SUPPORTING RANKING FOR DATA RETRIEVAL

BY

SEUNG-WON HWANG

B.S., Korea Advanced Institute of Technology, 1998
M.S., University of Illinois at Urbana-Champaign, 2000

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2005

Urbana, Illinois
SUPPORTING RANKING FOR DATA RETRIEVAL

SEUNG-WON HWANG, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 2005
Prof. Kevin Chen-Chaun Chang, Advisor

The explosion of internet usage has provided users with access to information in an unprecedented scale—The data retrieval problem of finding relevant data has thus become a clear challenge. Such retrieval, with the large scale of data, has naturally demanded ranked answers, or “best first,” to enable users to focus on a few top results.

This thesis presents techniques to support this ranked data retrieval efficiently and effectively. First, efficient processing: As data retrieval scenarios naturally situate in large dataset, such retrieval should be amenable to efficient processing. Second, intuitive formulation: Data retrieval should be intuitive for ordinary users to easily express their ranking criteria.

Toward this goal, we first study how to support efficient rank processing. Existing rank processing algorithms are all designed with specific access scenario in mind and follow statically designed behaviors. As a result, it is often unclear how to support data retrieval scenarios involving sources with heterogeneous access capabilities and dynamically changing costs. In contrast, we develop a systematic “cost-based” optimization—By dynamic search over a space of algorithms, cost-based optimization is general across a wide range of access scenarios, yet adaptive to any specific scenario at runtime. We develop this cost-based optimization framework, first for an important and fundamental special case of “expensive predicates”, and then extend for any arbitrary scenarios.

Second, we study how to support intuitive rank formulation. While existing rank processing algorithms require a global ranking function $F$, it is far from trivial for everyday user to articulate such a function $F$ evaluating each and every object into a numerical score. We thus develop an interactive “rank-by-examples” paradigm to support intuitive rank formulation. In particular, we apply a machine learning approach to infer the underlying ranking function from a few given examples.

While building this system, we also observe that ranking functions learned often turn out to be quite complicated. Meanwhile, rank processing techniques have been focusing on supporting monotonic functions. We address this new challenge by developing a fresh new perspective ab-
stracting query answering as an optimization problem, which enables to extend rank processing for any arbitrary functions.
To my family
ACKNOWLEDGMENTS

During my graduate years, I have been truly fortunate to have the greatest mentors. My advisor, Prof. Kevin Chang, has constantly challenged me to grow into an independent researcher. It often amazed me how he could make himself available to students and strive for the excellency in every work we do. I thank Prof. Marianne Winslett and Prof. Geneva Belford for helping me get over the biggest hurdle in my graduate years. I thank Prof. Jiawei Han and Prof. Chengxiang Zhai for being the readers of my thesis and providing valuable comments. I thank Dr. Kaushik Chakrabarti, Dr. Surajit Chaudhuri, Dr. Ekow Otoo, and Dr. Arie Shoshani for guiding my summer research at Microsoft Research and Berkeley National Laboratory. Those were exciting four summers! I also thank Prof. Werner Kiessling for hosting my visit at University of Augsburg.

I deeply thank Korea Foundation for Advanced Studies for supporting me with a fellowship for five years. I am indebted by my church friends, especially Jinju Lee, Sooyoung Kim, and Sun-A Kim, for their discipleship and fellowship. I thank my group mates, Zhen Zhang, Bin He, and Chengkai Li, for sharing my last three years with me both as great colleagues and friends. I thank my pseudo-family at school for making the life in Urbana-Champaign fun and exciting. I salute my six big sisters who started graduate studies with me, Eunyoung Jung, Heekyung Choi, Ejjung Choi, Hyunjoo Song, Kay Yoon, and Jinkyung Park, and my little sister Yoonkyong Lee. I thank my friends in the computer science department, especially Jonghyun Lee for his advice on many aspects of life, William Conner for his impeccable recommendations on movies, and Hwanjo Yu for exciting research we did together. I am also deeply indebted to my dearest friends all over the world. I thank Noll Park in Germany for being my best friend for the last 18 years. I thank Jeeyoung Kim in California for her constant support and encouragement. I thank my college buddies Hisun Kim, Hyejin Moon, and Seungwoon Kim and my old friends Seungyoon Lee and Sungah Jang, for their precious friendship.
I dedicate this my thesis to my parents and sister for their endless love and support. They are the greatest gift I’ve got from God. I especially miss my grandmother who would have been so happy to see the three generations of women PhDs in the family. Personally, this thesis is my offering to God. I am thankful for this opportunity to grow academically and in Him. I pray that I would never take this for granted and make a good use of it.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>LIST OF FIGURES</th>
<th>.................................................................</th>
<th>xi</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHAPTER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1   Introduction</td>
<td>.............................................................................</td>
<td>1</td>
</tr>
<tr>
<td>2   Preliminaries</td>
<td>.............................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.1  Query Semantics</td>
<td>.............................................................................</td>
<td>7</td>
</tr>
<tr>
<td>2.2  Cost Model for Middleware Accesses</td>
<td>..........................................................................</td>
<td>8</td>
</tr>
<tr>
<td>2.3  Current State-of-the-art</td>
<td>........................................................................</td>
<td>10</td>
</tr>
<tr>
<td>3   Probe Minimization by Cost-based Schedule Optimization</td>
<td>..................................................................</td>
<td>13</td>
</tr>
<tr>
<td>3.1  Minimal Probing: Cost-based Schedule Optimization</td>
<td>..................................................................</td>
<td>14</td>
</tr>
<tr>
<td>3.1.1 Schedule Optimization: Overview</td>
<td>.........................................................................</td>
<td>15</td>
</tr>
<tr>
<td>3.1.2 Object Scheduling: Analytic Principle</td>
<td>......................................................................</td>
<td>18</td>
</tr>
<tr>
<td>3.1.2.1 Necessary-Probe Principle</td>
<td>.........................................................................</td>
<td>19</td>
</tr>
<tr>
<td>3.1.2.2 Algorithm MPro</td>
<td>.............................................................................</td>
<td>23</td>
</tr>
<tr>
<td>3.1.3 Predicate Scheduling: Dynamic Optimization</td>
<td>..................................................................</td>
<td>26</td>
</tr>
<tr>
<td>3.1.3.1 Compositional Scheduling: Context Independence</td>
<td>..................................................................</td>
<td>27</td>
</tr>
<tr>
<td>3.1.3.2 Global Scheduling</td>
<td>.............................................................................</td>
<td>29</td>
</tr>
<tr>
<td>3.2  Extensions and Scalability</td>
<td>.............................................................................</td>
<td>33</td>
</tr>
<tr>
<td>3.2.1 Fuzzy Joins</td>
<td>.............................................................................</td>
<td>34</td>
</tr>
<tr>
<td>3.2.2 Approximate Queries</td>
<td>.............................................................................</td>
<td>36</td>
</tr>
<tr>
<td>3.2.3 Data-Partitioned Parallel Processing</td>
<td>..................................................................</td>
<td>37</td>
</tr>
<tr>
<td>3.2.4 Scalability</td>
<td>.............................................................................</td>
<td>39</td>
</tr>
<tr>
<td>3.3  Case Study: Integration with RDBMS for Post-Filtering Ranking</td>
<td>........................................</td>
<td>42</td>
</tr>
<tr>
<td>3.3.1 Overview</td>
<td>.............................................................................</td>
<td>42</td>
</tr>
<tr>
<td>3.3.2 Design</td>
<td>.............................................................................</td>
<td>43</td>
</tr>
<tr>
<td>3.4  Experiments</td>
<td>.............................................................................</td>
<td>45</td>
</tr>
<tr>
<td>3.4.1 Baseline Comparison</td>
<td>.............................................................................</td>
<td>46</td>
</tr>
<tr>
<td>3.4.2 Predicate Scheduling Effectiveness</td>
<td>....................................................................</td>
<td>48</td>
</tr>
<tr>
<td>3.4.3 Benchmarks over Real-Life Database</td>
<td>..................................................................</td>
<td>52</td>
</tr>
</tbody>
</table>


4 Extending to Unified Cost-based Optimization for Top-k Queries ................................................. 55
  4.1 Defining a Comprehensive Space ................................................. 55
  4.2 Defining a Focused Space ...................................................... 59
    4.2.1 Logical View: Scoring Tasks .............................................. 59
    4.2.2 Identifying Unsatisfied Tasks ........................................... 61
  4.3 Putting Together: Framework NC .............................................. 64
    4.3.1 The Framework ............................................................. 65
    4.3.2 Comprehensive and Focused Space ...................................... 68
  4.4 Search: Dynamic Optimization ................................................ 70
    4.4.1 Space Reduction ............................................................. 71
    4.4.2 Search ........................................................................... 73
    4.4.3 Cost Estimation .............................................................. 75
  4.5 Unification and Contrast .......................................................... 76
    4.5.1 Algorithm TA ................................................................. 76
    4.5.2 Algorithm MPro .............................................................. 79
  4.6 Experiments ........................................................................ 82
    4.6.1 Real Web Scenarios ........................................................... 82
      4.6.1.1 Parallelizing NC for Concurrent Accesses ......................... 82
      4.6.1.2 Results ....................................................................... 83
    4.6.2 Synthetic Middleware Scenarios .......................................... 85

5 Integrating with Intuitive Rank Formulation ................................................................. 87
  5.1 Overview: Bridging Rank Formulation and Processing ......................................................... 87
  5.2 The RankFP Framework: Enabling Rank Formulation and Processing Online ....................... 90
    5.2.1 Duality of Ranking and Classification View ................................... 91
    5.2.2 Incorporating SVM Learning for Rank Formulation ......................... 92
      5.2.2.1 SVM Classification ....................................................... 93
      5.2.2.2 SVM for Ranking ........................................................ 93
      5.2.2.3 Incorporating Nonlinear Ranking ...................................... 95
    5.2.3 Satisfying Online Requirements .............................................. 96
      5.2.3.1 Online Formulation: Top Sampling ................................... 96
      5.2.3.2 Online Processing ....................................................... 98
  5.3 Experimental Evaluation ........................................................................ 99
    5.3.1 Experiment on Synthetic Queries ........................................... 100
    5.3.2 Experimental on Real Queries .............................................. 101

