

# Optimizing Top- $k$ Queries for Middleware Access: A Unified Cost-based Approach <sup>1</sup>

Seung-won Hwang

Department of Computer Science and Engineering

Pohang University of Science and Technology

swhwang@postech.ac.kr

and

Kevin Chen-Chuan Chang

Computer Science Department

University of Illinois at Urbana-Champaign

kcchang@cs.uiuc.edu

---

This paper 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 “generality” to support a wide range of access scenarios and 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 paper, we thus identify and address the barriers to 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, as 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 such operators (or “query plans”). In contrast, top- $k$  queries have no such notion of *logical tasks* as a unit of scheduling. We thus define 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 [Systems]: Query Processing; H.3.3 [Information Search and Retrieval]: Retrieval models; H.3.4 [Performance evaluation]:

General Terms: Algorithms, Performance

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

---

## 1. INTRODUCTION

To enable non-traditional *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  $k$  top answers among a database of  $n$  data objects, each evaluates  $m$  soft *predicates*  $p_1, \dots, p_m$  to scores in  $[0:1]$ , aggregated by some *scoring function*  $\mathcal{F}$  (e.g., *min*). In

---

<sup>1</sup>This paper is based on and significantly extends our preliminary works: “Minimal Probing: Supporting Expensive Predicates for Top- $k$  Queries” in the ACM SIGMOD 2002. See Section 2 for details on the extension.

particular, this paper focuses on top- $k$  queries in a middleware system— *i.e.*, a *middleware* processes queries over subsystems (*e.g.*, an RDBMS integrated with multimedia subsystems or text search engines [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, because of their inherent “data retrieval” nature of retrieving and combining data from multiple sources, top- $k$  queries have emerged to be of particular importance, with many different scenarios studied (as Section 2 will overview).

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 a descending order, one in each access, or 2) *random access*, which returns the score for a given object. As a main motivation of this paper, 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, as 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 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  $\min(p_1 : \text{rating}(r.\text{stars}), p_2 : \text{close}(r.\text{addr}, \text{myaddr} ))$ 
stop after 5
(Query  $Q_1$ )

```

To answer the 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<sup>2</sup>.

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: To contrast, consider another scenario in which our middleware now works with a different source hotel.com (and possibly also with a local RDBMS), to answer query  $Q_2$ :

```

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

```

<sup>2</sup>While these sources may support a multi-dimensional sorted access, *e.g.*, sorted by *price + location*, such access is ordered by a fixed and implicit combining function. As the implicit underlying function is unclear, we focus on leveraging one-dimensional accesses as other middleware top- $k$  algorithms.

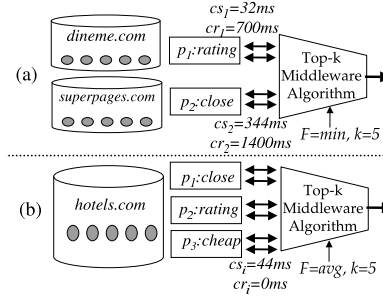


Fig. 1. Scenarios for (a)  $Q_1$  and (b)  $Q_2$ .

Sorted Access	Random 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
expensive: $cs_i = h$	none	FA, TA, Quick-Combine	Stream-Combine
impossible: $cs_i = \infty$	TA <sub>Z</sub> , MPro, Upper		

Fig. 2. Access scenarios and algorithms.

In this setting, since a sorted access (*e.g.*, for *close*), when retrieving a hotel object, also retrieves all its attributes (*e.g.*, stars and price), the subsequent random accesses<sup>3</sup> to the same hotel are thus essentially “piggybacked” with zero costs—*e.g.*, using stars and price, the middleware can locally compute *rating* and *cheap*. 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 either type of access, *e.g.*, *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 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.<sup>4</sup> While these algorithms pioneered in various top- $k$  settings

<sup>3</sup>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].

<sup>4</sup>In a middleware setting, the no wild guess constraint [Fagin et al. 2001] is often assumed to require at least a source must support sorted access. We note that our approach does not make this assumption and thus works with and without constraint, as Section 8 will discuss.

and are appropriate for their specific scenarios, we are clearly lacking “generality”, such that an algorithm is typically not generally applicable to real-life scenarios where sources differ in cost characteristics (*e.g.*, random access is cheap in some source while impossible in another) that 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 some limited attempts with rather ad-hoc heuristic-based adaptation (as Section 2 discusses). In contrast, as we will argue below, a systematic adaptation to runtime specifics can make a significant difference in performance.

**Our Approach:** To fulfill this critical lacking of generality and adaptivity, this paper aims at a *general* framework over such various scenarios that is *adaptive* to the given scenario 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  $cr_i$ . That is, by varying  $cs_i$  and  $cr_i$  for different predicates and accesses, we will capture various 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]). That is, given some algorithm  $\mathcal{M}$ , let  $S_i$  and  $R_i$  be the number of sorted and random accesses respectively for predicate  $p_i$ , the total cost is

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

In particular, as the main thesis of this paper, we propose a systematic and unified “cost-based optimization” framework, which will adapt to various cost scenarios— That is, 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 corresponding cost of Eq. 1. While such 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, despite active research (where each algorithm focuses on specific scenarios, as Figure 2 summarizes).

Our cost-based optimization framework complements existing algorithms with its *generality* and *adaptivity*: First, in terms of *principle*, our framework aims at providing a unified framework to be *general* over any arbitrary scenario. We prove the generality, by showing any possible algorithm has a counterpart algorithm generated by our framework, doing the same job with no more access (Theorem 4). Our framework, being general, generates the behaviors of the existing algorithm in their target scenarios, as we discuss further in Section 8. Beyond existing algorithms, our framework will also handle unstudied scenarios. For instance, note that the scenario of Figure 1(b), in which sorted accesses (cost 44ms per access) are more expensive than random accesses (zero cost), has not been specifically studied (the “none” cell in Figure 2).

Second, in terms of *performance*, our framework is *adaptive* and thus enables a potentially significant cost difference from existing algorithms, by a systematic adaptation to runtime specifics. As an illustration, Figure 3 summarizes our evaluation results (which will be reported in details in Section 9, Figure 18b), comparing the cost of 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 existing algorithm TA by orders of magnitude. Such cost difference results

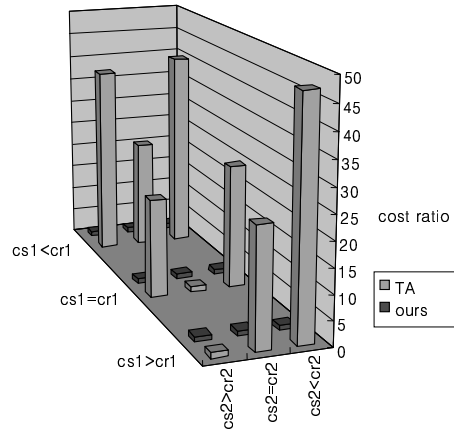


Fig. 3. Adaptivity over costs.

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 for its realization. As the first barrier, we need to define a “comprehensive” space, which we denote as  $\Omega$ , encompassing all possibly optimal algorithms to search over. Second, as a conflicting goal, such  $\Omega$  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  $\Omega$  is, by definition, all query plans of such operators that are algebraically equivalent (*e.g.*, by their commutativity and associativity). However, what is a “query plan”<sup>5</sup> for a top- $k$  query? It is not obvious how a top- $k$  query, as an arbitrary scoring function, *e.g.*,  $\mathcal{F} = \min(p_1, p_2)$ , 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  $\Omega$  (Section 4). Second, to define a *focused* space, we develop the insight that query processing can focus, without compromising any generality, on only *necessary choices* (Section 5), based on which we define a comprehensive and focused algorithm space (Section 6). With this comprehensive space  $\Omega$  defined, we further reduce the search space to reduce the overhead of finding the optimal algorithm in the space. In particular, we adopt systematic space reduction schemes, such as focusing on algorithms performing sorted access first before any random access and those with each object following the same random access schedule. We will formally and empirically

<sup>5</sup>We view our top- $k$  retrieval as a “query” with a “query plan”—In a different view, one may think ranking as part of a relational query and thus corresponds 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 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.

argue in Section 7 that such schemes effectively reduces the space without significantly compromising the comprehensiveness. With the reduced space  $\Omega$ , cost-based optimization is to identify the optimal algorithm  $\mathcal{M}_{\text{opt}}$  in  $\Omega$ , with respect to the cost model  $\mathcal{C}$ , *i.e.*,

$$\mathcal{M}_{\text{opt}} = \operatorname{argmin}_{\mathcal{M} \in \Omega} \mathcal{C}(\mathcal{M}). \quad (2)$$

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 was also studied in [Bruno et al. 2002]). However, in contrast to the necessary-probe principle focusing only on scheduling random access (as our preliminary work focuses on specific scenarios where random access cost dominates), we generalize the techniques to arbitrary accesses (*i.e.*, sorted access and random access) assumed by top- $k$  works and thus achieve general applicability to any top- $k$  scenarios. 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 midleware scenarios. The results are indeed encouraging; our framework not only unifies but also outperforms the existing algorithms specifically designed for their 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 their cost-based optimization.
- 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 (*i.e.*, *instance optimality*), *e.g.*, TA (for uniform cost scenarios), NRA (when random access is impossible), and TA<sub>Z</sub> (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 then a random access phase, CA alternates  $h$  sorted access phases then a random access phase, to favor sorted accesses (which is  $h$  times cheaper) over random accesses, in evaluating the

undetermined predicate scores. Their heuristics-based optimization has limited applicability: Algorithm CA adapts to a runtime parameter  $h$ —*i.e.*, the cost ratio of random versus sorted accesses, which is assumed to be the same for *all* predicates. In practice, such an assumption is unlikely to hold: Across autonomous sources for different predicates, this ratio will be varying (for Figure 1a) or simply zero (for 1b). The rest three algorithms use the partial derivative of scoring functions as an indicator to optimize, which may not be applicable to all functions (*e.g.*,  $\mathcal{F}=\min$  in  $Q_1$ ).

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 where 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 with 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 access (as random access cost dominates in its expensive random access scenarios) into a complete framework scheduling arbitrary accesses assumed by top- $k$  works, *i.e.*, random access and sorted access, and thus achieve general applicability to *any* top- $k$  scenarios. 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 further specifically assumes *weighted average* scoring functions. In contrast, we propose a more general adaptation framework, which enables to generalize Upper to not only arbitrary (random and sorted) accesses but also any monotonic 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 is lacking 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 similarity; *e.g.*, they keep retrieved objects in some order and terminate at some 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.

To fulfill these critical lackings, we 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 a 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, *i.e.*, 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, *e.g.*, according to [Fagin et al. 2001],  $c$  can be up to  $m(m-1) + m \frac{cr_i}{cs_i}$  for TA for  $m$  predicates with a unit sorted and random access cost  $cs_i$  and  $cr_i$  respectively. In other words, as varying cost ratios and query sizes are common in practice, the distance  $c$  is in many cases *not* a constant to be ignored in optimization: First, by ignoring *access cost ratios* in optimization, more specifically  $\forall i, j : \frac{cs_i}{cs_j} = \frac{cr_i}{cr_j} = 1$ , the optimality is meaningful only to limited uniform-cost scenarios, *e.g.*, the cells in the current matrix. However, actual application scenarios are unlikely to be uniform, especially over autonomous middleware sources (*e.g.*,  $\frac{cr_2}{cr_1} = 20$  in Figure 1a). 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 the query size (*i.e.*,  $|Q| = m$ ) and the database size (*i.e.*,  $|D| = n$ ). However, instance optimality assumes  $m$  as a constant, which reduces the optimality to only with respect to the number of objects evaluated; *regardless* of the predicates each object evaluates. (To contrast, Boolean query optimizers, *e.g.*, [Hellerstein and Stonebraker 1993], mainly strive to minimize such predicate access costs.) By systematically adapting to all these runtime cost parameters, our framework aims at generally optimizing for virtually all access scenarios (as briefly demonstrated in Section 1).

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. Towards the goal, our first task is to define the algorithm space  $\Omega$  (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 to guide narrowing 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, based on 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. Towards the goal, 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 top-1 restaurant, *i.e.*, thus setting retrieval size  $k = 1$ . For our illustration, let’s assume Dataset 1 (Figure 5) as 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 input, top- $k$  algorithms will return an answer  $\mathcal{K} = \{u_3:0.7\}$ , *i.e.*,  $u_3$  is the top-ranked object with score  $\mathcal{F}[u_3]=0.7$ .

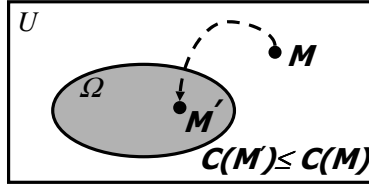


Fig. 4. Cost comprehensiveness.

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

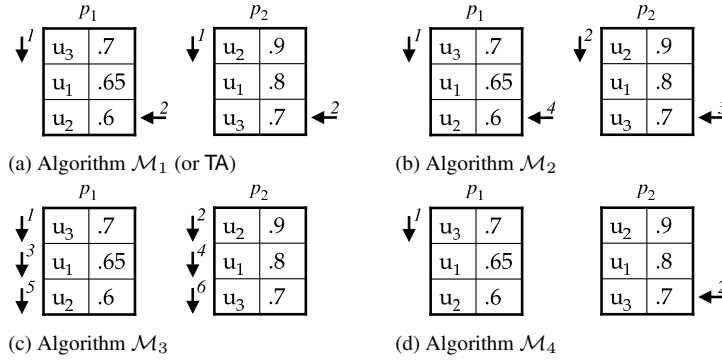


Fig. 6. Example algorithms.

Recall that, as Section 1 mentioned, this paper 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  $sa_i$ ; 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 aspects:

- **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, e.g.,  $u_1$  and  $u_2$ , with this *last-seen* score—e.g.,  $p_1[u_1] \leq .7$ . In contrast, random access  $ra_i(u_j)$  has no effect on other objects 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  $u_2$  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.

Built upon these access methods, to answer a query, an algorithm performs accesses to gather necessary scores. To illustrate how it works, Figure 6 illustrates example algorithms, using a “sorted” list for each predicate—refer to Figure 6(a) for  $p_1$  and  $p_2$ —in which objects are ordered by the corresponding predicate scores. On the sorted list, we will use  $\downarrow$  and  $\leftarrow$  to represent sorted and random accesses performed respectively, and annotate by their “time” of access, e.g.,  $\leftarrow_2$  in Figure 6(a) on  $p_2$  represents 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  $Q_1$  in Figure 6(a): 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, i.e.,  $u_3$  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  $u_3$ , i.e.,  $\mathcal{F}[u_3] = \min(0.7, 0.7) = 0.7$ , is no less than that of the “unseen” object (i.e.,  $u_1$ )—As  $p_1[u_1]$  is bounded by the “side-effect”, i.e.,  $p_1[u_1] \leq .7$ ,  $\mathcal{F}[u_1] = \min(p_1[u_1], p_2[u_1])$  cannot be higher than  $\mathcal{F}[u_3] = .7$ . ■

As Example 2 shows one possible algorithm (i.e.,  $\mathcal{M}_1$  as TA), depending on which accesses are performed at a time, there can be many different algorithms answering the same query. To illustrate, consider 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 different orders, incur different costs. For instance,  $\mathcal{M}_4$  can answer the same query, performing only a part of 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. Toward the goal, we must guarantee the algorithm space includes the optimal algorithm: For this guarantee, we first formalize the notion of *cost comprehensiveness* (as Figure 4 illustrates): While a space  $\Omega$  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  $\Omega$  that answers the same query with *no more* cost. Such  $\Omega$  is thus comprehensive enough for optimization; no algorithms outside of the space can be better than all algorithms in the space, since its counterpart  $\mathcal{M}'$  is at least as good as  $\mathcal{M}$ . We formalize this notion of “comprehensiveness” below, which provides a foundation for finding such a space.

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

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

In summary, while  $\mathcal{U}$  is our universe in principle, if some space  $\Omega$  satisfies cost comprehensiveness (Definition 1), it is sufficient to search in  $\Omega$  instead of  $\mathcal{U}$ , i.e.,  $\operatorname{argmin}_{\mathcal{M} \in \mathcal{U}} \mathcal{C}(\mathcal{M}) = \operatorname{argmin}_{\mathcal{M} \in \Omega} \mathcal{C}(\mathcal{M})$ : As a key contribution, we identify such comprehensive space  $\Omega$ , for our objective of minimizing total access costs (Eq. 1): As a key insight, observe Algorithm  $\mathcal{M}_2$  (Figure 6b) performing the exact same set of accesses as  $\mathcal{M}_1$  (Figure 6a) 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.

