers from a database of size sN will be less than s times that for finding the same number of answers from a database of size N. Such data scalability is promising, considering that the retrieval size k tends to stay small regardless of N, for example, users retrieving only the first page or two of search results, regardless of the number of hits. The same evaluation also suggests that when $k = 10$, as N increases from $N = 1k$ to $N = 10k, 100k$, and $1,000k$, namely, $10-$, $100-$, and $1,000$-fold, the cost increases sublinearly by $6-, 37-$, and 215-fold, respectively, as Figure 21(b) presents.

Similarly, Figure 22 presents the cost ratio of NC with respect to the cost of TA when varying $m = 2, 3$, and 4 in the default setting with $x = 1$ and $r = 10$. Note that the cost ratio significantly decreases as m increases, which suggests the scalability of NC over m. In other words, there is more room to improve existing works when m is large, as the performance of NC illustrates.

Generality of NC: To validate the generality of NC over existing algorithms, we study the performance NC over 1,000 synthetic scenario configurations. In particular, each configuration consists of randomly generated parameters representing unit costs, scoring functions, and scoring distributions. First, for costs, we randomly generated cr_i and cs_i in the range of [1:100] units (to better accommodate CA, we generate cost configurations according to its target scenarios of expensive random access by enforcing $cr_i > cs_i$). Second, for scoring functions, we randomly generate the weight w_i in [1:100] of $\mathcal{F} = wavg$ for each

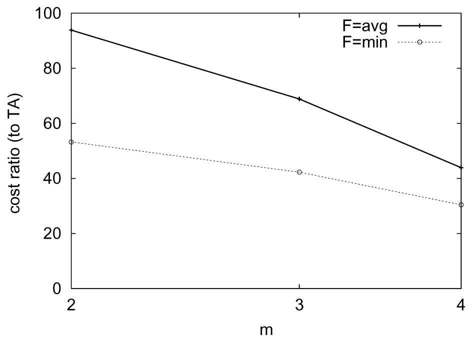


Fig. 22. Scalability of NC over m.

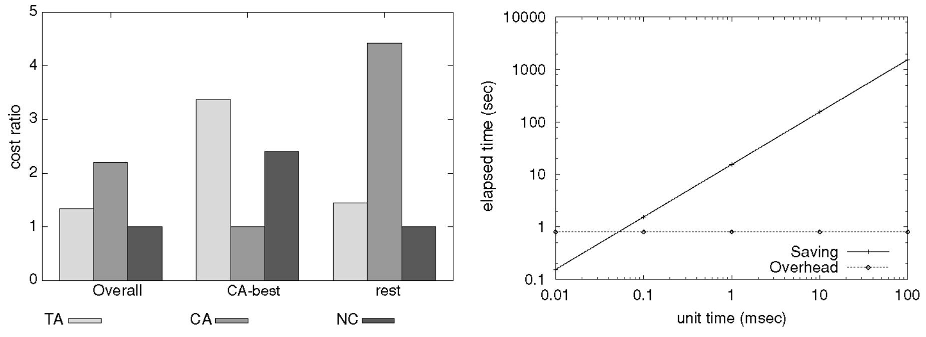


Fig. 23. Generality of NC.

configuration. Lastly, for score distributions, we randomly generate the mean and variance of a normal distribution.

Figure 23(a) compares the average total access cost of the three algorithms over such 1,000 random configurations. Observe that overall, NC as a unified algorithm saves over TA and CA by 25% and 55%, on average, by succesfully adapting to a combination of cost factors. We then, for closer observation, divide the 1,000 settings into two groups: those in which CA works best (about 25% of the configurations, which we denote as "CA-best"), and the rest. From such groupings, we can observe how NC unifies and contrasts with existing algorithms. First, regarding unification, observe that NC behaves similarly to CA in CA-best scenarios (the middle-bar group of Figure 23(a)), with a smaller cost difference to CA than has TA. Second, regarding contrast, observe that NC outperforms existing algorithms in the majority of the "rest" scenarios (i.e., 75% of the configurations represented by the rightmost group in Figure 23(a)).