6 Supporting Non-monotonic Ranking Function ................................................................. 107
  6.1 Framework AsOpt: Top-k As Score Optimization ......................................................... 107
    6.1.1 Motivation ................................................................. 108
    6.1.2 Framework ............................................................... 109
    6.1.3 Unification and Generalization ............................................ 112
  6.2 Implementation: Using Boolean Selection Queries ................................................... 114
  6.3 Implementation: Dealing with Middleware Source Restrictions ............................. 116
  6.4 Discussion: Collective I/O ..................................................... 120
  6.5 Experiments ................................................................. 122
7 Related Work ................................................................. 128
  7.1 Rank Processing .......................................................... 128
  7.2 Rank Learning ............................................................ 129

8 Conclusions ................................................................. 130
  8.1 Summary ................................................................. 130
  8.2 Future Work ............................................................. 131
    8.2.1 Categorization ..................................................... 131
    8.2.2 Fuzzy Joins ....................................................... 132

REFERENCES .............................................................. 134

AUTHOR'S BIOGRAPHY ................................................ 137
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Web query scenarios for (a) ( Q_1 ) and (b) ( Q_2 ).</td>
<td>3</td>
</tr>
<tr>
<td>1.2 Framework RankFP: Rank formulation and processing for data retrieval.</td>
<td>5</td>
</tr>
<tr>
<td>2.1 Dataset 1</td>
<td>9</td>
</tr>
<tr>
<td>2.2 Access scenarios and their proposed algorithms.</td>
<td>10</td>
</tr>
<tr>
<td>3.1 Dataset 2 for query ( \mathcal{F}(x, p_c, p_l) = \min(x, p_c, p_l) ).</td>
<td>15</td>
</tr>
<tr>
<td>3.2 Algorithm MP( \text{Ro} )</td>
<td>24</td>
</tr>
<tr>
<td>3.3 Illustration of OSch.</td>
<td>25</td>
</tr>
<tr>
<td>3.4 Dataset 3</td>
<td>27</td>
</tr>
<tr>
<td>3.5 Dataset 4 for join query ( \mathcal{F} = \min(x, p_r, p_j) ).</td>
<td>35</td>
</tr>
<tr>
<td>3.6 Illustration of MP( \text{Ro} ) for a join query.</td>
<td>35</td>
</tr>
<tr>
<td>3.7 ( \text{dpMP}( \text{Ro} )</td>
<td>39</td>
</tr>
<tr>
<td>3.8 Two possible execution plans generated by PostgreSQL to evaluate ( Q_4 ).</td>
<td>43</td>
</tr>
<tr>
<td>3.9 Two possible execution plans to evaluate transformed ( Q_4 ).</td>
<td>45</td>
</tr>
<tr>
<td>3.10 Different score distributions (a) and scoring functions (b) with ( m = 1k ).</td>
<td>47</td>
</tr>
<tr>
<td>3.11 Different database sizes ( m ) with ( \mathcal{F} = \text{norm} ).</td>
<td>48</td>
</tr>
<tr>
<td>3.12 Scheduling for random configurations.</td>
<td>52</td>
</tr>
<tr>
<td>3.13 Benchmark queries.</td>
<td>52</td>
</tr>
<tr>
<td>3.14 (a)–(c): Probe ratio for ( Q_1 - Q_3 ). (d)–(f): Elapsed time for ( Q_1 - Q_3 ).</td>
<td>53</td>
</tr>
<tr>
<td>4.1 Cost comprehensiveness.</td>
<td>56</td>
</tr>
<tr>
<td>4.2 Dataset 4</td>
<td>56</td>
</tr>
<tr>
<td>4.3 Example algorithms.</td>
<td>57</td>
</tr>
<tr>
<td>4.4 Algorithm “skeleton” SE( \text{Q} ).</td>
<td>58</td>
</tr>
<tr>
<td>4.5 The score state of Example 11.</td>
<td>62</td>
</tr>
<tr>
<td>4.6 Framework NC.</td>
<td>65</td>
</tr>
<tr>
<td>4.7 Illustration of NC.</td>
<td>67</td>
</tr>
<tr>
<td>4.8 Illustration of SR/G heuristics.</td>
<td>72</td>
</tr>
<tr>
<td>4.9 Select with SR/G heuristics.</td>
<td>72</td>
</tr>
<tr>
<td>4.10 NC with no wild guesses.</td>
<td>76</td>
</tr>
<tr>
<td>4.11 Illustration of TA and NC.</td>
<td>77</td>
</tr>
<tr>
<td>4.12 Comparison of TA and NC.</td>
<td>79</td>
</tr>
<tr>
<td>4.13 Default setting for synthetic dataset.</td>
<td>82</td>
</tr>
<tr>
<td>4.14 Comparison of total access costs.</td>
<td>84</td>
</tr>
</tbody>
</table>
5.1 The front-end rank formulation: “learning-till-convergence” .......................... 89
5.2 Margin maximized boundary in a two-dimensional space .............................. 94
5.3 Linear projection of ranking ........................................................................ 94
5.4 Preference queries ...................................................................................... 100
5.5 Definitions of fuzzy predicates ................................................................. 105
5.6 Performance convergence of two sampling techniques on the linearly rankable queries. TOP: top sampling; RAN: random sampling; X-axis: # of rounds; Y-axis: accuracy .................................................................................................................. 105
5.7 Distribution of user preferences generating over 90% (left) and 100% (right) accuracy. X-axis: # of rounds; Y-axis: # of preferences ................................................................. 106
5.8 Screen shot of our running system. The candidate set: houses in Chicago ……… 106

6.1 Example score space .................................................................................. 109
6.2 Illustration of framework AsOpt ............................................................... 110
6.3 Framework AsOpt ..................................................................................... 111
6.4 Crossover example ................................................................................... 112
6.5 Generating Algorithm TA ................................................................. 113
6.6 Illustration of object search using selection queries .................................. 115
6.7 Example query interface ........................................................................... 117
6.8 Query constraints of realtor.com ............................................................. 118
6.9 Illustration of object search for a web source .......................................... 118
6.10 Collective I/O techniques ....................................................................... 121
6.11 A benchmark query .............................................................................. 122
6.12 AsOpt over local databases ..................................................................... 124
6.13 AsOpt over web sources ........................................................................ 125
6.14 AsOpt-Collective with histograms with varying accuracy ....................... 126
CHAPTER 1

Introduction

The explosion of internet usage has provided users with access to information in an unprecedented scale—The data retrieval problem of finding relevant data has thus become a clear challenge. Such retrieval, with the large scale of data, has naturally demanded ranked answers, or “best first,” to enable users to focus on a few top results. A ranked query computes the scores of individual fuzzy predicates (typically normalized in $[0;1]$), combines them with a scoring function, and returns a small number of top-$k$ answers.

This thesis presents techniques to support this ranked data retrieval both efficiently and effectively:

1. **Efficient processing**: As data retrieval scenarios naturally situate in large datasets, such retrieval should be amenable to efficient processing.

2. **Intuitive formulation**: To be effective, such retrieval should be intuitive for ordinary users to easily express their ranking criteria.

To pursue the first goal of this thesis, we study the problem of efficiently processing ranked queries in the context of “middleware” (e.g., over multimedia subsystems or Web sources). To motivate, consider a Web middleware scenario for finding restaurants and hotels from web sources, as Examples 1 and 2 illustrate.

**Example 1**: To find top-5 restaurants (say, in the Chicago area) that are highly-rated and close to her place “myaddr,” a user may ask a ranked query $Q_1$ (in SQL-like syntax):

```sql
select name from restaurant r
order by min(rating(r), close(r, myaddr))
stop after 5
```

(Query $Q_1$)
For query answering, our middleware will access some Web sources to evaluate the predicates, e.g., rating and close into scores in $[0;1]$, which are then aggregated by some scoring function $\mathcal{F}$, e.g., $\mathcal{F} = \text{min}$, to determine the highest-scored 5 restaurants.

Our middleware can use various sources in query answering: To illustrate, Figure 1.1(a) shows one possible scenario: For evaluating close: superpages.com is capable of 1) returning the close score for a specific restaurant (“random access”) and 2) returning restaurants in their descending order of scores (“sorted access”). For rating: dineme.com similarly provides both sorted and random accesses.

The middleware will coordinate these accesses to find the top results. Such accesses are typically expensive (as compared to local computations) with varying costs: To characterize, Figure 1.1(a) shows the average access latency (thus including both network and server times) of both sorted and random access (denoted $cs$ and $cr$ respectively) for each predicate: In this scenario, random accesses are more expensive in both sources (i.e., $cr > cs$), but with different actual scales (i.e., $cr$) and ratios (i.e., $\frac{cr}{cs}$).

Example 2: Consider query $Q_2$ for the top-5 hotels that are close, with high star-rating, yet within the budget:

```
select name from hotel h
order by avg(close(h, myaddr), rating(h), cheap(h))
stop after 5
```

(Query $Q_2$)

Figure 1.1(b) describes another scenario, with hotels.com providing sorted access to all the predicates. In this setting, since a sorted access (e.g., for close) also retrieves all the attributes of a hotel (e.g., “stars” and “price”), the subsequent random accesses$^1$ to the same hotel are essentially of zero access costs ($cr = 0\text{ms}$) — e.g., using “stars” and “price,” the middleware can locally compute rating and cheap. This scenario thus significantly contrasts with expensive random accesses of Example 1.

Our goal is thus to develop efficient middleware algorithms, or “query plans” for coordinating sources, to minimize access costs. This task is challenging: First, sources are heterogeneous, with widely varying access capabilities and costs (e.g., as the real Web sources in Figure 1.1

---

$^1$In a middleware, random accesses to an object $h$ can only occur after $h$ is first “seen” from sorted accesses—or, “no wild guess” [19].
show)– Our algorithms must be general for various capability configurations. Second, sources can dynamic, with cost scenarios changing over time (e.g., Web sources depending on source load and availability). Our algorithm must be adaptive to runtime factors.