```

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.

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<sup>6</sup>. More formally, we model such 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  $\mathcal{M}_{\text{opt}}$ , with respect to the cost scenario, *i.e.*,

$$\mathcal{M}_{\text{opt}} = \underset{\mathcal{M} \in \mathcal{G}(\text{SEQ})}{\text{argmin}} \mathcal{C}(\mathcal{M}). \quad (3)$$

While we have successfully fulfilled our first objective of achieving the comprehensiveness (as Section 3 motivated), we are now facing the second challenge of defining a *focused* space: Unfortunately, though comprehensive,  $\mathcal{G}(\text{SEQ})$  is extremely large, as algorithms vary depending on access  $A$  selected at each iteration, which can be *any* among a huge set of supported accesses— To illustrate, for  $n = 100000$  objects with  $m = 5$  predicates, at each iteration, there can be as many as  $m + m \cdot n = 500005$  different supported accesses to choose from, as our framework do not make “no wild guess” assumption. (Note, if 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. Our next goal is thus to refine the space to be as focused as possible, without compromising the comprehensiveness.

Towards the goal, as Section 3 motivated, it is critical to first decompose a top- $k$  query into “logical tasks” as building blocks, as Section 5 will develop, based on which we define a comprehensive and focused search space in Section 6.

<sup>6</sup>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, *e.g.*, as Web sources typically do. Section 9 will show that such parallelization can successfully “build upon” our access minimization framework.

## 5. DEFINING A FOCUSED SPACE

We now ask a fundamental question: While access methods are “physical means” for gathering object scores, what are “logical tasks” that a top- $k$  query must fulfill? Such logical tasks are determined only by the objective of a query, and it is independent of any physical implementation. That is, any algorithms (with whatever access methods) must successfully carry out these tasks. Such a logical view is thus a critical foundation for systematic algorithm design.

### 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  $\mathcal{D} = \{u_1, \dots, u_n\}$ , any algorithm  $\mathcal{M}$  must gather certain score information for *each* object  $u_j$ , to determine the top- $k$ . We can thus “compose” the work of  $\mathcal{M}$  by a set of required *scoring tasks*,  $\{w_1, \dots, w_n\}$ . To define such tasks, let  $\mathcal{K} = \{v_1, \dots, v_k\}$  be the top- $k$  answers (where each  $v_i$  represents some  $u_j$  from  $\mathcal{D}$ ). In this paper, we assume 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  $k^{\text{th}}$  answer).

**Definition 2 (Scoring Tasks):** Consider a top- $k$  query  $Q = (\mathcal{F}, k)$ , with top- $k$  answers  $\mathcal{K} = \{v_1, \dots, v_k\}$ . 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; or
2. otherwise:  $w_j$  must indicate (by some partial scores) the *maximal-possible*  $\mathcal{F}[u_j]$  score, tight enough to support that  $\mathcal{F}[u_j] < \mathcal{F}[v_k]$ .<sup>7</sup>

As a remark, note that these tasks are specified with *given*  $\mathcal{K}$  (the top- $k$  answers) and  $\mathcal{F}[v_k]$  (the  $k^{\text{th}}$  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—  $p_1[u_3]$  and  $p_2[u_3]$ — for computing  $\mathcal{F}[u_3]$ . Note  $w_3$  can do so in various ways, *e.g.*, 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  $p_2[u_3] = .7$ ).