Lastly, we study whether the overhead of runtime optimization justifies its cost savings in access costs. Clearly, such optimization is only justified if cost savings outweigh the computation overhead of optimization. To analyze this tradeoff, we also compare savings with the overhead. In particular, as the savings of access cost will have different impacts, depending on the actual response time t of a single unit cost, Figure 23(b) compares these saving and overhead with respect to t (note that we compare only with CA, as it outperformed TA in Figure 23(a)). Observe that the optimization overhead of NC can be justified in a large range of realistic scenarios, such as, when the unit time t is larger than 0.05ms. We believe this range ($\geq 0.05ms$) covers most middleware scenarios, as they are characterized by nontrivial access costs.

9.2 Real Web Scenarios

To validate the practicality of NC, we study its absolute performance over real-life web sources. As web sources typically handle concurrent accesses, we first discuss how parallelization can be built on the access minimization framework NC. We then report the results of our experiments.

9.2.1 *Parallelizing NC for Concurrent Accesses.* We now discuss our development of a simple extension of NC to enable concurrent accesses. To reflect the limitation in resources (e.g., network bandwidth or server load), our parallelization assumes a bounded concurrency C, that is, at most, C outstanding accesses can be performed concurrently.

Our parallelization is in fact straightforward by performing accesses *asynchronously*: Without waiting for preceding accesses to complete, NC will continue to issue the next access, as long as the number of outstanding accesses does not exceed the concurrency limit C. The queue $\mathcal{K}_{\mathcal{P}}$ is updated asynchronously as well, whenever an access completes.

While such extension enables overlaping up to C accesses, it also slightly complicates the access selection: Recall that *Select* (Figure 12) picks a sorted access sa_i as the next access, when the last-seen score $\overline{p_i}$ from the preceding sa_i has not reached the suggested depth δ_i, namely, $\overline{p_i} > \delta_i$. Note that the exact $\overline{p_i}$ is not known until all outstanding sorted accesses complete.

However, since synchronizing to get the exact $\overline{p_i}$ defeats the whole purpose of asynchronous accesses, we continue with an estimated, $\overline{p_i}$, instead, by computing its expected decrement D_i. Assuming d_i is the expected decrement of $\overline{p_i}$ after a single sa_i, and that n_i is the number of outstanding sa_i, we estimate D_i as $d_i \cdot n_i$. Initially, we set d_i as $\frac{1}{n}$, assuming that all n objects are uniformly distributed over the score range of [0:1]. Then, d_i can be adapted to a more realistic value based on actual scores retrieved from actual sorted accesses.

Note that in contrast to NC, most other top-k algorithms have inherent parallelism. For instance, consider TA and CA [Fagin et al. 2001], both of which perform sorted accesses to all m predicates in parallel. TA then performs random accesses to completely evaluate the objects seen (up to $m-1$ random accesses per each m object seen), while CA may withhold such random accesses after $h = \frac{cr_i}{cs_i}$ iterations of parallel sorted accesses (to trade expensive random accesses with more economical sorted accesses). Observe that TA, by issuing such sorted and random accesses asynchronously, can overlap up to $m + m(m-1) = m^2$ accesses. Similarly, CA can parallelize the sorted accesses over multiple iterations. Thus, to better accommodate TA and CA, our experiments set the concurrency bound C as m^2, as well.

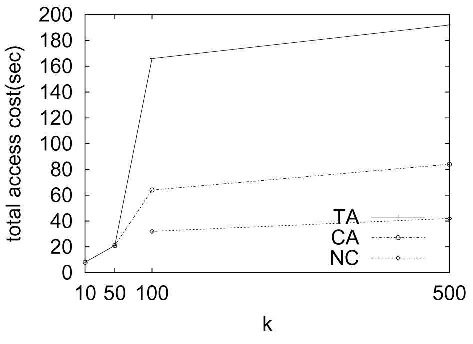
9.2.2 *Results.* This section evaluates NC over actual web sources. In particular, we experiment with the travel agent scenarios Q_1 and Q_2 (Example 1) as our benchmark queries. For extensive evaluation, we evaluate each of the queries Q_1 and Q_2 with both *min* and *avg* as scoring functions. We use the real web sources suggested in Figure 1 to access the restaurants and hotels in Chicago (by issuing an additional Boolean selection "city = Chicago"). As these sources allow sorted access only in small batches (e.g., per page of 25 objects), we regard a batch access as a single sorted access. For simplicity, predicates are evaluated by linearly normalizing the corresponding attribute value into the range of [0 : 1], for example, a *rating* of a two-star hotel in the five-star rating will be evaluated as $\frac{2}{5} = 0.4$.