While many rank processing algorithms already exist, they do not satisfy these requirements: First, for generality, the existing algorithms have mostly been designed with specific cost scenarios in mind, which do not cover important access scenarios. Second, for adaptivity, algorithms lack systematic runtime optimization, with at most only limited heuristics.

To tackle these challenges, we take a two-phase approach:

- As an important and fundamental special case for general top-k optimization, we first study unexplored access scenarios where some sources support only random accesses. We characterize expensive predicates as those evaluated from such sources, evaluated only by a per-object random access, or a probe. Such scenarios generally abstract ranked queries, such as user-defined, external, post-filtering, and join predicates, as Chapter 3 will discuss. For such scenarios, to minimize probes in answering the given query, we develop “cost-based” optimization framework MPro.

- Later, we leverage and generalize the insight of MPro for general top-k optimization of minimizing costs of arbitrary accesses (not only random accesses, but also sorted and potentially beyond): By dynamic search over some space $\Omega$ of algorithms, cost-based optimization is general across virtually all cost scenarios, yet adaptive to the specific one at runtime.

While such optimization has been “taken for granted” for relational queries from early on [42], it has been clearly lacking for ranked queries. To begin with, what is a “comprehensive” yet “focused” algorithm space $\Omega$ for general top-k algorithms? Our developments answer this critical question by overcoming these dual barriers: First, to define a comprehensive space, we
show abstracting a top-$k$ algorithm as an access scheduling problem defines a comprehensive space $\Omega$. Second, to define a focused space, we identify the required “information” for answering a top-$k$ query and decompose it into logical tasks, based on which we further narrow down the space $\Omega$. With this comprehensive yet focused $\Omega$ defined, we develop systematic optimization schemes to effectively identify, in principle, the optimal algorithm in $\Omega$. Such search must balance both the overhead and the quality of optimization.

The second goal of this thesis is to support intuitive rank formulation. As it is far from trivial for everyday user to articulate a ranking function $\mathcal{F}$ representing how she would evaluate each and every object into a score, we believe an effective retrieval system should provide a more intuitive formulation front-end. First, it has been studied that ranking often stems from relative ordering (e.g., “I like A better than B”) rather than explicit scores. The system thus should allow users to specify only relative ordering or partial orders (and not absolute scores). Second, ranking often requires context knowledge of what objects are available in the database to be effectively formulated. The system should thus also present what are available in the database and let users focus on only those presented. Together, both requirements lead us to pursue an interactive “rank-by-examples” paradigm to support rank formulation—This need for “inference by examples” (for finding the implicit ranking criteria) clearly suggests a machine learning approach. With interactive sampling and labeling of training examples, our “learning machine” will infer the desired ranking expression.

However, it is not trivial to incorporate learning machines (e.g., SVM) for rank formulation. To begin with, unlike a conventional learning problem of classifying objects into groups, we need to learn a ranking function $\mathcal{F}$ for inducing an desired ordering of all objects. Consequently, this thesis develops how to “connect” ranking view (for processing back-end) and classification view (for learning front-end), and integrates them into a systematic framework—Figure 1.2 illustrates our integrated system RankFP (for RANK Formulation and Processing).

From this integrated framework, we observe that rank functions formulated from the front-end often turn out to be quite complicated—They may not be monotonic, in particular. Meanwhile, rank processing techniques have been focusing on supporting monotonic functions. We address this new challenge by developing a fresh new perspective abstracting query answering as an optimization problem, which enables to extend rank processing for any arbitrary functions.

To highlight, we summarize the main contributions of this thesis as follows:
Rank formulation and efficient processing.

The best of our knowledge, RankP is the first system that seamlessly integrates intuitive efficient processing algorithms to develop a systematic data retrieval framework RankP. To achieve this, we integrate the Rank formulation framework with existing ad-hoc ranking techniques, which have been successfully applied in various domains. This results in a novel framework that allows for efficient processing of large datasets while maintaining high accuracy.

**Integrating Rank formulation:** This process develops an interactive rank formulation framework.

**Query optimization:** This process proposes a cost-based optimization, which is an important aspect of efficient processing.

**Prove minification:** This process proposes a cost-based optimization, which is an important aspect of efficient processing.

**Figure 1.2 Framework RankP:** Rank formulation and processing for data retrieval.
• **Supporting non-monotonic ranking:** While the function learned from the formulation front-end can be arbitrarily complicated, rank processing techniques have been focusing on a limited set of ranking functions, *i.e.*, monotonic functions. This thesis thus develops a fresh perspective of abstracting query answering as an optimization problem, which enables to extend rank processing to any arbitrary function.

The remainder of this thesis is organized as follows. To prepare foundation, in Chapter 2, we first overview top-\(k\) query semantics and its cost model, and introduce the current state-of-art. Then, in Chapter 3, we discuss our cost-based probe minimization framework MPr\(\phi\) for expensive predicates, as an important and fundamental special case for general top-\(k\) optimization. In Chapter 4, we show how we generalize and leverage the insight of MPr\(\phi\) to enable cost-based optimization over general cost scenarios of arbitrary accesses. In Chapter 5, we discuss our rank formulation framework and propose a system integrating rank formulation front-end with rank processing algorithms. In Chapter 6, we discuss how we address a new challenge of supporting non-monotonic ranking functions. In Chapter 7, we present related works. Finally, in Chapter 8, we conclude the thesis and outline future tasks.
CHAPTER 2

Preliminaries

To establish the context of our discussion, this chapter describes the semantics and a cost model for top-k queries.

2.1 Query Semantics

A top-k query $Q = (\mathcal{F}, k)$, with scoring function $\mathcal{F}$ and retrieval size $k$, selects $k$ top objects ranked by $\mathcal{F}$, from database $\mathcal{D} = \{u_1, \ldots, u_n\}$. Each object $u$ has a predicate score $p_i[u]$ for every $p_i$ and an overall query score $\mathcal{F}(p_1, \ldots, p_m)[u] = \mathcal{F}(p_1[u], \ldots, p_m[u])$. Without loss of generality, we assume that all scores are in $[0 : 1]$. For now, as a standard assumption, $\mathcal{F}$ is monotonic, i.e., $\mathcal{F}(x_1, \ldots, x_m) \geq \mathcal{F}(y_1, \ldots, y_m)$ when $\forall i : x_i \geq y_i$. (Later, Chapter 5 will discuss how we support non-monotonic functions.)

As output, a top-k query returns a sorted list $\mathcal{K}$ of $k$ top objects (i.e., $|\mathcal{K}| = k$), along with and ranked by their overall $\mathcal{F}$ scores, such that $\mathcal{F}[u] > \mathcal{F}[v]$, $\forall u \in \mathcal{K}$ and $\forall v \notin \mathcal{K}$. Note that, to give deterministic semantics, we assume that there are no ties – otherwise, a deterministic “tie-breaker” function can be used to determine an order, e.g., by unique object IDs (e.g., hotel names).

Alternatively, top-k queries can be viewed as retrieving objects $u$ that score no less than some threshold $\theta_k$, i.e., $\mathcal{F}[u] \geq \theta_k$. Note this thresholding view will be semantically equivalent to top-k query view, when $\theta_k$ corresponds to the lowest score of top-k results: Though such $\theta_k$ is

\footnote{To be more rigorous, $\mathcal{F}(p_1, \ldots, p_m)[u]$ is in fact $(\mathcal{F} \circ \tilde{p})[u]$, where $\tilde{p} = (p_1, \ldots, p_m)$, i.e., a composition of $\mathcal{F}$ and predicates.}

\footnote{Such enforcement of certain tie breaker enables optimization to compare only “truly comparable” algorithms returning the same results.}
not known \textit{a priori} in practice, this semantically equivalent view helps conceptualize important insights, as we will revisit in Chapter 3.

As our running example, we will consider $Q_1$ (Example 1) for finding top-1 restaurant, \textit{i.e.}, $k = 1$. (For notational brevity, we will write predicates \textit{rating} and \textit{close} as $p_1$ and $p_2$ respectively.) For our illustration, let’s assume Dataset 1 (Figure 2.1) as our example restaurant “objects” (\textit{i.e.}, $u_1, u_2,$ and $u_3$) and their scores (which can only be known by accessing the Web sources). For instance, object $u_1$ scores $p_1[u_1] = 0.65$, $p_2[u_1] = 0.8$, and $\mathcal{F}[u_1] = \min(0.65, 0.8) = 0.65$. Overall, as a top-1 query, $Q_1$ will return an answer $\mathcal{K} = \{u_3: 0.7\}$, \textit{i.e.}, $u_3$ is the top-ranked object with score $\mathcal{F}[u_3]=0.7$.

2.2 Cost Model for Middleware Accesses

For ranked querying, a middleware algorithm will gather predicate scores by some supported accesses to sources: As Section 1 introduces, a source may support 1) \textit{sorted} access on predicate $p_i$, denoted $sa_i$; or 2) \textit{random} access on predicate $p_i$ for object $u_j$, denoted $ra_i(u_j)$. To illustrate, consider our example $Q_1$ over Dataset 1. Figure 2.1(b) illustrates the sorted accesses—For instance, dineme.com supports $sa_1$ (sorted access on $p_1$; Note $p_1=\textit{rating}$). Each $sa_1$ will return one next-ranked object in the order of $p_1$—\textit{i.e.}, $p_1[u_3]=.7$, $p_1[u_1]=.65$, and $p_1[u_2]=.6$. Alternatively, random access will directly return an object’s score on some predicate: For instance, superpages.com supports $ra_2(u_i)$ by returning the $p_2$ score (note $p_2=\textit{close}$) for $u_i$, \textit{e.g.}, $p_2[u_3] = 0.7$.