To contrast, task  $w_2$  for  $u_2$  (and similarly  $w_1$  for  $u_1$ ) needs only 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

<sup>7</sup>Note that, 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, *e.g.*, by unique object IDs. Such enforcement of certain tie breaker enables optimization to compare candidate algorithms that return the exact same set of top- $k$  results.

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.

$sa_1$  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, *i.e.*,  $p_1[u_2] \leq .65$ . As  $\mathcal{F}[u_2] = \min(p_1[u_2], p_2[u_2])$ ,  $\mathcal{F}[u_2]$  cannot be higher than  $p_1[u_2]$ , *i.e.*,  $\mathcal{F}[u_2] \leq .65 < .7$ . ■

We stress that these scoring tasks are both necessary and atomic: First, each  $w_j$  is *necessary*: If any  $w_j$  is not satisfied,  $\mathcal{M}$  cannot properly handle object  $u_{j-1}$  if  $u_j$  is a top- $k$  answer,  $\mathcal{M}$  cannot return its final score; 2) otherwise, without proving  $\mathcal{F}[u_j] < \mathcal{F}[v_k]$ ,  $\mathcal{M}$  cannot safely exclude  $u_j$  from the top- $k$ . Second, 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, 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  $\{w_1, \dots, w_n\}$ — Each task  $w_j$ , for object  $u_j$ , gathers the required per-object information. Only when (and clearly when) all the tasks are fulfilled, the query can 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 it is 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 unsatisfied tasks and complete such tasks first. However, *during* query processing, it is challenging to judge whether a task is satisfied, since  $\mathcal{K} = \{v_1, \dots, v_k\}$ , 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 any unsatisfied task must eventually be fulfilled.) Our insight is, 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, 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 “side-effect” (Section 4), the “unseen” objects (*i.e.*,  $u_2$ ) will be bounded by the last-seen score, *i.e.*,  $p_1[u_2] \leq .65$ .

- The one 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 together, Figure 8 summarizes the current “score state.” For  $u_1$ : The above 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  $\mathcal{F}[u_3] \leq .7$

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):  $w_3$  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 hints, 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 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 *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, i.e.,  $\mathcal{F}(x_1, \dots, x_m) \geq \mathcal{F}(y_1, \dots, y_m)$  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  $\overline{p}_i$ . (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,  $\overline{\mathcal{F}}_{\mathcal{P}}(p_1, \dots, p_m)[u_j] =$

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

Further, we focus on the current top- $k$  objects by their potentials. Let  $\mathcal{K}_{\mathcal{P}} = \{v_1, \dots, v_k\}$  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 if any current top objects are “incomplete”:

*First*, if  $\mathcal{K}_{\mathcal{P}}$  contains any *incomplete* object– one that has not been fully evaluated (i.e., with only partial scores): As Example 4 argued for  $u_3$  (an incomplete top-1), such  $v_j$  needs further accesses either way, 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, to be safely excluded from top- $k$ . Thus, task  $w_j$  for such incomplete  $v_j$  is clearly unsatisfied.

*Second*, if all objects  $v_1, \dots, v_k$  in  $\mathcal{K}_{\mathcal{P}}$  are complete: 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, i.e.,  $\mathcal{F}[v_j] = \overline{\mathcal{F}}_{\mathcal{P}}[v_j]$ . 2) Every object  $u_i \notin \mathcal{K}_{\mathcal{P}}$ , with the current ranking, has its maximal-possible score lower than the above exact scores, i.e.,  $\forall v_j \in \mathcal{K} : \overline{\mathcal{F}}_{\mathcal{P}}[u_i] \leq \mathcal{F}[v_j]$ . It follows that those  $v_j$  are the top- $k$  answers, fully evaluated. With these two observations, Definition 2 will declare 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  $\mathcal{D} = \{u_1, \dots, u_n\}$ . With respect to a set  $\mathcal{P}$  of performed accesses, let  $\mathcal{K}_{\mathcal{P}} = \{v_1, \dots, v_k\}$  be the current top- $k$  objects ranked by  $\overline{\mathcal{F}}_{\mathcal{P}}[\cdot]$ .

1.  $\forall v_j \in \mathcal{K}_{\mathcal{P}}$  s.t.  $v_j$  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 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  $v_j \notin \mathcal{K}$ : Suppose its scoring task  $w_j$  is satisfied: It 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,  $\mathcal{F}[v_j] = \overline{\mathcal{F}}_{\mathcal{P}}[v_j] > \overline{\mathcal{F}}_{\mathcal{P}}[u] \geq \mathcal{F}[u], \forall u \notin \mathcal{K}_{\mathcal{P}}$ , and thus  $\mathcal{K}_{\mathcal{P}} = \mathcal{K}$ . 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  $\mathcal{F}[v_j] > \mathcal{F}[u_i], \forall v_j \in \mathcal{K}$  (as shown above),  $w_i$  is thus satisfied (by Definition 2, Case 2).

In summary, Theorem 1 states that, at any point, top- $k$  objects need to be further evaluated, *i.e.*, at least one or more sorted or random access on its unevaluated predicates on the current top- $k$  objects at any point. However, note that, it 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 it 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 consists of those query plans that are equivalent algebraically. The algebraic framework induces a space of query

---

**Framework NC**( $Q, \mathcal{D}$ ): Necessary Choices  
**Input:** query  $Q = (\mathcal{F}(p_1, \dots, p_m), k)$ ,  
 database  $\mathcal{D} = \{u_1, \dots, u_n\}$   
**Output:**  $\mathcal{K}$ , top- $k$  objects from  $\mathcal{D}$  w.r.t. to  $\mathcal{F}$

- (1)  $\mathcal{P} \leftarrow \phi$ ; // *accesses-so-far*
- (2)  $\mathcal{K}_{\mathcal{P}} \leftarrow \{v_1, \dots, v_k \mid \text{top-}k \text{ from } \mathcal{D} \text{ ranked by } \overline{\mathcal{F}}_{\mathcal{P}}[\cdot]\}$ ;
- (3) **while** (true)
- (4)  $U \leftarrow \{v_j \mid v_j \in \mathcal{K}_{\mathcal{P}}; v_j \text{ is incomplete}\}$
- (5) **if** ( $U = \{\}$ ) **break**;
- (6)  $v_j \leftarrow$  any object in  $U$ ; // *e.g., the highest-ranked*
- (7)  $N_j \leftarrow \{sa_i, ra_i(v_j) \mid p_i[v_j] \text{ is undetermined by } \mathcal{P}\}$ ;
- (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.

plans, each as a different schedule. Optimization is to find a good schedule of operations, conforming to the framework.

Built 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 accesses can be scheduled at iteration, is essentially a template SEQ rendering a space of all sequential algorithms (Section 4). In contrast, in this section, to render a more focused space, we develop a framework that hinges on the insight that query processing can focus on *only* unsatisfied tasks, without compromising optimality. That is, 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 on task decomposition (Section 5)– that top- $k$  query processing is *equivalent* to fulfilling a set of (necessary and atomic) tasks  $\{w_1, \dots, w_n\}$ . 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 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  $w_j$ ’s unevaluated predicates. (As Theorem 1 states, for such unsatisfied  $w_j$ , its object  $u_j$  must be still 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 on  $u_3$ , especially 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  $p_2[u_3]$ : As  $\mathcal{P}$  (Example 4) has already one  $sa_2$ , the next  $sa_2$  will return  $u_1$  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 turning  $u_3$  into completely evaluated, with score state ( $p_1=.7, p_2=.7 \rightarrow \mathcal{F}=.7$ ). In fact,  $w_3$  is now satisfied.

Putting 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  $N_j = \{sa_i, ra_i(u_j) \mid p_i[u_j]$  is undetermined by  $\mathcal{P}\}$ , without performing at least one of which  $w_j$  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 predicate  $p_i$ .

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 predicate by  $ra_i(v_j)$ , or by lowering the upper bounds by  $sa_i$ , for all unevaluated predicate  $p_i$ .

Observe Figure 9 describing 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, which variable  $U$  collects. 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 any such  $v_j$  (say, the highest-ranked one) without compromising optimality, and constructs the necessary choices  $N_j$  (by Theorem 2) as **alternatives** for selecting further access. As each unsatisfied task remain unsatisfied until at least one among its necessary choices is performed, arbitrarily picking one of such unsatisfied tasks does not compromise the 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  $\mathcal{K}_{\mathcal{P}}$ , the current top- $k$  objects. Thus, a “search mechanism” for finding unsatisfied tasks should return top- $k$  objects when requested – e.g., a *priority queue* that orders objects by maximal-possible scores as priorities. Note that, initially, all objects have the same maximal-possible score (i.e., 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 (Section 5). While any tie-breaker can be used, for the sake of presentation, our examples will assume some OID as a tie-breaker, e.g., when  $u_i$  and  $u_j$  tie and  $i > j$ , then we consider  $u_i$  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, e.g., the one with the highest

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.

partial score, and still ensures the 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, and is thus “equally” necessary. In other words, any  $N_j$  is “complete” (Theorem 3), which guarantees the comprehensiveness of the algorithm space NC renders (Theorem 4). Example 6 illustrates how NC works.

**Example 6 (Framework NC):** Figure 10 shows the execution of algorithm  $\mathcal{M}_4$  (Figure 6d) that NC can generate: Initially, at Step 1 (Figure 10), as all the maximal-possible scores tie as 1.0,  $\mathcal{K}_{\mathcal{P}}$  is set to  $\{u_3\}$  (by the highest OID, our tie-breaker), which induces  $\text{alternatives} = N_3$ . According to NC,  $\mathcal{M}_4$  then **Select** an access,  $sa_1$  in this case, among the **alternatives**, which returns  $p_1[u_3] = .7$  (see Figure 5) and lowers  $\overline{p}_1$  to .7.

At Step 2, as all the maximal-possible scores tie as .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  $p_2$ .  $\mathcal{M}_4$  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 the space framework NC renders is not only far more focused than the space SEQ renders but also sufficiently comprehensive. First, we note that NC, by focusing on only necessary choices, *i.e.*,  $|\text{alternatives}| = 2 \cdot m$ , it is clearly more focused than SEQ selecting an access from any supported accesses, *i.e.*,  $|\text{alternatives}| = m + m \cdot n$ . 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 below.