As metrics, we use both the *total access cost* and actual *elapsed time* to capture two different performance aspects: the resource usage (e.g., of network and web servers) and processing time, respectively. The total access cost is measured by adding up the latency of all accesses (as in Eq. (1)), while the elapsed time simply measures the processing time (including local computation and optimization time). Note that with concurrency, the elapsed time is typically shorter due to by overlapping some high-latency accesses. Using these metrics, we compare NC to TA and CA. However, note that CA is not applicable for Q_2, where the ratio h of random versus sorted access costs is 0.

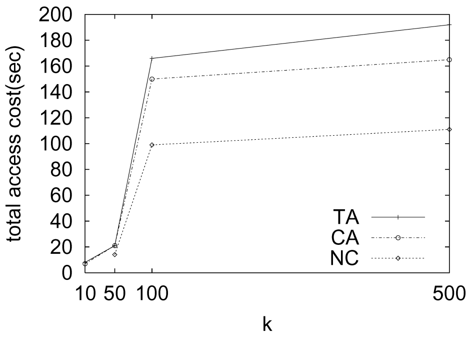
Figures 24(a) and (b) compare the total access cost of TA, CA, and NC for query Q_1 when $\mathcal{F} = min$ and $\mathcal{F} = avg$, respectively. Observe that NC significantly outperforms TA as the retrieval size k (x-axis) increases: For instance, when $k = 500$ in Figure 24(a), the access cost (y-axis) of NC is 42 seconds, which saves 80% and 60% over TA and CA, respectively. In fact, NC outperforms TA by adapting to this scenario with expensive random accesses at runtime: In particular, NC performs deeper sorted accesses than does TA, so as to trade random accesses with less expensive sorted ones.

Similarly, Figures 24(c) and (d) compare the total access cost for query Q_2. Again, the runtime optimization enables NC to outperform TA significantly, such as, when $k = 10$ in Figure 24(c), NC saves up to 66% over the access cost of TA. However, in contrast to Q_1, as random accesses are cheaper than sorted in this scenario, NC generates a totally different algorithm. In particular, to fully exploit free random accesses, NC focuses sorted accesses on a single predicate and evaluates the rest with random accesses (e.g., as in the focused configuration in Example 8), while TA still performs sorted accesses on every predicate with no adaptation. Note that these results are also consistent with our observations in Section 8.

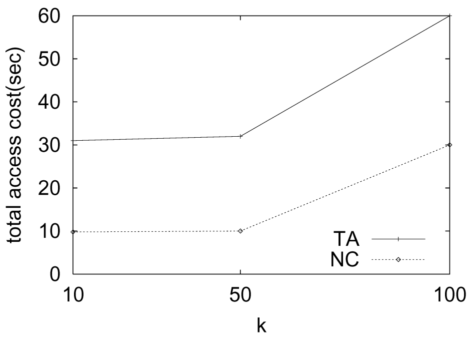
Finally, Figure 25 compares the elapsed time of Q_1 and Q_2 when $\mathcal{F} = min$. Comparing the access cost to the elapsed time of Q_1 (i.e., Figures 24(a) and 25(a)), we observe similar relative behaviors, although in different cost ranges due to the parallel speedup. This consistency suggests that NC can benefit from concurrency to the same extent as do TA and CA, which have inherent parallelism. The same observation holds for Q_2 as well, except when the suggested depth of NC is too shallow. For instance, when $k = 10$, NC can answer the query with a single sorted access (to a page of 25 hotels), and thus cannot benefit