A middleware “algorithm” is thus a \textit{query plan} that uses (and schedules) such accesses for query answering. Different algorithms will perform different set of accesses to gather the scores needed, as we illustrate below:

\textbf{Example 3 (Performed Accesses):} To illustrate, consider an algorithm $\mathcal{M}_1$ performing the following accesses: $\mathcal{P}(\mathcal{M}_1) = \{sa_1, ra_2(u_1), sa_1, ra_2(u_2), sa_1, ra_2(u_3)\}$. Note we use $\mathcal{P}(\mathcal{M}_1)$ to denote the “performed accesses” by $\mathcal{M}_1$. With these accesses, $\mathcal{M}_1$ has gathered enough information to answer $Q_1$: In particular, it simply gathers the exact scores of every object for every predicate—The top-1 can then be identified by sorting objects by their $\mathcal{F}$ scores. Note, the same query can be answered by different algorithms with different sets of accesses, \textit{e.g.}, $\mathcal{P}(\mathcal{M}_2) = \{sa_1, sa_2, sa_1, sa_2, sa_1, sa_2\}$. \hfill $\blacksquare$
### Figure 2.1 Dataset 1

As a remark, 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 2.1(b), the first \( sa_1 \) not only evaluates \( p_1[u_3] = .7 \) but also *bounds* the “maximal-possible” score of \( p_1 \) for every “unseen” objects 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, by as accessing *deeper* into \( p_1 \)’s sorted list. In contrast, \( ra_i(u_j) \) will return the same \( p_i[u_j] \) every time—and thus it should not be repeated.

In the middleware context, each access incurs some cost, e.g., network communication or server computation. As such costs often dominate, our goal is to minimize the “total access cost,” which represents the total resource usage. To capture various scenarios, our cost model uses \( cs_i \) and \( cr_i \) to specify the unit cost of a sorted and random access respectively for predicate \( p_i \). The total access cost will then aggregate the costs of all accesses; i.e., let \( S_i \) and \( R_i \) be the number of sorted and random accesses respectively, for \( p_i \) performed by some algorithm \( M \), the total cost is

$$ C(M) = \sum_i S_i \cdot cs_i + R_i \cdot cr_i. \quad (2.1) $$

**Example 4 (Cost Model)**: To illustrate how our cost model works, continue Example 3. In an access scenario illustrated in Figure 1.1(a), where \( cs_1 = 32ms, cr_1 = 700ms, cs_2 = 344ms, \) and \( cr_2 = 1400ms, \) Algorithm \( M_1 \) performing 3 \( sa_1 \) and 3 \( ra_2 \) (i.e., \( S_1 = 3 \) and \( R_2 = 3 \)) incurs the total cost of 4296. Meanwhile, \( M_2 \) performing 3 \( sa_1 \) and 3 \( sa_2 \) incurs a smaller cost of 1128. However, observe that optimization is specific to the given cost scenario at run time: In another scenario like Figure 1.1(b), where \( cs_1 = cs_2 = 44ms \) and \( cr_1 = cr_2 = 0ms, \) Algorithm \( M_1 \) is more efficient than \( M_2. \)
<table>
<thead>
<tr>
<th>Sorted Access</th>
<th>Random Access</th>
</tr>
</thead>
<tbody>
<tr>
<td>cheap: $c_{s_i} = 1$</td>
<td>FA, TA, Quick-Combine</td>
</tr>
<tr>
<td>expensive: $c_{s_i} = h$</td>
<td>?</td>
</tr>
<tr>
<td>impossible: $c_{s_i} = \infty$</td>
<td>?</td>
</tr>
</tbody>
</table>

**Figure 2.2** Access scenarios and their proposed algorithms.

Note the total access cost, as a standard cost model used in top-$k$ works [19], reflects not only total resource usage, but also elapsed time as well, when accesses are performed sequentially. Thus, in general, our access minimization framework will naturally optimize for both. However, the two optimization goals can conflict, when sources can handle concurrent accesses (e.g., as Web sources typically do): While elapsed time benefits from high concurrency, unrestrained concurrent accesses will certainly abuse resources (e.g., causing the server to congest). To address the conflicting goals, we model concurrency as bounded and optimize within this "concurrency limit"—We will later show that such parallelization can simply build upon our access minimization framework as well.

### 2.3 Current State-of-the-art

For various middleware cost scenarios, many algorithms have been proposed—Figure 2.2 summarizes a “matrix” of access scenarios that have been studied, each characterized by how sources relatively support either type of access, e.g., cheap, expensive, or impossible. Fagin pioneered Algorithm FA [18, 45] for scenarios where random and sorted accesses are supported with uniform cost (the diagonal cells in Figure 2.2). [37, 19] then proposed TA (or equivalents) with a stronger sense of optimality. Meanwhile, some works [19, 22, 3] explored “non-uniform” scenarios: e.g., CA (when random access is expensive), NRA (when random access is impossible). Further, SR-Combine[3], Quick-Combine[21], and Stream-Combine [22] enhance the above base algorithms with some runtime optimization. However, their heuristics has limited applicability—e.g., it uses the partial derivative of scoring functions as an indicator, which may not applicable to all functions (e.g., min).
While these assorted algorithms have been proposed for supporting top-$k$ queries, as summarized by the "matrix" over various access scenarios (in Figure 2.2), none is generally applicable to a wide range of data retrieval scenarios:

First, the current matrix lacks \textit{generality}, in order to support a wide range of access scenarios. To begin with, the current matrix (Figure 2.2) has a missing cell, \textit{i.e.}, the (expensive, cheap) cell, which makes us wonder: When an application emerges in such a scenario, do we need to propose yet another algorithm to fill the cell? More significantly, even with all the cells filled, the matrix is still rather incomplete, since it assumes access costs are the same for all predicates, \textit{i.e.}, $\forall i : c_{s_i} = s$ and $c_{r_i} = r$. The current matrix thus fails to capture many real-life access scenarios, \textit{e.g.}, Figure 1.1, where access costs vary significantly across predicates.

Second, the current matrix lacks systematic \textit{adaptivity}, in order to account for runtime specifics for better performance. That is, existing algorithms only follow statistically designed behaviors and thus largely lack systematic runtime optimization, with at most limited heuristics. In contrast, as we will argue later, a systematic adaptation to runtime specifics can make a significant difference in performance.

Lastly, the current matrix lacks conceptual \textit{unification}. While these assorted algorithms, as designed for different scenarios, naturally behave differently, they seem to share some subtle similarity- \textit{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: Such 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.

In contrast, in this thesis, we will complement current state-of-the-art, by developing a general and adaptive optimization framework that conceptually unifies existing algorithms. Chapter 3 will study \textit{expensive predicate} scenarios (when sorted access is impossible) as an important and fundamental special case. In particular, we minimize random accesses in answering the query, by developing a cost-based optimization framework MPro. (Around the same time, references [5] and [20] studied similar scenarios, which we compare and contrast in Section 3.4.) Chapter 4 will leverage and generalize the insight of MPro for general top-$k$ optimization of minimizing costs of \textit{arbitrary} accesses (not only random accesses, but also sorted and potentially beyond).
Meanwhile, note also that, there have been works on supporting ranked queries over relational databases as well: Carey et al. [7, 8] presented optimization techniques for exploiting the limited cardinalities of ranked queries. References [17, 13] then proposed to exploit probabilistic distributions and histograms respectively, to process rank queries as equivalent Boolean selections. More recently, PREFER [28] used materialized “views” to evaluate preference queries efficiently, defined as linear sums of attribute values. We will later discuss how these works are relevant to our developments.
CHAPTER 3

Probe Minimization by Cost-based Schedule Optimization

As overviewed in Chapter 1, as a fundamental and important special case, this chapter studies expensive predicate scenarios. Such scenarios are fundamental as this problem of minimizing probes for expensive predicate can serve as a building block for general top-k optimization of minimizing the cost of arbitrary accesses. Meanwhile, such scenarios are important as they generally abstract important operations for ranked queries as follow:

- **External Predicates**: External sources (e.g., web source) used for predicate evaluation often have the limited capability of supporting only random accesses. To illustrate, in Q1, maps.yahoo.com can be used instead to evaluate close, which only supports random accesses of evaluating the distance of the given two locations.

- **User-defined Predicates**: User-defined functions are expensive because they are dynamically defined and thus require per-object evaluation. While they are commonly supported in Boolean database systems, we believe such functions will be even more important for ranked queries, as rank predicates are inherently imprecise and user (or application) specific (e.g., the notion of cheap depends on the budget of the very user).

- **Fuzzy join Predicates**: Join predicates are expensive, because they are inherently user-defined operations: In general, a join operation can dynamically associate any attributes from multiple tables (as in the Boolean context such as SQL), and in addition the associating function can be user-defined. Section 3.2.1 will discuss how our expensive predicate abstraction enables the unified framework for both selection and join predicates.

- **Post-filtering Predicates**: Ranking predicates can be turned into expensive ones, when combined with Boolean filtering conditions in relational database systems. To illustrate, suppose there exist index structures to access houses in the decreasing order of predicates,
e.g., large, cheap, and new. Now that user is interested only in houses in Chicago, the indices built on base table (of all houses) may no longer be efficient in retrieving high-scored Chicago houses, especially when Chicago houses are very few in the database. Section 3.3 will discuss how supporting post-filtering predicate enables a simple integration of ranking and filtering in relational database system.

Given predicates $p_1, \ldots, p_m$, these scenarios are characterized by some predicates $p_1, \ldots, p_q$ supporting only random accesses, while the rest $p_{q+1}, \ldots, p_m$ are search predicates supporting cheap sorted accesses (e.g., index lookup) as well. To stress this focus, we assume (without loss of generality) queries of the form $F(x, p_1, \ldots, p_q)$, with a search predicate $x$ and some expensive predicate $p_i$. Note that, multiple search predicates $p_{q+1}, \ldots, p_m$ can be merged into a single sorted stream $x$ by applying the well-known Fagin’s algorithm [18] (Section 2.3). In such scenarios, as expensive random accesses (or probes) dominate the cost, our goal is thus clear: to minimize probe cost. We discuss “cost-based” probe minimization in this chapter, which will later be generalized for arbitrary accesses. Section 3.1 presents our cost-based optimization framework, based on which we develop Algorithm MPro. Section 3.2 then discusses several extensions and scalability analysis of the basic algorithm. Section 3.3 demonstrates how our support for post-filtering ranking predicates enables a simple integration of ranking and Boolean filtering in with a relational DBMS. Section 3.4 reports our experimental evaluation.