**Theorem 3 ( $N_j$  Completeness):** A set of necessary choices  $N_j$  for every  $j$ , NC identifies for an unsatisfied task  $w_j$  is *complete* with respect to ‘accesses-so-far’  $\mathcal{P}$ , such that any algorithm having done  $\mathcal{P}$  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 and thus the query cannot be 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  $\mathcal{G}(\text{NC})$ , is comprehensive for optimization (*i.e.*, cost comprehensiveness in Definition 1), as Theorem 4 below 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  $\mathcal{G}(\text{NC})$  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, *i.e.*,  $\mathcal{P}(\mathcal{M}_1) = \mathcal{P}_1$ . Since  $\mathcal{M}_2$  follows the iterative framework (Figure 10), let  $\mathcal{P}_2^j$  be the accesses of  $\mathcal{M}_2$  *before* the  $j^{\text{th}}$  iteration; initially,  $\mathcal{P}_2^1 = \phi$ . Similarly, let **alternatives<sup>j</sup>** be alternatives of  $\mathcal{M}_2$  at  $j^{\text{th}}$  iteration.

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

- $L_1$ : **alternatives<sup>j</sup>**  $\cap (\mathcal{P}_1 - \mathcal{P}_2^j) \neq \phi, \forall j$ .
- $L_2$ :  $\mathcal{P}_2^j \subseteq \mathcal{P}_1, \forall j$ .

Note that, by  $L_1$ , we show NC can construct algorithm  $\mathcal{M}_2$  to follow the access of  $\mathcal{M}_1$  at each iteration. More specifically, for every iteration  $j$ ,  $\mathcal{M}_2$  selects one access from **alternatives<sup>j</sup>** that is performed by  $\mathcal{M}_1$  but not yet by  $\mathcal{M}_2$ , *i.e.*,  $\mathcal{P}_1 - \mathcal{P}_2^j$ , which is possible if  $\mathcal{P}_2^j \subseteq \mathcal{P}_1, \forall j$ . We then show, by  $L_2$ , such algorithm  $\mathcal{M}_2$  incurs no more access than  $\mathcal{M}_1$ , when  $\mathcal{M}_2$  halts at some iteration  $j$  (denoted as  $\mathcal{M}_2^j$ ):  $\mathcal{P}_2^j \subseteq \mathcal{P}_1$ . Note, 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 accesses performed: If  $\mathcal{M}_1$  performs more times of *every* kind of access than  $\mathcal{M}_2^j$ , then  $\mathcal{M}_1$  will have an overall higher cost, *i.e.*,  $\mathcal{P}(\mathcal{M}_2^j) \subseteq \mathcal{P}(\mathcal{M}_1) \implies \mathcal{C}(\mathcal{M}_2^j) \leq \mathcal{C}(\mathcal{M}_1)$ . 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, 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, **alternatives<sup>j</sup>** is “complete”—that any algorithm (like  $\mathcal{M}_1$ ) that has performed  $\mathcal{P}_2^j$  must have performed ‘*in addition*’ some access  $A$  among **alternatives<sup>j</sup>**. Thus, as  $\mathcal{M}_1$  has performed  $\mathcal{P}_2^1$  (trivially, since  $\mathcal{P}_2^1 = \phi$ ), it must have performed access  $A \in$  **alternatives<sup>1</sup>** in addition. That is,  $A$  is in both **alternatives<sup>1</sup>** and  $\mathcal{P}_1 - \mathcal{P}_2^1$ , and thus  $L_2$  holds.  $L_2$  is trivial, since initially  $\mathcal{P}_2^1 = \phi$ .
- $j = k$ : As the induction hypothesis, assume for  $j = k$ , 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 that  $\mathcal{M}_2$  select one access from **alternatives<sup>k</sup>** to continue: We will let  $\mathcal{M}_2$  choose an access  $A^k$  that is also in  $\mathcal{P}_1 - \mathcal{P}_2^k$ —Such  $A^k$  must exist by  $L_2$ , *i.e.*,  $A^k \in$  **alternatives<sup>k</sup>**  $\cap (\mathcal{P}_1 - \mathcal{P}_2^k)$ .

- $j = k + 1$ : First,  $L_1$  holds: By  $L_1$  (just proved above) that  $\mathcal{P}_2^{k+1} \subseteq \mathcal{P}_1$ ,  $\mathcal{M}_1$  has performed  $\mathcal{P}_2^{k+1}$ . By the completeness of **alternatives<sup>k+1</sup>**,  $\mathcal{M}_1$  must have performed, ‘*in addition*’ to  $\mathcal{P}_2^{k+1}$ , some access  $A \in$  **alternatives<sup>k+1</sup>**. That is,  $A$  is in both **alternatives<sup>k+1</sup>** 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 that is comprehensive and focused. Our goal, in principle, is thus to “instantiate” an optimal algorithm  $\mathcal{M}_{\text{opt}}$  in  $\mathcal{G}(\text{NC})$ , which depends on query and data-specific

factors. Section 7 will discuss optimization techniques for finding  $\mathcal{M}_{\text{opt}}$  such that, further refining Eq. 3:

$$\mathcal{M}_{\text{opt}} = \operatorname{argmin}_{\mathcal{M} \in \mathcal{G}(\text{NC})} \mathcal{C}(\mathcal{M}). \quad (5)$$

## 7. SEARCH: DYNAMIC OPTIMIZATION

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

1. *Space reduction*: While already much focused,  $\mathcal{G}(\text{NC})$  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 the 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, *e.g.*, dynamic programming.

### 7.1 Space Reduction

While already much focused,  $\mathcal{G}(\text{NC})$  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 varying a few configuration parameters. In particular, we use the following techniques for optimization:

First, we choose to focus on *SR* algorithms (for sorted-then-random), which perform all  $sa_i$  on predicate  $p_i$ , if exists, before any  $ra_i(\cdot)$ . We argue 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  $\mathcal{M}_1 \in \mathcal{G}(\text{NC})$ , there exists its *SR*-counterpart  $\mathcal{M}_2$  with no more cost, *i.e.*,  $\mathcal{C}(\mathcal{M}_2) \leq \mathcal{C}(\mathcal{M}_1)$ . ■

**PROOF.** We prove by constructing *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  $p_i$ , *i.e.*,  $\mathcal{P}(\mathcal{M}_1) = \sum_i \mathcal{P}_1^i$ . That is,  $\mathcal{P}_1^i$  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 sorted-then-random manner, *i.e.*,  $\mathcal{P}_2^i$  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, *i.e.*,  $\mathcal{C}(\mathcal{M}_2) \leq \mathcal{C}(\mathcal{M}_1)$ .

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

---

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

Second, we assume that random access on every object follows the same “global” order  $\mathcal{H}$ . That is, when multiple random accesses exist in **alternatives**, we follow some particular order  $\mathcal{H}$  (given by the optimizer; See Section 7.3) 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 as the next unevaluated predicate of  $u_1$  is  $p_i$  according to  $\mathcal{H}$ , which we denote as  $\text{next}(u_1, \mathcal{H}) = p_i$ . According to our preliminary study [Chang and Hwang 2002], global scheduling is as effective as local scheduling (and thus hardly compromising comprehensiveness), while significantly efficient reducing the per-object overhead, as expensive predicates evaluated at runtime makes it infeasible to obtain enough knowledge to customize predicate scheduling for each object.

By focusing on the above two techniques, we propose Framework **NC** with SR/G (SR-subset and Global scheduling) techniques, trading high efficiency over 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 two parameters  $\Delta = (\delta_1, \dots, \delta_m)$  and  $\mathcal{H} = (p_1, \dots, p_m)$ , which will be determined by the optimizer (Section 7.3). In essence, *Select* chooses sorted access whenever there exists  $sa_i$  which hasn’t reached the suggested depth  $\delta_i$ , i.e.,  $\overline{p}_i > \delta_i$ .<sup>8</sup> 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  $\Delta_1 = (0.8, 0.8)$  and  $\mathcal{H} = (p_1, p_2)$ .

At step 1, among necessary choices **alternatives** =  $N_3$ , *Select* focuses on  $sa_1$  and  $sa_2$ , as the suggested sorted access depths haven’t been reached yet i.e.,  $\overline{p}_1 > \delta_1 = 0.8$  and  $\overline{p}_2 > \delta_2 = 0.8$ . (We arbitrary pick one, e.g.,  $sa_1$ .) Similarly, at step 2 and 3, *Select* chooses  $sa_2$ , until it lowers  $\overline{p}_2$  below the suggested depth  $\delta_2$  after step 3. Then, at step 4,

<sup>8</sup>While rare in practice, there can be an extreme case where the suggested depth is too shallow 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 the suggested depths.

we perform  $ra_2(u_3)$ , which completes the evaluation on  $u_3$ . **NC** can thus 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, the SR/G techniques enable to enumerate algorithms by parameters  $\Delta$  and  $\mathcal{H}$ , *i.e.*, every SR algorithm can be identified by  $(\Delta, \mathcal{H})$  pair. Consequently, our optimization problem can now be restated as identifying the minimal-cost algorithm  $(\Delta_{opt}, \mathcal{H}_{opt})$  such that  $(\Delta_{opt}, \mathcal{H}_{opt}) = \operatorname{argmin}_{\Delta, \mathcal{H}} \mathcal{C}((\Delta, \mathcal{H}))$ .

## 7.2 Cost Estimation

As a prerequisite to identify the cost-optimal algorithm  $(\Delta_{opt}, \mathcal{H}_{opt})$ , we need to develop how to estimate the cost of a SR/G top- $k$  algorithm  $(\Delta, \mathcal{H})$ . To motivate, recall the cost estimation for Boolean query plans. The cost of Boolean query is essentially 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 the Boolean selectivity, *i.e.*, the ratio of the number of data objects that evaluate the given predicate to be true, obtained from some “statistical samples”, *e.g.*, histograms. For instance, in a simple conjunctive query,  $N_i$  is simply the product of the predicate selectivities of those predicates 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”, which is the selectivity of a set of evaluated predicates as the ratio of the number of objects whose aggregate scores that can still make to the top- $k$  answers.