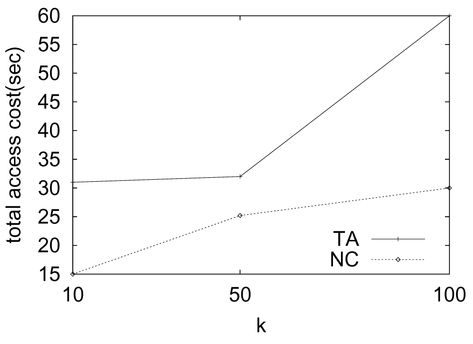
(a) Q_1, $\mathcal{F}$=min



(b) Q_1, $\mathcal{F}$=avg

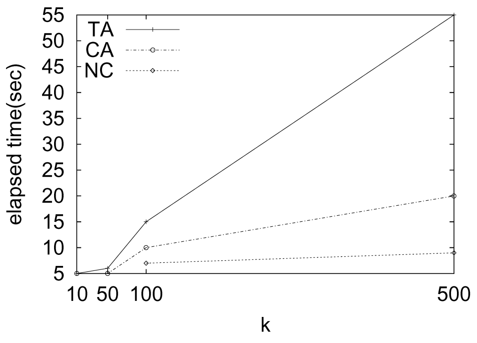


(c) Q_2, $\mathcal{F}$=min

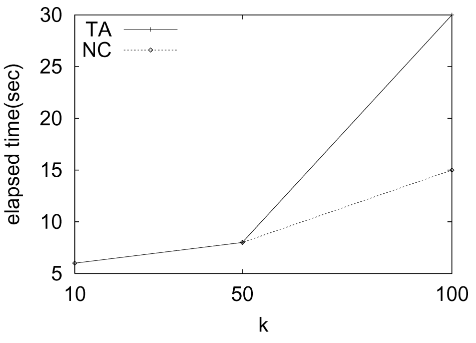


(d) Q_2, $\mathcal{F}$=avg

Fig. 24. Comparison of total access costs.



(a) Q_1, $\mathcal{F}$=min



(b) Q_2, $\mathcal{F}$=min

Fig. 25. Comparison of elapsed time.

from concurrency, while TA, with more accesses, overlaps them and performs comparably.

10. CONCLUSION

This article has developed a cost-based optimization framework for top-k querying in middlewares. We develop framework NC as a comprehensive and focused algorithm space, within which we design runtime search schemes for finding optimal algorithms. Our experimental results are very encouraging: Framework NC significantly outperforms existing algorithms. Further, as a unified top-*k* framework, NC generally works for a wide range of middleware settings.

REFERENCES

BALKE, W., GUENTZER, U., AND KIESSLING, W. 2002. On real-time top-k querying for mobile services. In *Proceedings of the International Conference on Cooperative Information System (CoopIS)*.

BRUNO, N., GRAVANO, L., AND MARIAN, A. 2002. Evaluating top-k queries over web-accessible databases. In *Proceedings of the International Conference on Cooperative Information System (ICDE)*.

CAREY, M. J. AND KOSSMANN, D. 1997. On saying "enough already!" in SQL. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

CAREY, M. J. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

CHANG, K. C.-C. AND HWANG, S. 2002. Minimal probing: Supporting expensive predicates for top-k queries. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

CHAUDHURI, S. AND GRAVANO, L. 1999. Evaluating top-*k* selection queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

DONJERKOVIC, D. AND RAMAKRISHNAN, R. 1999. Probabilistic optimization of top *n* queries. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

FAGIN, R. 1996. Combining fuzzy information from multiple systems. In *Proceedings of the ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

FAGIN, R., LOTE, A., AND NAOR, M. 2001. Optimal aggregation algorithms for middleware. In *Proceedings of the ACM-SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*.

GUENTZER, U., BALKE, W., AND KIESSLING, W. 2000. Optimizing multi-feature queries in image databases. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

GUENTZER, U., BALKE, W., AND KIESSLING, W. 2001. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the International Conference on Information Technology: Coding and Computing (ITCC)*.

HELLERSTEIN, J. M. AND STONEBRAKER, M. 1993. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access path selection in a relational database. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*.

Received October 2005; revised May 2006; accepted October 2006