3.1 Minimal Probing: Cost-based Schedule Optimization

Given a ranked query characterized by scoring function $F(x, p_1, \ldots, p_q)$ and retrieval size $k$, with a search predicate $x$ and probe predicates $p_1, \ldots, p_q$, our goal is to minimize probe cost in processing the query. In this section, we cast the problem as a cost-based optimization of identifying the optimal cost algorithm in the search space. Further, we propose Algorithm MPro (for minimal probing) which realizes the framework. Section 3.1.1 overviews our cost-optimization framework and identifies object and predicate scheduling as sub-problems, which correspond to “phases” of OSch and PSch of our algorithm MPro (Figure 3.2). Section 3.1.2 and 3.1.3 then tackle object and predicate scheduling respectively.
<table>
<thead>
<tr>
<th>OID</th>
<th>$x$</th>
<th>$p_c$</th>
<th>$p_l$</th>
<th>$\mathcal{F}(x, p_c, p_l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>0.90</td>
<td>0.85</td>
<td>0.75</td>
<td>0.75</td>
</tr>
<tr>
<td>b</td>
<td>0.80</td>
<td>0.78</td>
<td>0.90</td>
<td>0.78</td>
</tr>
<tr>
<td>c</td>
<td>0.70</td>
<td>0.75</td>
<td>0.20</td>
<td>0.20</td>
</tr>
<tr>
<td>d</td>
<td>0.60</td>
<td>0.90</td>
<td>0.90</td>
<td>0.60</td>
</tr>
<tr>
<td>e</td>
<td>0.30</td>
<td>0.70</td>
<td>0.80</td>
<td>0.50</td>
</tr>
</tbody>
</table>

Figure 3.1 Dataset 2 for query $\mathcal{F}(x, p_c, p_l) = \min(x, p_c, p_l)$.

3.1.1 Schedule Optimization: Overview

In this section, we develop top-$k$ query processing as a cost-based optimization and identify the focus of object and predicate scheduling as its sub-problems (which will be discussed by the following sections 3.1.2 and 3.1.3 respectively). To begin with, we focus on a general class of sequential algorithms, as any algorithm can be considered sequential by flattening out the operations performed. (Section 3.2.3 will present "simple extension" to parallel probing, while it is not the focus of this thesis.) A sequential algorithm executes probes one at a time and stops if no further probes are necessary. Each probe (or random access) evaluates a predicate $p$ for an object $u$, designated by $ra_p(u)$, to determine the score $p[u]$. Given a query with $q$ expensive predicates, for a database with $n$ objects, there are thus $n \cdot q$ possible probes (one for a pair of an object and a predicate). A sequential algorithm will execute these probes in some order (possibly terminate before exhausting all probes), forming a particular probe schedule. We will thus view any sequential algorithm as equivalent to a probe schedule $S$ (which the algorithm produces). That is, an optimal sequential algorithm must in effect determine the optimal probe schedule that requires the least probe cost. Our task of sequential-algorithm optimization is thus precisely a probe scheduling problem, i.e., finding the minimal-cost probe schedule.

Example 5 (Scheduling Space): To illustrate, consider a query $\mathcal{F}(x, p_c, p_l) = \min(x, p_c, p_l)$ over Dataset 2, there are 2 expensive predicates, $p_c$ and $p_l$ (e.g., cheap and large for finding an ideal house), and 5 objects ($a, \ldots, e$), and thus $(5 \cdot 2) = 10$ probes.

Different sequential algorithms will schedule these probes in different orders. The followings are two possible orders: $S_1: ra_p(a), ra_p(a), ra_p(b), ra_p(b), \ldots, ra_p(e), ra_p(e)$ and $S_2: ra_p(a), ra_p(a), ra_p(b), ra_p(b), ra_p(c), ra_p(c), ra_p(d), ra_p(d), ra_p(e), ra_p(e)$. All together, there are $(5 \cdot 2)! = 3,628,800$ possible schedules.
Depending on the particular databases, some schedules may terminate earlier before exhausting all 10 probes (as Example 8 later will illustrate). As a result, different algorithms, by using different probe schedules, will result in different costs. ■

As Example 5 shows, given a query, possible sequential algorithms or probe schedules are numerous in any practical settings, resulting in a large search space for optimization. In particular, given \( n \cdot q \) probes, the “space” of sequential algorithms, which we designate as \( \Omega \), consists of \( (n \cdot q)! \) different schedules (each of which sequences the probes in a different way), i.e., \(|\Omega| = (n \cdot q)!\). (Strictly speaking, the value is a tight lower bound of \(|\Omega|\), as an algorithm may terminate earlier with less than \( m \cdot q \) probes as Example 8 will illustrate.) Among all the algorithms \( S \) in \( \Omega \), our goal is to find the optimal one with the least probing cost, i.e.,

\[
\arg\min_{S \in \Omega} \mathcal{O}(S, \mathcal{F}, \mathcal{D}, k)
\]

(3.1)

Such optimization is clearly challenging as the space is prohibitively large, e.g., even for a toy setting illustrated in Example 5. (To reinforce, for a slightly larger setting of, say, 50 objects and 2 predicates, \(|\Omega| = (50 \cdot 2)! = 9.3e + 157\).

Conceptually, we view a probe schedule \( S \) as consisting of object and predicate scheduling: During execution, any sequential algorithm must schedule the next probe \( r_p(u) \) by selecting both an object \( u \) and a predicate \( p \), or object scheduling and predicate scheduling respectively. Example 6 illustrates such dual schedules.

**Example 6 (Object vs. Predicate Scheduling):** Continue Example 5. \( S_1 \) executes an object schedule \( \mathcal{O}_1 = (a, a, b, b, \ldots, c, c) \), in which each object appears twice, i.e., once for each predicate. Further, \( S_1 \) evaluates the predicates in the same order \((p_c, p_l)\) for each object, i.e., with a predicate schedule \( \mathcal{P}_1 = [a: (p_c, p_l), b: (p_c, p_l), \ldots, c: (p_c, p_l)] \). Note that \( \mathcal{P}_1 \) and \( \mathcal{O}_1 \) together determine the complete schedule, which we denote by \( S_1 = (\mathcal{P}_1, \mathcal{O}_1) \). Similarly, we can view \( S_2 = (\mathcal{P}_2, \mathcal{O}_2) \), where \( \mathcal{O}_2 \) is the same as \( \mathcal{O}_1 \), and \( \mathcal{P}_2 = [a: (p_c, p_l), b: (p_l, p_c), c: (p_c, p_l), d: (p_l, p_c), e: (p_c, p_l)]. \)

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

**Example 7 ($\Omega_\mathcal{P}$ and $\Omega_\mathcal{O}$):** Continue our running example with $n = 5$ objects and $q = 2$ predicates. First, consider $\Omega_\mathcal{P}$: Since each of the 5 objects has 2! predicate schedules, which are $(p_c, p_l)$ and $(p_l, p_c)$, there are combinatorially $(2!)^5 = 32$ predicate schedules, i.e., $|\Omega_\mathcal{P}| = 32$. Second, consider $\Omega_\mathcal{O}$: Any permutation of $O_1 = (a, a, b, b, \ldots, e, e)$ is also a valid object schedule, and therefore there are $|\Omega_\mathcal{O}| = \frac{(5 \cdot 2)!}{(2)!^5} = 113,400$. Together, we see that $|\Omega_\mathcal{P}| \cdot |\Omega_\mathcal{O}| = |\Omega|$ (as Example 5 computed).

Can separating object and predicate schedulings help decouple their combinatorial complexity? In fact, our approach exploits such separation to reduce the search space: As Section 3.1.2 will discuss, the crucial foundation of our approach is Necessary-Probe principle (Theorem 1) which enables to analytically determine the best object schedule, denoted $\mathcal{O}_\mathcal{P}$, *for any predicate schedule* $\mathcal{P}$ (Theorem 2). Therefore, to find the optimal complete schedule $S$ (in Equation 3.1), we can specifically focus on $S = (\mathcal{P}, \mathcal{O}_\mathcal{P})$, where each predicate schedule $\mathcal{P}$ pairs with its corresponding optimal object schedule $\mathcal{O}_\mathcal{P}$, instead of arbitrary $\mathcal{O}$. Our task is thus reduced to finding the optimal predicate schedule $\mathcal{P}$ which, together with its implied object schedule $\mathcal{O}_\mathcal{P}$, results in the least probing cost. Note that, our Algorithm MPro (Figure 3.2) also reflects this separation in design: PSch phase finds the optimal predicate schedule $\mathcal{P}$ and OSch phase identifies the optimal object schedule with respect to $\mathcal{P}$. (For notational simplicity, we use OSch both as a phase of MPro and as the object schedule $\mathcal{O}_\mathcal{P}$ it produces, i.e., $\mathcal{O}_\mathcal{P} = \text{OSch}(\mathcal{P})$.)

Formally, we thus transform our optimization problem into

$$\arg\min_{\mathcal{P} \in \Omega_\mathcal{P}} C((\mathcal{P}, \mathcal{O}_\mathcal{P} = \text{OSch}(\mathcal{P})), \mathcal{F}, \mathcal{D}, k)$$  \hspace{1cm} (3.2)

Note that the transformation from Equation (3.1) to Equation (3.2) significantly reduces the search space from $|\Omega| = (n \cdot q)!$ to $|\Omega| = |\Omega_\mathcal{P}| = (q!)^n$, e.g., for our toy example of $n = 5$ and $q = 2$: from 3,628,800 to 32; for $n = 50$ and $q = 2$: from $9.3e + 157$ to $1.1e + 15$.