Further, let  $\theta$  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 eventually only the final answers will surface to and remain on the top).

We thus define the *aggregate selectivity*  $\mathcal{S}_{\mathcal{F}}^{\theta}(\mathcal{P}, \Delta)$  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  $\Delta$ . (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 the random accesses after the preceding sorted access phase. (We will later discuss how to estimate the cost of sorted accesses up to  $\Delta$ .) That is, when  $\mathcal{H} = (p_1, \dots, p_m)$  and  $\mathcal{S}_{\mathcal{F}}^{\theta}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta)$  is the ratio of the number of objects scoring over  $\theta$  after the sorted access phase up to depth  $\Delta$  followed by the random accesses up to subschedule  $\mathcal{H}_i = (p_1, \dots, p_i)$  up to  $i^{\text{th}}$  predicate in the schedule, the number of random accesses on  $p_i$  is  $R_i = N \cdot \mathcal{S}_{\mathcal{F}}^{\theta}(\mathcal{P}(\Delta, \mathcal{H}_{i-1}), \Delta)$ , and the cost of random access phase is thus:

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

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 no less than  $\delta_i$  as  $n(p_i, \delta_i)$ , the overall cost  $\mathcal{C}((\Delta, \mathcal{H}))$  can be formulated as follows:

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

Such cost can be estimated by “simulating” the aggregate effect on the statistical samples, mimicking the actual execution with the retrieval size  $k'$  proportional to the sample size  $s$ , *i.e.*,  $k' = \lceil k \cdot \frac{s}{n} \rceil$ . We first mimic the sorted access phase by performing sorted accesses on these sample (we discuss how to get such samples later) up to the depth  $\Delta$  to get the estimated cost of sorted accesses<sup>9</sup>. 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 is not available, one can still generate “dummy” synthesis based on some assumed distribution (*e.g.*, uniform) that is probably only a crude approximation. Though such samples cannot represent actual score distributions, they help optimize for other important aspects, such as the effects of scoring function and asymmetry between predicates. While our optimizer will certainly benefit from more accurate samples and statistics, Section 9 will implement our optimization framework using dummy synthesis, 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 two subtasks of identify  $\Delta_{opt}$  first and  $\mathcal{H}_{opt}$  in turn. In fact,  $\Delta$ -optimization and  $\mathcal{H}$ -optimization are mutually dependent. That is, we can identify the optimal  $\Delta$  with respect to some random access scheduling  $\mathcal{H}_0$ . Denoting this optimal depth identified as  $\Delta_0$ , the optimal random access scheduling for such depth may not be identical to  $\mathcal{H}_0$ , which we denote as  $\mathcal{H}_1$ . In other words, due to the mutual recursiveness in  $\Delta$ - and  $\mathcal{H}$ -optimization, finding an optimal pair of  $(\Delta_{opt}, \mathcal{H}_{opt})$  is essentially continuing the above iterations until the convergence. However, for simplicity, our two-phase approach was designed to perform a one-iteration approximation of the above iterative processes. That is, we find  $\Delta_{opt}$  with respect to some initial random access scheduling  $\mathcal{H}_0$  and find  $\mathcal{H}_{opt}$  with respect to  $\Delta_{opt}$  identified in the first phase, as we discuss further below.

- $\Delta$ -optimization: We first identify the optimal depth  $\Delta_{opt}$ , with respect to some initial schedule  $\mathcal{H}_0$ , *i.e.*,  $\Delta_{opt} = \operatorname{argmin}_{\Delta} \mathcal{C}((\Delta, \mathcal{H}_0))$ .
- $\mathcal{H}$ -optimization: We then identify the optimal scheduling  $\mathcal{H}_{opt}$  with respect to  $\Delta_{opt}$  identified.

For  $\mathcal{H}$ -optimization, we adopt global predicate scheduling proposed in our preliminary study [Chang and Hwang 2002] that uses online sampling to identify a predicate scheduling with the highest filtering effect, according to the notion of aggregate selectivity discussed in 7.2. For  $\Delta$ -optimization, we first study how it is specific to runtime factors, *e.g.*, score functions, predicate score distributions, and cost scenarios, as Example 8 will illustrate.

<sup>9</sup> $\Delta$  corresponds to a set of threshold scores to reach, which will stay the same for samples of any size as long as they preserve the statistical properties of the dataset.

**Example 8 ( $\Delta$ –Optimization Possibilities):** To illustrate, we continue Example 7 with a different depth configuration  $\Delta_2 = (0.8, 1)$ . In fact,  $\Delta_2$  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  $\Delta_1 = (0.8, 0.8)$  required 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  $Q_1$ –For instance, when scoring function  $\mathcal{F}$  is *avg* (the average function) for the same query  $Q_1$ ,  $\Delta_1$  requires less accesses (4 accesses) than  $\Delta_2$  (6 accesses). ■

Consequently, we need search schemes that systematically adapt to the given query, in exploring the  $\Delta$ –space, *i.e.*,  $m$ -dimensional space of  $\delta_1 \times \dots \times \delta_m = [0 : 1]^m$ . In particular, we propose three search schemes. First, we implement an approximate exhaustive search to the  $\Delta$ –space, we discretize each dimension  $\delta_i$  into a set of  $x$  finite values in the range  $[0:1]$ . We can then implement an exhaustive search scheme **Naive**, which checks each and every combination among  $x^m$  possible values. 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 towards 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, to avoid local minimum.

## 8. UNIFICATION AND CONTRAST

Framework **NC**<sup>10</sup>, in addition to being a general and adaptive optimization framework, enables conceptual *unification* of existing algorithms. It complements existing algorithms, by (1) generating similar behaviors when such behaviors are desirable, while (2) optimizing beyond their “static” behaviors when such behaviors are not desirable. In particular, we first discuss how it 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], to be more comparable, we transform **NC** to handle this restriction (while **NC** can generally work with or without). 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–  $u$  will remain *unseen* until hit by some sorted access, when it becomes *seen*. We introduce a *virtual* object **unseen** to represent all unseen objects. Note all such objects share the same maximal-possible score  $\overline{\mathcal{F}}[\text{unseen}] = \mathcal{F}(\overline{p}_1, \dots, \overline{p}_m)$ . This virtual object needs special handling, as Figure 13 shows with query  $Q_1$ : First, initially all objects are unseen, so **NC** now initializes  $\mathcal{K}_{\mathcal{P}}$  with only the **unseen**. Second, when this **unseen** is at the top (*e.g.*, step 1), its induced choices  $N_{\text{unseen}}$  will contain only sorted accesses, since random access is not allowed for an “unseen” object, by the no-wild-guesses assumption. Third, objects

<sup>10</sup>For notational simplicity, we use **NC** interchangeably as an abstract framework and as the optimal algorithm generated.

step	$\bar{p}_1$	$\bar{p}_2$	$\mathcal{K}_p$	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 guesses.

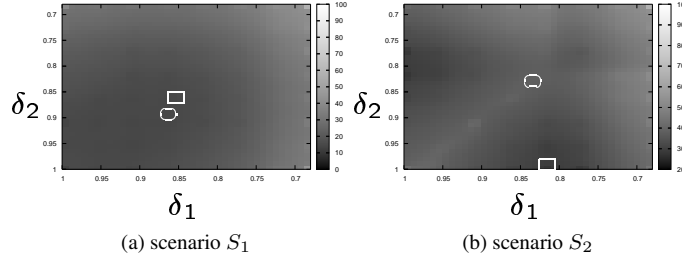


Fig. 14. Illustration of TA and NC.

hit by some sorted access will become “seen” (e.g.,  $u_3$  seen by  $sa_1$  at step 1)– They will be then handled as usual and may surface to  $\mathcal{K}_P$  (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- $k$  algorithms of all. As Figure 2 lists, TA aims at access scenarios where sorted access and random access have uniform unit costs, i.e.,  $\frac{cr_i}{cs_i} \approx 1$ . Let’s call it the *uniform scenario*. As illustrated in Example 2, the “behaviors” of TA can be characterized as follow: (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. (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, e.g.,  $\mathcal{F} = avg$  with equal predicate access costs, NC will indeed generate TA which is build for such scenario in mind: We illustrate with 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  $[0 : 1]$  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  $\Delta_{opt}$ , or the darkest cell marked by a rectangle, at around  $(.85, .85)$ . To compare, 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 only perform less random accesses than TA. We thus expect NC to be slightly better overall. (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

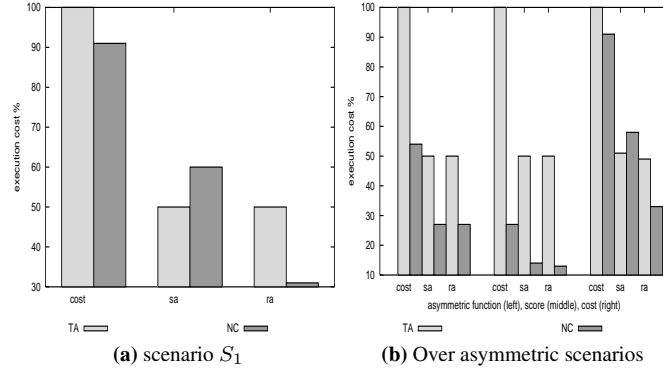


Fig. 15. Comparison of TA and NC when  $sa = \sum_i S_i \cdot cs_i$  and  $ra = \sum_i R_i \cdot cr_i$ , i.e.,  $cost = sa + ra$ .

scenarios, in the “asymmetric” cases, TA’s characteristic behaviors cannot adapt well. To illustrate such cases, Figure 14(b) shows scenario  $S_2$  by only changing  $\mathcal{F}$  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 a sense that only one predicate with minimum score determines the overall score. Observe how NC adapts beyond TA and thus generates a rather different algorithm with less cost (thus a darker cell). 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 above). In such a case, both algorithms behave similarly (except NC slightly outperforms by going deeper in sorted accesses to trade future random accesses.) To contrast, as “asymmetric” cases, 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  $\mathcal{F} = \text{avg}$  in  $S_1$ , where each predicate equally contributes  $\mathcal{F}$ , the optimal configuration for 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  $p_2$  is distributed normally with mean .2 and variance .1, NC adapts to perform more sorted access 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 expensive accesses by cheaper alternatives— When  $p_1$  is three times more expensive (for both sorted access 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], a simpler random-only scenario (similar to Upper [Bruno et al. 2002]

and  $\text{TA}_Z$  [Fagin et al. 2001] as we will discuss later). Consider  $\mathcal{F}(p_1, \dots, p_m, p_{m+1}, \dots, p_n)$ . **MPro** distinguishes two types of predicates: While  $p_{m+1}, \dots, p_n$  are simply *ordinary* (or “indexed”) predicates, the other group  $p_1, \dots, p_m$  are *expensive* predicates— They are “probe only” or *random-only*.

- $\forall p_i \in \{p_1, \dots, p_m\}$ :  $p_i$  supports only random access with unit cost  $cr_i$ ; thus  $cs_i = \infty$ .
- $\forall p_i \in \{p_{m+1}, \dots, p_n\}$ :  $p_i$  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, it works as follows:

1. Merge the ordinary predicates  $p_{m+1}, \dots, p_n$  into one single list  $x$  (or, a conceptual predicate), using **TA**. By this merging, we view the scoring function as  $\mathcal{F}(p_1, \dots, p_m, x)$ .
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  $P_j = \{ra_i(v_j) \mid p_i[v_j] \text{ is unevaluated so far}\}$ . Pick a probe  $ra_i(j)$ , according to some predicate schedule  $\mathcal{H}$ , from  $P_j$  and execute it. 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, *i.e.*,  $p_{m+1}, \dots, p_n$ , from the random access-only ones, *i.e.*,  $p_1, \dots, p_m$ , 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  $\forall v \in \mathcal{K}_{\mathcal{P}} : \mathcal{F}[v] \geq \mathcal{F}(1, \dots, 1, \overline{p_{m+1}}, \dots, \overline{p_n})$ , where  $\mathcal{K}_{\mathcal{P}}$  is the final top- $k$  answers, and the depths  $\overline{p_{m+1}}, \dots, \overline{p_n}$  are determined by the merging algorithm **TA**. (3) *p-minimization*: For those retrieved objects, **MPro** will minimize probe cost, by finding an optimal  $\mathcal{H}$ .

**Unification**: As **MPro** aims at minimizing random accesses, for expensive predicates, we can see that **NC**, if “projected” to *only* these predicates, can generate **MPro**. (With this projection, let’s ignore *x-separation*, which we will discuss later.) First, **NC** will satisfy the same *x-stop* condition: Note that the *unseen* object from the  $x$ -list have  $\overline{\mathcal{F}}[\text{unseen}] = \mathcal{F}(1, \dots, 1, \overline{p_{m+1}}, \dots, \overline{p_n})$ , and thus the stop-condition holds similarly as for **TA** just discussed.

Second, **NC** will naturally perform the same *p-minimization*: As outlined above, for probe-only predicates, **MPro** essentially operates on the same machinery as **NC**: *i.e.*, 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  $P_j$ — with *only* random access probes— for further probing, corresponding to **alternatives** of **NC**.

Such a “probe-only” scheme is indeed a *special case* that **NC** will unify: As a general mechanism, for such  $v_j$ , it will construct  $N_j$  with *both* sorted access and random access (line 5, Figure 9). As these probe-only predicates do not support sorted accesses, such “special restrictions” are naturally captured by the cost-based framework simply by setting  $cs_i = \infty$ . The optimizer (Section 7) will then algorithmically “ignore” the sorted accesses, by configuring the sorted access depths as  $\Delta$ : ( $\delta_1=1, \dots, \delta_m=1$ )— *i.e.*, no sorted access at all. Note that **MPro** *implicitly* assumes the same  $\Delta$ , by using the maximal-possible scores of those  $p_i$  as  $\overline{p_i}=1, \forall i \in [1 : m]$ . Thus, in principle, with respect to this same  $\Delta$ , our optimizer will generate the same global schedule  $\mathcal{H}$ . In summary, **NC** adapts to achieve the same *p-minimization*, while using a general-purpose mechanism.

**Contrast:** Unified for probe-only predicates, however, the two algorithms differ fundamentally for the “ordinary” predicates with both sorted access and random access. While NC integrates their optimization in exactly the same framework, MPro isolates (and thus ignores) such predicates by  $x$ -separation. By using TA as a “blackbox” for merging  $p_{m+1}, \dots, p_n$ , MPro will suffer all the contrasts that Section 8.1 discussed: In particular, equal-depth-sorted access into these predicates, and exhaustive-random access.

**Remark:** Although correctly unified for the “probe-only” special-case, NC has generalized beyond MPro significantly, by generally handling both sorted access and random access. Such a generalization is non-trivial. Essentially, as Section 4 identified, sorted access is fundamentally different with its progressiveness and side-effects. By focusing on only random accesses, MPro does not deal with defining a complete framework: To illustrate, consider a random-only setting, when some  $v_j \in \mathcal{K}_P$  is incomplete, and thus with further accesses, say,  $P_j = \{ra_1(v_j), ra_2(v_j)\}$ . To contrast, in its sorted-also counterpart, by adding sorted accesses, the further accesses are  $N_j = \{sa_1, ra_1(v_j), sa_2, ra_2(v_j)\}$ .

To contrast, 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 current top- $k$ ), simply by 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 what such a schedule is, or what to schedule exactly: As every sorted access can repeat for progressive accesses, there are generally an infinite number of such schedules; e.g., for  $N_j$ :  $(sa_1, sa_2, ra_2(v_j), ra_1(v_j)), (sa_1, sa_1, sa_2, ra_2(v_j), ra_1(v_j)), \text{etc.}$

In summary, NC generalizes MPro, which focuses only on random-only and thus reduces the optimization to a “barebone” of finding the optimal predicate scheduling  $\mathcal{H}$ : In these limited scenarios, Properties 1 and 2 above together *univocally* determine a *required* probe on a particular  $v_j$  for a particular  $p_i$ , i.e., *the necessary probe principle* [Chang and Hwang 2002]. Targeting at the same scenarios, Upper applies the same principle of evaluating the object with the highest maximal score; However, their runtime adaptation heuristics further restricts its applicability to weighted average scoring functions, in addition to being limited to random access optimization, by using weights for predicate scheduling.  $TA_Z$  similarly evaluates the object with the highest maximal score, but it 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 to handle both random access and sorted access, by 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, and adaptivity of NC, Section 9.1 studies the performance of NC over a wide range of middleware settings, by simulating over extensive synthetic scenarios. Second, to validate practicality, Section 9.2 quantifies the absolute performance of NC over real

retrieval size ( $k$ )	100
number of predicates ( $m$ )	2
predicate score distribution	uniform
database size ( $n$ )	10000
scoring function ( $\mathcal{F}$ )	$avg(p_1, \dots, p_m)$
unit costs	$\forall i : cs_i = cr_i = 1$
score distributions	$\forall i : D_i = unif$

Fig. 16. Default setting.

Web sources. Our experiments were conducted with a Pentium III 933MHz machine with 256M RAM, using our implementation of NC in Python. Note, 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 these 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) 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 shows. In particular, we compare with 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, though 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 its applicability over our synthesized scenarios. For instance, they cannot support  $\mathcal{F}=\min$  (*e.g.*,  $Q_1$ ). Second, the rest of algorithms can be considered as the 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 to MPro, 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 a variance .16.

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

Fig. 17. Unit cost scenarios.

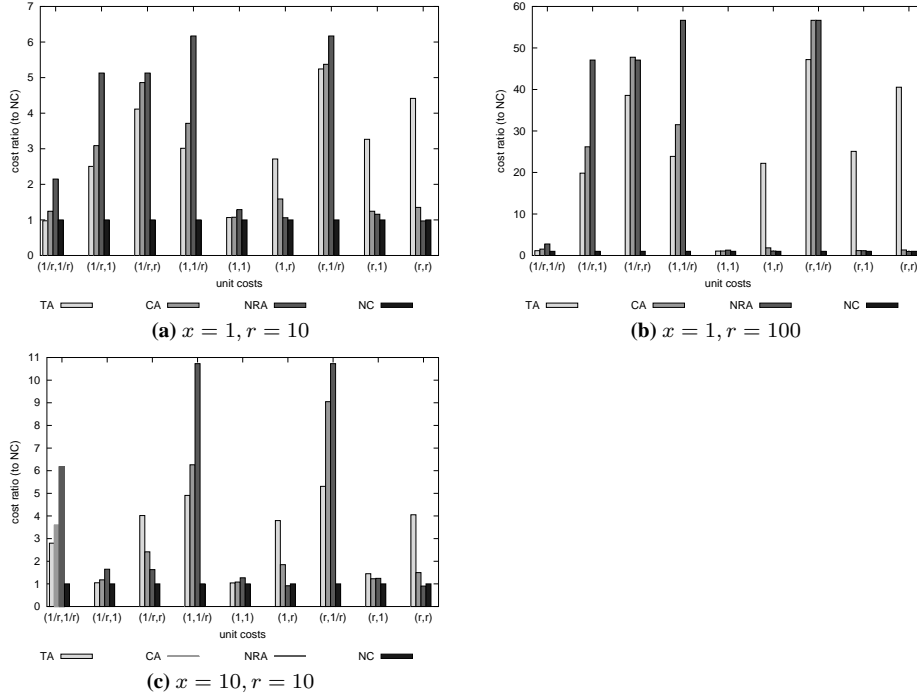


Fig. 18. Adaptivity of NC over unit costs

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 scenarios— 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 addition to the robust performances of NC, it is also interesting to observe how NC unifies existing algorithms. For instance, in uniform cost scenarios, *e.g.*,  $(h_1, h_2) = (1, 1)$ , which is ideal for TA, NC will similarly perform equal-depth sorted accesses (Section 8) and perform comparably to TA. For scenarios where random access is expensive, which is ideal for CA, *e.g.*,  $(h_1, h_2) = (10, 10)$ , NC unifies CA and NRA, and performs comparably by similarly trading some expensive random accesses with cheaper

ACM Transactions on Database Systems, Vol. V, No. N, Month 20YY.