Further, we must optimize predicate scheduling in the space of $\Omega_\mathcal{P}$. First, we develop the context-independence of predicate scheduling (Theorem 3), which states that the scheduling of
each object is independent of others. In other words, for our database with \( n \) objects, \( D = \{ u_1, \ldots, u_n \} \), our goal is to simply find the best “sub-schedule” \( \mathcal{P}^i \) for each object \( u_i \), and together these sub-schedules will form the complete best schedule, which we write as \( \mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n \).

The context-independence of predicate scheduling can be best observed by taking semantically equivalent “thresholding view” (Chapter 2): To isolate sub-scheduling from the context of other objects, we replace “context-sensitive” parameter \( k \) with “context-independent” parameter \( \theta_k \). (Without further complicating the notations, we abuse the notation of PC to use \( k \) and \( \theta_k \) interchangeably.) With this context-independent parameter \( \theta_k \), our goal is now to find the optimal sub-schedule \( \mathcal{P}^i \) independently from its own space \( \omega_{\mathcal{P}} \), i.e.,

\[
\arg\min_{\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n \in \omega_{\mathcal{P}}} \mathcal{C}(\mathcal{P}^1 \times \cdots \times \mathcal{P}^n, \mathcal{O}_{\mathcal{P}}, \mathcal{F}, \{ u_1, \ldots, u_n \}, k) = \left[ \arg\min_{\mathcal{P}^1 \in \omega_{\mathcal{P}}} \mathcal{C}(\mathcal{P}^1, \mathcal{F}, \{ u_1 \}, \theta_k) \right] \times \cdots \times \left[ \arg\min_{\mathcal{P}^n \in \omega_{\mathcal{P}}} \mathcal{C}(\mathcal{P}^n, \mathcal{F}, \{ u_n \}, \theta_k) \right] \quad (3.3)
\]

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

In summary, we have overviewed how we reduce the search space of probe scheduling \( \Omega \), by decoupling it into that of object and predicate schedulings, which the following sections will discuss in detail: Section 3.1.2 will discuss our provably optimal object scheduling, which analytically pairs predicate schedule \( \mathcal{P} \) with its corresponding optimal object schedule \( \mathcal{O}_{\mathcal{P}} \) and thus reduces the search space into that of predicate scheduling, i.e., \( |\Omega| = |\Omega_{\mathcal{P}}| = (q!)^n \). Section 3.1.3 will then follow to discuss how to optimize in the space of \( \Omega_{\mathcal{P}} \). The section will first develop context-independence to further reduce the search space into \( |\Omega| = (q!) \cdot n \), as we discussed. It will then develop a precise analytic cost model and argue to focus on finding a global schedule, i.e., the same \( \mathcal{P}^i \) for every object, which further reduces the search space to only \( q! \)– by removing the factor \( n \) necessary for per-object scheduling.

3.1.2 Object Scheduling: Analytic Principle

As overviewed in Section 3.1, this section first focuses on object scheduling problem: Section 3.1.2.1 develops Necessary-Probe principle (Theorem 1) which analytically determines the best
object schedule $O_P$ for the given $P$, which will also be implemented as an object scheduling phase $OSch$ of our probe-optimal algorithm $MPro$ in Section 3.1.2.2.

3.1.2.1 Necessary-Probe Principle

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

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

We stress that, with given predicate schedule $P$ determining the next predicate to evaluate for each object, determining necessary probes is essentially object scheduling of determining next objects to evaluate. According to the definition, $u$ will be further probed, only if it is absolutely required—thus $u$ must be a part of $O_P$: First, the probe is required independent of algorithms—any correct algorithm with predicate schedule $P$ must pay the probe. Second, the probe is required independent of the outcomes of other probes—it may be performed before or after others, but any particular order will not change the fact that the probe is required. While this notion of necessary probes seems intuitively appealing, first, how can we determine if a particular probe is absolutely required? Second, given there are many possible probes (at least one for each object at any time), which ones of them are actually necessary, and how to effectively find all the necessary probes? We next develop a simple principle for answering these questions.

Let's start with the first question: how to determine if a probe is necessary? To illustrate, consider finding the top-1 object for $F(x, p_c, p_l)$ with Dataset 2 (Figure 4.2), given schedule...
\[ \mathcal{P} = [a: (p_c, p_i), b: (p_c, p_i), c: (p_i, p_c), d: (p_i, p_c), e: (p_i, p_c)]. \]

Our starting point, before any probes, is the sorted output of search predicate \( x \) (Figure 4.2 sorts objects by their \( x \) scores). As \( \mathcal{P} \) specifies, we thus can choose to probe the next predicate \( p_c \) for any object \( a \) or \( b \), or predicate \( p_i \) for \( c, d, \) or \( e \). (Note that \( \mathcal{P} \) only specifies predicate orders, but not which object to probe next.) Let’s consider the top object \( a \) and determine if \( ra_{p_c}(a) \) is necessary. (We can similarly consider any other object.) The probe is actually necessary for answering the query, no matter how other probes turn out: Assume we can somehow determine the top-1 answer to be some object \( u \) without probing \( ra_{p_c}(a) \).

- **Suppose \( u \neq a \):** Note that \( \mathcal{F}[u] = \min(x[u], p_c[u], p_i[u]) \leq x[u] \), and \( x[u] \leq 0.8 \) for \( u \neq a \) (see Figure 4.2). Consequently, \( \mathcal{F}[u] \leq 0.8 \). However, without probing \( ra_{p_c}(a) \) and then \( ra_{p_i}(a) \), \( u \) may not be safely concluded as the top-1. For instance, suppose that the probes would return \( p_c[a] = 1.0 \) and \( p_i[a] = 1.0 \), then \( \mathcal{F}[a] = \min(0.9, 1.0, 1.0) = 0.9 \). That is, \( a \) instead of \( u \) should be the top-1, a contradiction.

- **Suppose \( u = a \):** In order to output \( a \) as the answer, we must have fully probed \( a \), including \( ra_{p_c}(a) \), to determine and return the query score.

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

\[
\mathcal{F}_T(t_1, \ldots, t_q)[u] = \begin{cases} 
t_i = t_i[u] & \text{if } t_i \in T \\
t_i = 1.0 & \text{otherwise.} 
\end{cases} 
\] (3.4)

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

- Suppose that \( r_{ap}(a) \) returns \( p_1[a] = 1 \) and thus \( \mathcal{F}[a] = 0.85 \). For finding the top-1 answer, we do not need to further evaluate \( b \) because \( \bar{\mathcal{F}}_{\{x\}}[b] = 0.8 < \mathcal{F}[a] = 0.85 \), and thus \( b \) cannot make to the top-1. That is, depending on \( p_1[a] = 1 \), we can answer the query without probing \( r_{ap}(b) \).

- Suppose that \( r_{ap}(a) \) returns \( p_1[a] = 0 \) and thus \( \mathcal{F}[a] = 0 \). Now \( b \) becomes the “current” top object (with the highest ceiling score \( \bar{\mathcal{F}}_{\{x\}}[b] = 0.8 \)). That is, depending on \( p_1[a] = 0 \), we can reason that \( r_{ap}(b) \) is necessary, similar to \( r_{ap}(a) \) above.

Further, we consider the second question: how to find all the necessary probes? Let \( u \) be any object in the database, and \( p \) be the next unevaluated predicate (on the predicate schedule \( \mathcal{P} \)) for \( u \). Potentially, any probe \( r_p(u) \) might be necessary. However, it turns out that at any point during query processing, there will be at most \( k \) probes that are necessary, for finding top-\( k \) answers. That is, we can generalize our analysis (see Theorem 1) to show that only those probes for objects that are currently ranked at the top-\( k \) in terms of their ceiling scores are necessary. Note that this conclusion enables an efficient way to “search” necessary probes: by ranking objects in the order of their current ceiling scores. (As Chapter 2 discussed, we assume that a deterministic tie-breaker will determine the order of ties.) For any object \( u \) in the top-\( k \) slots, its next probe \( r_p(u) \) for \( p \) specified by the given predicate schedule \( \mathcal{P} \) is necessary. Theorem 1 formally states this result.

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

\[ \blacksquare \]
**Proof:** We will show by contradiction that \( r_\alpha \) is necessary according to Definition 1. That is, we want to show that no algorithms can determine correct top- \( k \) answers without probing \( r_\alpha \), no matter what other probe results are. To contradict, suppose that there exists some algorithm \( \mathcal{A} \) to the contrary. Let \( \mathcal{K} \) be the top- \( k \) output of \( \mathcal{A} \). We show that \( \mathcal{K} \) can either be incomplete or incorrect-- and thus such \( \mathcal{A} \) does not exist.

- Suppose that \( u \notin \mathcal{K} \): Let \( \mathcal{P}(p) \) be predicates scheduled in prior to predicate \( p \) in schedule \( \mathcal{P} \). Since \( p \) is the next predicate for \( u \), it must have \( T_u = \mathcal{P}(p) \) as the evaluated predicates. There exist less than \( k \) objects \( v_1, \ldots, v_g \) \( (g < k) \) such that \( \mathcal{F}_{T_u}[u] < \mathcal{F}_{T_{v_i}}[v_i] \).

Suppose that \( \mathcal{A} \) returns answers \( \mathcal{K} = (a_1, \ldots, a_k) \). Since \( g < k \) (i.e., there are more \( a_j \) than \( v_i \)), there exists some \( a_j \) such that \( a_j \) is not among \( v_1, \ldots, v_g \). That is, for some \( T_{a_j} \) (as some subset of the schedule), \( u \) has a higher ceiling score: (Note that we assume no ties in score comparison; see Chapter 2.)

\[
\mathcal{F}_{T_u}[u] > \mathcal{F}_{T_{a_j}}[a_j]
\]  

We show that, if the unevaluated predicates of \( u \) are all perfectly scored, then \( \mathcal{K} \) will be incorrect, and thus \( \mathcal{A} \) does not always return correct answers. In this case, \( u \) has predicate scores (in addition to those already probed in \( T_u \)) \( \forall t \notin T_u : t[u] = 1.0 \) and thus \( \mathcal{F}[u] = \mathcal{F}_{T_u}[u] \). It thus follows that \( \mathcal{F}[u] > \mathcal{F}_{T_{a_j}}[a_j] \). Since \( \mathcal{F} \) is monotonic and \( \mathcal{F}_{T_{a_j}}[a_j] \) upper bounds \( \mathcal{F}[a_j] \), it follows that \( \mathcal{F}[u] > \mathcal{F}[a_j] \). That is, \( u \) outperforms some top- \( k \) answer, and thus \( \mathcal{K} \) is incorrect.

- Suppose that \( u \in \mathcal{K} \): Since \( r_\alpha \) is not performed, \( \mathcal{A} \) cannot determine the predicate score \( p[u] \) and thus neither the query score \( \mathcal{F}[u] \). Consequently, \( \mathcal{K} \) does not return all the query scores as required; it is thus incomplete.

\( \blacksquare \)

Theorem 1 provides an “operational” definition to actually determine if a given probe is necessary as well as to effectively search those probes. As the notion of necessary probes (Definition 1) isolates probes absolutely required by any correct algorithm by definition, we can immediately conclude that an \( \mathcal{A} \) is probe-optimal (Section 2.2), if it only performs necessary probes determined as in Theorem 1. Lemma 1 formally states this intuition.
Lemma 1 (Probe-Optimal Algorithms): Consider a ranked query with scoring function $\mathcal{F}$ and retrieval size $k$. Given a predicate schedule $\mathcal{P}$, an algorithm $\mathcal{A}$ for processing the query is probe-optimal if $\mathcal{A}$ performs only the necessary probes as Theorem 1 specifies. ■

Proof: Since every necessary probe (as Theorem 1 specifies) is required by any (correct) algorithms, according to Definition 1, no algorithms can save such a probe. Suppose we denote the probe cost of algorithm $\mathcal{A}$ by $C(\mathcal{A}) = \sum_{i=1}^{n} m_i C_i$ when $C_i$ is the per-probe cost for $p_i$ and $\mathcal{A}$ performs $m_i$ probes for $p_i$. Since every $m_i$ includes only necessary probes, $m_i$ is minimal, and thus $\mathcal{A}$ is probe-optimal. ■

Our goal is thus to develop such an optimal algorithm that schedules only objects that are absolutely necessary to evaluate, which thus analytically determines $\mathcal{O}_P$ with respect to predicate schedule $\mathcal{P}$. However, note such algorithm is not necessarily unique, as at any point, there can be multiple necessary probes to choose from; However, any such object schedule will be “equally” optimal. Section 3.1.2.2 next develops one such optimal object scheduling algorithm $\text{OSch}$.

3.1.2.2 Algorithm MPro

We next develop Necessary-Probe principle into a probe-optimal object scheduling algorithm $\text{OSch}$ which identifies the optimal object schedule $\mathcal{O}_P$ with respect to predicate schedule $\mathcal{P}$. As Figure 3.2 illustrates, $\text{OSch}$ phase is the crucial foundation of Algorithm MPro, together with $\text{PSch}$ phase which identifies the optimal predicate schedule $\mathcal{P}$ for $\text{OSch}$. We first focus on $\text{OSch}$ phase in this section; Algorithm MPro will later be completed in Section 3.1.3 by discussing $\text{PSch}$ phase.

Given predicate schedule $\mathcal{P}$ (identified from $\text{PSch}$ phase), $\text{OSch}$ phase optimizes probe cost by ensuring that every probe performed is indeed necessary (for finding $k$ top answers). Essentially, during query processing, $\text{OSch}$ phase keeps “searching” for a necessary probe to perform next. Progressing with more probes, eventually $\text{OSch}$ phase will have performed all (and only) the necessary probes, at which point the top-$k$ answers will “surface.” Note that Theorem 1 identifies a probe $\alpha_p(u)$ as necessary if $u$ is among the current top-$k$ in terms of ceiling scores. Thus, a “search mechanism” for finding necessary probes can return top ceiling-scored objects.
Algorithm MPro($\mathcal{F}, k, \mathcal{D}$): Minimal-probing algorithm
Input:
- $\mathcal{F}(x, p_1, \ldots, p_k)$: scoring function // with expensive predicates $p_1, \ldots, p_k$.
- $k$: retrieval size, i.e., to return top-$k$ answers.
- $\mathcal{D}$: input database // assume selection predicates over single relation for simplicity.
Output: $K$, the top-$k$ answers with respect to $\mathcal{F}$.
Procedure:
1. $\mathcal{P}$Sch: identifies predicate schedule $\mathcal{P}$ for $\mathcal{O}$Sch.
   // can precede or interleave with object scheduling.
2. $\mathcal{O}$Sch: identifies the optimal object schedule $\mathcal{O}^*$ with respect to $\mathcal{P}$ from $\mathcal{P}$Sch.

(2.1) Queue Initialization:
// search $\mathcal{X}$ over $\mathcal{D}$ to prepare sorted output queue $\mathcal{X}$.
- $\mathcal{K} \leftarrow \emptyset$; $\mathcal{Q} \leftarrow \emptyset$ // $\mathcal{K}$: output; $\mathcal{Q}$: ceiling queue to prioritize by ceiling score.
- initialize $\mathcal{Q}$ to buffer objects prioritized by their ceiling scores from $\mathcal{X}$.
- this “full” initialization is only conceptual; $\mathcal{X}(\text{top})$ can be on demand.
- while $\mathcal{X}$ is not empty:
  - $u \leftarrow \mathcal{X}(\text{top})$ // pop next top object out of $\mathcal{X}$.
  - $\mathcal{Q}$.insert($u$); $u$ceiling $\leftarrow \mathcal{F}_{T_u}[u]$ // initialize ceiling score with $x$.
  - $\mathcal{Q}$.insert($u$); $u$.ceiling $\leftarrow \mathcal{F}_{T_u}[u]$ // insert $u$ into $\mathcal{Q}$ prioritized by its ceiling score.

(2.2) Necessary Probing:
// set up the stop condition $\mathcal{S}\mathcal{C}$ for determining whether to stop probing.
- $\mathcal{S}\mathcal{C} \leftarrow |\mathcal{K}| \geq k$; i.e., at least $k$ completely evaluated objects seen on the top.
- while ($\mathcal{S}\mathcal{C} = 0$): // keep performing necessary probes until $\mathcal{S}\mathcal{C}$ becomes true.
  - $u \leftarrow \mathcal{Q}$.top() // the current (top) object with the highest ceiling score.
  - if $u$ is completely evaluated:
    - $u$.score $\leftarrow u$.ceiling; append $u$ to $\mathcal{K}$ // add $u$ to the top-$k$ output.
    - else: // $u$ must be probed further.
    - $p \leftarrow$ next unevaluated predicate of $u$ on schedule $\mathcal{P}$
    - $\mathcal{P}(u) \leftarrow \mathcal{F}_{T_u}[u]$ // perform $\mathcal{P}(u)$ must be necessary by Theorem 1.
    - $\mathcal{Q}$.insert($u$); $u$ceiling $\leftarrow \mathcal{F}_{T_u}[u]$ // update the ceiling score of $u$, as $\mathcal{P}(u)$ is just obtained.

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

Figure 3.2 Algorithm MPro

When requested — i.e., a priority queue [16] that buffers objects using their ceiling scores as priorities.

We thus design $\mathcal{O}$Sch phase to operate on such a queue, called ceiling queue and denoted $\mathcal{Q}$, of objects prioritized by their ceiling scores. As Figure 3.2 shows, $\mathcal{O}$Sch phase mainly consists of two steps: First, in the queue initialization step, starting with the sorted output stream $\mathcal{X}$ of search predicate $x$, $\mathcal{Q}$ is initialized based on initial ceiling scores $\mathcal{F}(x)[\cdot]$ with $x$ being the only evaluated predicate (see Eq. 1). Note that, although for simplicity our discussion assumes that $\mathcal{Q}$ is fully initialized (by drawing every object from $\mathcal{X}$), this initialization is only conceptual: It is important to note that $\mathcal{F}(x)[\cdot]$ will induce the same order in $\mathcal{Q}$ as the $\mathcal{X}$ stream, i.e., if $x[u] \leq x[v]$, then $\mathcal{F}(x)[u] \leq \mathcal{F}(x)[v]$, since $\mathcal{F}$ is monotonic. Thus $\mathcal{X}$ can be accessed incrementally to move objects into $\mathcal{Q}$ when more are needed for further probing. (It is entirely possible that some objects will not be accessed from $\mathcal{X}$ at all.)

Second, in the necessary probing step, $\mathcal{O}$Sch phase keeps on requesting from $\mathcal{Q}$ the top-priority object $u$, which has the highest ceiling score. Let $u$ be completely evaluated if $\mathcal{F}_{T_u}[u]=$
\( \mathcal{F}[u] \), i.e., either \( u \) is fully probed or \( \overline{\mathcal{F}}_{T_u}[u] \) has already reached minimal-possible score 0. Thus, if \( u \) is completely evaluated when it surfaces the top, \( u \) must be among the top-\( k \) answers, because its final score is higher than the ceiling scores of objects still in \( Q \). That is, \( u \) has “surfaced” to the top-\( k \) answers, which will be moved to an output queue, denoted \( K \) in Figure 3.2. Otherwise, if \( u \) is incomplete with the next unvisited predicate \( p \), according to Theorem 1, \( ra_p(u) \) is necessary. Thus OSch phase will perform this probe, update the ceiling score of \( u \), and insert it back to \( Q \) by the new score.

Incrementally, more objects will complete and surface to \( K \), and OSch phase will eventually stop when there are \( k \) such objects (which will be the top-\( k \) answers). As Figure 3.2 shows, OSch phase checks this stop condition, designated SC, to halt further probing. It is interesting to observe the “dual” interpretations of SC: On one hand, SC tells that there are already \( k \) answers in \( K \), and thus no more probes are necessary. On the other hand, when SC holds, it follows from Theorem 1 that no more probes can be necessary, and thus the top-\( k \) answers must have fully surfaced, which is indeed the case. (We discuss in Section 3.2.2 how the stop condition can be “customized” for approximate queries.)