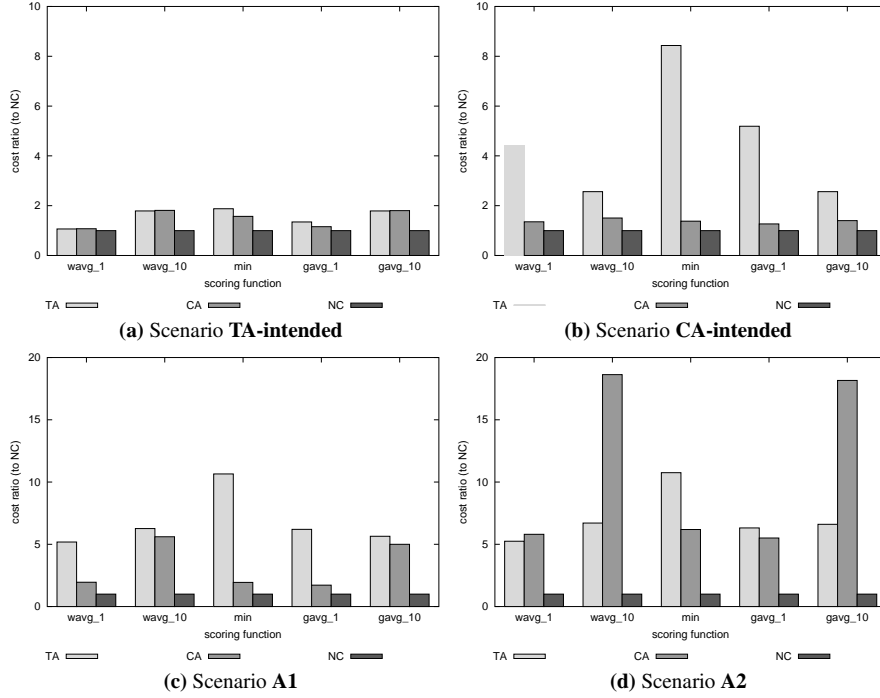


Fig. 19. Adaptivity of NC over scoring functions.

sorted accesses.

However, in other scenarios where existing algorithms are ineffective, NC 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 the performances when the asymmetry across the access costs of sorted and random accesses are more significant, *i.e.*,  $r = 100$ . Observe that the cost saving 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 saving of NC is more significant in Figure 18(c) where the asymmetry of access costs across predicates are more significant, *i.e.*,  $x = \frac{cs_2}{cs_1} = 10$ . Note, 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 the previous evaluations. In summary, NC performs robustly across wide ranges of scenarios and significantly outperforms existing algorithms when they cannot adapt to the asymmetry in costs, as consistently observed in Section 8.

We now study how NC adapts to other cost factors—*i.e.*, 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—To show the unification behaviors, we pick the intended scenarios for TA and CA, *i.e.*,  $(h_1, h_2) = (1, 1)$  and  $(r, r)$  respectively, which we denote as Scenarios **TA-intended** and

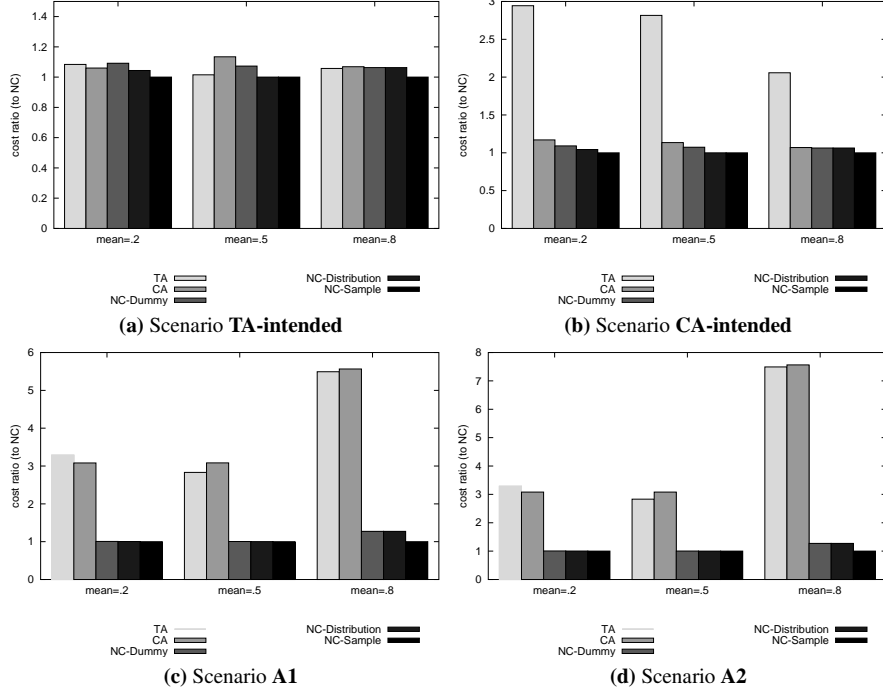
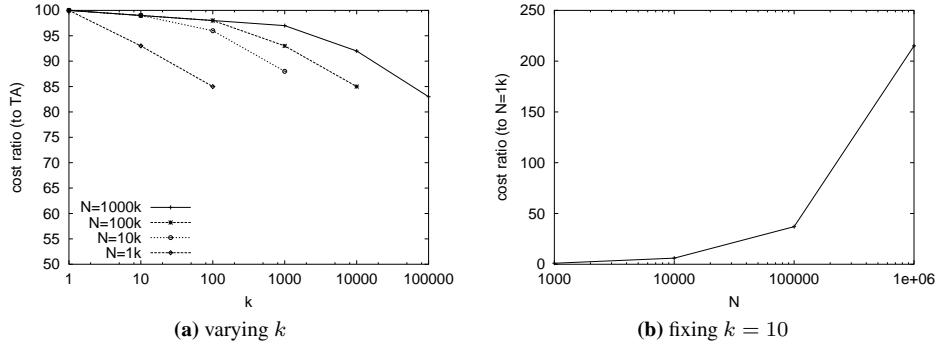


Fig. 20. Adaptivity of NC over score distributions.

Fig. 21. Scalability of NC over  $N$ .

**CA-intended.** In contrast, 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 as Scenario **A2**). For brevity sake, we drop the comparison with NRA, which has shown mostly worse or sometimes similar performances to CA in various settings 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 18a). As Section 8 discussed and also

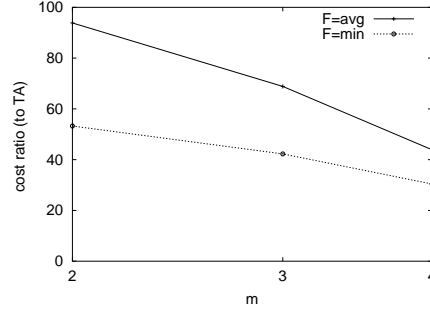


Fig. 22. Scalability of NC over  $m$ .

observed here, when function is symmetric, *e.g.*,  $\mathcal{F} = avg$ . However, in all other functions with asymmetry, *e.g.*,  $\mathcal{F} = wavg$  or  $\mathcal{F} = gavg$ , NC will outperform existing algorithms that fail to adapt to such asymmetry. Observe that, the cost differences grow significantly as such the asymmetry of weighted and geometric average increases, *e.g.*, the weight ratio  $c = \frac{w_2}{w_1}$  increases 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 true across different cost scenarios, only the cost savings are more dramatic 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 18a). Recall that, in all our evaluations, we used NC using dummy synthesis (Section 7.2) instead of actual sampling. Meanwhile, in this experiment only, we also implement two versions of NC with closer samples that represent the data distribution: 1) NC-Sample that uses actual samples of 0.1% of data size, 2) NC-Distribution that uses synthetic “approximate” samples that are generated by an assumed 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 (*e.g.*, unit costs)— Observe that, in Scenarios **A1** and **A2**, while NC-Dummy does not know the actual distribution, it significantly outperforms existing algorithms by optimizing to other runtime factors such as query size or access costs, 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**, which is the most “difficult” scenario as the cost saving of NC was minimal in the previous experiments. Figure 21(a) presents the cost ratio of NC with respect to TA which is ideal for the given scenario, varying  $N = 1k, 10k, 100k$ , and  $1000k$  in the default setting with  $x = 1$  and  $r = 10$ . It is interesting to observe that, the cost ratios are approximately the same if the relative retrieval size is the same, regardless of the database size. For instance,

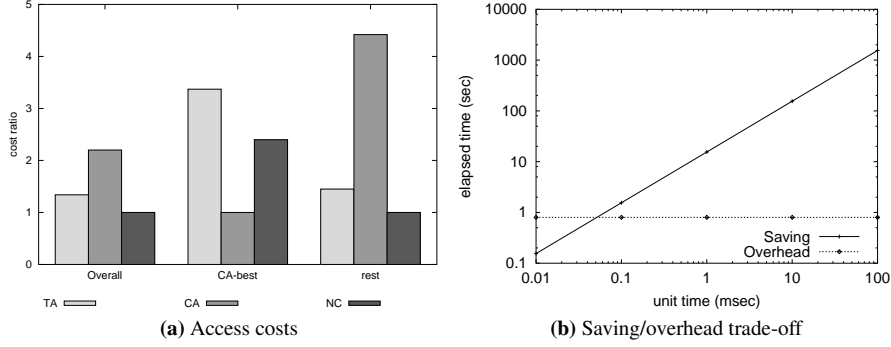


Fig. 23. Generality of NC.

for  $k = N \cdot 1\%$  (i.e.,  $k = 10, 100, 1000$ , and  $10000$  for  $N = 1k, 10k, 100k$ , and  $1000k$ ), the cost ratios are all about 95%. This observation shows that NC has good *data scalability*—That is, the cost ratio for finding  $k$  answers from database of size  $sN$  will be *less than*  $s$  times that for finding the same number of an answers from database of size  $N$ . Such data scalability is promising, considering the retrieval size  $k$  tends to stay small regardless of  $N$ , e.g., users retrieving only the first page or two of search results regardless of the number of hits—The same evaluation suggests that when  $k = 10$ , as  $N$  increases from  $N = 1k$  to  $N = 10k, 100k$ , and  $1000k$ , i.e., 10-, 100-, and 1000-fold, the cost increases sub-linearly by 6-, 37-, and 215-fold, 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  $m$ , i.e., the cost saving of NC will scale up as there are more predicates and thus a larger room for optimization.

**Generality of NC:** To validate the *generality* of NC over existing algorithms, we now study the performance NC over 1000 synthetic middleware scenarios, varying all the three performance factors above at the same time: In particular, 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  $cr_i > cs_i$ .) We also simulated different scoring functions, by randomly generating weight  $w_i$  in [1:100] of  $\mathcal{F} = w_{avg}$ , for each configuration. Further, to simulate different score distributions, we randomly generate the mean for score distribution  $D_i = norm$ .

Figure 23(a) compares the average total access cost of the three algorithms over these 1000 random configurations. Observe that, overall, NC as a unified algorithm saves from TA and CA by 25% and 55% in average, by successfully adapting to a combination of cost factors. We then, for closer observation, divide the 1000 settings into two groups in which CA works the best (about 25% of the configurations, which we denote as “CA-best”) and the rest. From such grouping, we can observe how NC unify and contrast with an existing algorithm like CA: First, unification: in the CA-best scenarios (the middle bar group of Figure 23a), NC is still the second-best, with a smaller cost margin to CA than TA does. Second, contrast: in the rest of the scenarios (i.e., 75% of the configurations represented by the rightmost group in Figure 23a), NC is indeed the best, outperforms CA significantly.

Further, as one would naturally wonder, will the overhead of run-time optimization justifies its cost saving in access costs? Clearly, such optimization is only justified if it can enable more access cost saving than the computation overhead of optimization. To analyze this trade-off, we also compare this saving to overhead: As the saving of access cost will have different impacts depending on the actual response time  $t$  of a single unit cost, Figure 23(b) compares this saving and overhead, with respect to  $t$ . (Note we compare only with CA, as it outperformed TA in Figure 23a.) Observe that the optimization overhead of NC can be justified in a large range of realistic scenarios— 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 non-trivial 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 experiments in Section 9.2.2.

**9.2.1 Parallelizing NC for Concurrent Accesses.** We now develop 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  $C$ , i.e., 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 outstanding accesses do not exceed the concurrency limit  $C$ . The queue  $\mathcal{K}_{\mathcal{P}}$  is updated asynchronously as well, whenever an access completes.

While such extension achieves speedup by overlapping 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  $\bar{p}_i$  from the preceding  $sa_i$  has not reached the suggested depth  $\delta_i$ , i.e.,  $\bar{p}_i > \delta_i$ . Note the up-to-date  $\bar{p}_i$  is not known until all outstanding sorted accesses complete.

However, as waiting to get the exact  $\bar{p}_i$  defeats the whole purpose of concurrent accesses, we continue with an estimated  $\bar{p}_i$  instead, by computing its expected decrement  $D_i$ : Assuming  $d_i$  is the expected decrement of  $\bar{p}_i$  after a single  $sa_i$  and  $n_i$  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 returned from accesses.

Note, 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 of the  $m$  objects seen), while CA may withhold such random accesses after  $h = \frac{ct_i}{cs_i}$  iterations of parallel sorted accesses (to trade expensive random accesses with cheaper sorted accesses.) Observe, TA, by issuing such sorted and random 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 will bound the concurrency of NC to  $C = m^2$ .

**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

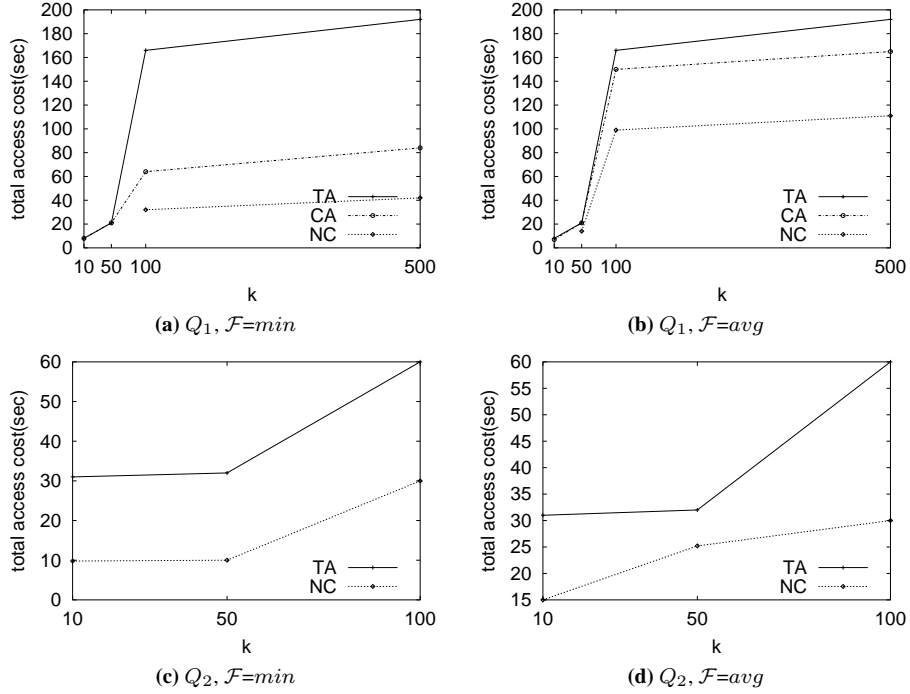


Fig. 24. Comparison of total access costs.

queries. For extensive evaluation, we combine each with different scoring functions, *i.e.*, *min* and *avg*. 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  $[0 : 1]$ , *e.g.*, *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, with concurrency, the elapsed time is typically shorter, by overlapping some high-latency accesses. Using these metrics, we compare NC with TA and CA. However, note 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  $\mathcal{F}=\text{avg}$  respectively. Observe that NC significantly outperforms TA, as 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% from that of 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 TA, to

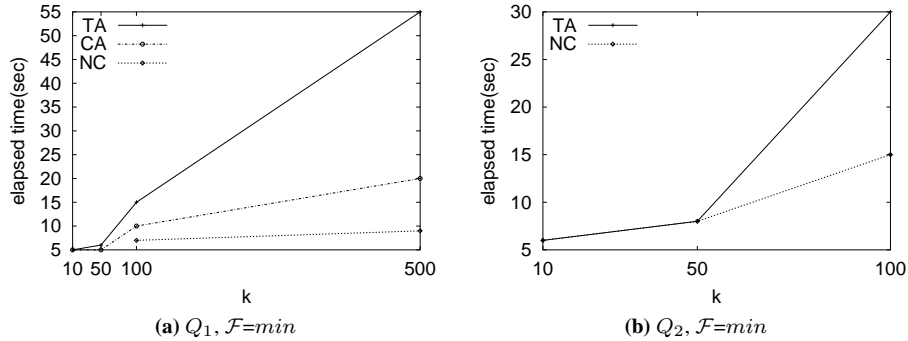


Fig. 25. Comparison of elapsed time.

trade random accesses with cheaper sorted accesses.

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, *e.g.*, when  $k = 10$  in Figure 24(c), NC saves up to 66% from the access cost of TA. However, in contrast to  $Q_1$ , as random accesses are cheaper than sorted accesses 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 *focused* configuration in Example 8), while TA still performs sorted accesses to every predicate with no adaptation. Note, 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 and elapsed time of  $Q_1$  (*i.e.*, Figure 24a and Figure 25a), we observe similar relative behaviors, though in different cost ranges due to the parallel speedup. This consistency suggests that NC can benefit from concurrency to the same extent as TA and CA which have inherent parallelism. The same observation holds true 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 from concurrency, while TA with more accesses overlaps them and performs comparably.

## 10. CONCLUSION

This paper 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, yet adapting at runtime to and even outperforms existing algorithms specifically designed for their scenarios.

## REFERENCES

- BALKE, W., GUENTZER, U., AND KIESSLING, W. 2002. On real-time top-k querying for mobile services. In *CoopIS 2002*.
- BRUNO, N., GRAVANO, L., AND MARIAN, A. 2002. Evaluating top-k queries over web-accessible databases. In *ICDE 2002*.

- CAREY, M. J. AND KOSSMANN, D. 1997. On saying "enough already!" in SQL. In *SIGMOD 1997*.
- CAREY, M. J. AND KOSSMANN, D. 1998. Reducing the braking distance of an SQL query engine. In *VLDB 1998*.
- CHANG, K. C.-C. AND HWANG, S. 2002. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD 2002*.
- CHAUDHURI, S. AND GRAVANO, L. 1999. Evaluating top- $k$  selection queries. In *VLDB 1999*.
- DONJERKOVIC, D. AND RAMAKRISHNAN, R. 1999. Probabilistic optimization of top  $n$  queries. In *VLDB 1999*.
- FAGIN, R. 1996. Combining fuzzy information from multiple systems. In *PODS 1996*.
- FAGIN, R., LOTE, A., AND NAOR, M. 2001. Optimal aggregation algorithms for middleware. In *PODS 2001*.
- GUENTZER, U., BALKE, W., AND KIESSLING, W. 2000. Optimizing multi-feature queries in image databases. In *VLDB 2000*.
- GUENTZER, U., BALKE, W., AND KIESSLING, W. 2001. Towards efficient multi-feature queries in heterogeneous environments. In *ITCC 2001*.
- HELLERSTEIN, J. M. AND STONEBRAKER, M. 1993. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD 1993*.
- SELINGER, P., ASTRAHAN, M., CHAMBERLIN, D., LORIE, R., AND PRICE, T. 1979. Access path selection in a relational database. In *SIGMOD 1979*.