Figure 3.3 illustrates how OSch phase actually works for our example of finding the top 2 object when \( \mathcal{F} = \min(x, p_c, p_l) \) and \( P = [a:(p_c, p_l), b:(p_c, p_l), c:(p_l, p_c), d:(p_l, p_c), e:(p_l, p_c)] \) over Dataset 2 (Figure 4.2). While we show the ceiling queue \( Q \) as a sorted list, full sorting is not necessary for a priority queue. After initialized from the sorted output of \( x \), we simply keep on probing the top incomplete object in \( Q \), resulting in the probes \( ra_{p_c}(a), ra_{p_c}(a), ra_{p_c}(b), \) and \( ra_{p_l}(b) \). Each probe will update the ceiling score of the object, and thus changing its priority in ceiling queue. Note that Figure 3.3 marks object with an underline (e.g., \( a:0.75 \)) when it is completely evaluated, at which point its ceiling score is actually the final score. Finally, we can
stop when $k = 2$ objects (in this case, $a$ and $b$ together) have completed and surfaced to the top.

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

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

### 3.1.3 Predicate Scheduling: Dynamic Optimization

In Section 3.1.2, we discussed our provably optimal object scheduling, which analytically determines the optimal object schedule $O_P$ with respect to the given predicate schedule $P$. This section completes the picture by studying predicate scheduling problem and proposing effective dynamic predicate scheduling schemes that identify the best predicate schedule for $O\text{Sch}$ phase.

Our optimization task now is thus to identify the best schedule $P$ as Equation 3.2 indicates. Recall that, given probe predicates $p_1, \ldots, p_q$, there are $q!$ possible predicate ordering (each as a permutation of $p_i$) for each object; for a database with $n$ object, i.e., $D = \{u_1, \ldots, u_n\}$, there are thus $(q!)^n$ predicate schedules. Different predicate schedules incur different probe cost in $O\text{Sch}$ phase, as Example 8 below shows.
<table>
<thead>
<tr>
<th>OID</th>
<th>$x$</th>
<th>$p_c$</th>
<th>$p_l$</th>
<th>$F(x, p_c, p_l)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a$</td>
<td>0.8</td>
<td>0.9</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>$b$</td>
<td>0.7</td>
<td>0.8</td>
<td>0.2</td>
<td>0.2</td>
</tr>
<tr>
<td>$c$</td>
<td>0.6</td>
<td>0.6</td>
<td>0.3</td>
<td>0.3</td>
</tr>
</tbody>
</table>

**Figure 3.4** Dataset 3.

**Example 8:** Consider $F = \min(x, p_c, p_l)$, using Dataset 3 (Figure 3.4). As Section 3.1.1 explained, for $n = 3$ (objects) and $q = 2$ (expensive predicates), there are $(2!)^3 = 8$ predicate schedules. To illustrate, consider $\mathcal{P}_1 = [a; (p_c, p_l), b; (p_c, p_l), c; (p_l, p_c)]$ and $\mathcal{P}_2 = [a; (p_l, p_c), b; (p_l, p_c), c; (p_l, p_c)]$. To find top-1, when given $\mathcal{P}_1$ and $\mathcal{P}_2$, OSch phase will perform 6 and 4 probes respectively (as shown below); $\mathcal{P}_2$ is better, with 33% less probes.

$\mathcal{P}_1$: $r_{a_p}(a), r_{a_p}(a), r_{a_p}(b), r_{a_p}(b), r_{a_p}(c), r_{a_p}(c)$

$\mathcal{P}_2$: $r_{a_p}(a), r_{a_p}(b), r_{a_p}(c), r_{a_p}(c)$

---

**3.1.3.1 Compositional Scheduling: Context Independence**

Toward the goal of identifying the mammal-cost predicate schedule, we first develop the context-independence of predicate scheduling. To illustrate, observe predicate schedule $\mathcal{P}_1$ in Example 8 which consists of sub-schedules $\mathcal{P}_1^1 = (p_c, p_l)$, $\mathcal{P}_1^2 = (p_c, p_l)$, and $\mathcal{P}_1^3 = (p_l, p_c)$ for object $a$, $b$, and $c$ respectively. Our first question is: does the best predicate schedule of an object, say $a$, depends on those of others, say $b$ and $c$? The following theorem of context independence addresses this question.

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

Theorem 3 states that the probe cost of an object depends only on its own schedule and will not be affected by others. For instance, consider $\mathcal{P}_1$ and $\mathcal{P}_2$ in Example 8: Since object $c$ has the same schedule in both, i.e., $\mathcal{P}_1^3 = \mathcal{P}_2^3 = (p_l, p_c)$, OSch phase perform indeed performs exactly the same probes on $c$, i.e., $r_{a_p}(c)$ and $r_{a_p}(c)$, in either $\mathcal{P}_1$ or $\mathcal{P}_2$ regardless of the different schedules of $a$ and $b$. 

27
This result follows directly from Lemma 2, which states that the probe cost of OSch on any object \( u_i \) depends only on its own sub-schedule \( P^i \) and the lowest top-\( k \) score \( \theta_k \). Intuitively, OSch performs probes on each object \( u_i \) to determine either of the followings:

1. its final score \( F[u_i] \), if \( u_i \) is a part of top-\( k \) answers; or

2. its disqualification from top-\( k \) answers, i.e., \( F_{T_{u_i}}[u_i] < \theta_k \).

To illustrate, consider predicate schedule \( P_2 \) in Example 8: the top-1 answer \( c \) is probed completely, while \( a \) and \( b \) are partially probed just enough to show their disqualification from top-\( k \) results, e.g., ceiling scores upper-bound scores are less than \( F_{T_a}[a] < \theta_k = 0.3 \). More formally, Lemma 2 states that each object \( u_i \) will be either completely evaluated, or partially up to \( j \)-prefix of its predicate schedule \( P^i \), denoted \( P^i(j) \), such that \( F_{P^i(j)}[u_i] < \theta_k \). As \( \theta_k \) is determined per query and thus independent from schedules of other objects, we can conclude that each sub-schedule impacts only the cost of its corresponding object.

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

\[
\begin{cases} 
  j \text{ is the smallest in } [1 : q] \text{ s.t. } F_{\{z, p_1, \ldots, p_j\}}[u_i] < \theta_k & \text{if such } j \text{ exists} \\
  j = q & \text{otherwise}
\end{cases}
\]

**Proof:** There are only two situations: either \( F[u_i] < \theta_k \) or \( F[u_i] \geq \theta_k \), which we consider in turn. Suppose that the top-\( k \) answers are \( v_1, \ldots, v_k \).

- \( F[u_i] < \theta_k \): Suppose that \( u_i \) has been evaluated some \( j \)-prefix, \( j < q \), of \( P^i \), i.e., \( T_{u_i} = \{ x, p_1, \ldots, p_j \} \). Note that our assumption of deterministic semantics (Chapter 2) enforces no ties in score comparison, thus \( F_{T_{u_i}}[u_i] < \theta_k \) or \( F_{T_{u_i}}[u_i] > \theta_k \) (or otherwise \( F_{T_{u_i}}[u_i] \) ties with the score of \( v_k \)).

- If \( F_{T_{u_i}}[u_i] > \theta_k \), OSch phase will further evaluate \( u_i \): Suppose not. In the end of Osch phase, \( v_1, \ldots, v_k \) will occupy the top-\( k \) slots of ceiling queue, with complete final scores
However, since $F_{T[u]}[u_i] > \theta_k$, $u_i$ must be also on the top-$k$ of ceiling queue, a contradiction.

- If $F_{T[u]}[u_i] < \theta_k$. OSch phase will stop evaluating $u_i$ further since $F_{T[u]}[u_i]$ will not be among the top-$k$ in the ceiling queue $Q$. At least, for each $v_i$, $F_{T[v]}[v_i] \geq F[v_i] \geq \theta_k > F_{T[u]}[u_i]$.

- $F[u_i] \geq \theta_k$: Since $u_i$ has a score higher than $\theta_k$ (and since we assume deterministic semantics; Chapter 2), $u_i$ must be one of the top-$k$ answers, and thus it must be fully evaluated. In this case, $\forall j \leq q$, $F_{\{x,p_{1},...,p_{j}\}}[u_i] \geq F[u_i] \geq \theta_k$, and thus $u_i$ indeed cannot stop earlier.

Putting together, we conclude that OSch phase will evaluate $u_i$ up to exactly the smallest $j$-prefix such that the latter condition starts to hold, or otherwise the entire $P^i$.

Such context independence (Theorem 3) enables a compositional approach (rather than a combinatorial one) over the predicate scheduling problem. Recall that the goal of predicate scheduling is to construct the overall schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n$ that minimizes $C(\cdot)$ over $\mathcal{D} = \{u_1, \ldots, u_n\}$. In fact, the cost $C(\cdot)$ to minimize is the sum of the per-object cost of each $u_i$. Since the per-probe cost of each object is independent from the schedules of others (as Lemma 2 states), the construction of $\mathcal{P}$ is a simple modular composition: $\mathcal{P}$ can be constructed by independently constructing each sub-schedule $\mathcal{P}^i$ for object $u_i$, as Equation (3.3) expresses (and as repeated below). As Section 3.1.1 explained, this compositional view reduces the search space to $(q!) \cdot n$.

$$\arg\min_{\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n \in \omega^p \times \cdots \times \omega^n} C((\mathcal{P}, \text{OSch}(\mathcal{P})), \mathcal{F}, \{u_1, \ldots, u_n\}, k)$$

$$= \arg\min_{\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n \in \omega^p \times \cdots \times \omega^n} C(\mathcal{P}^1, \mathcal{F}, \{u_1\}, \theta_k) + \cdots + C(\mathcal{P}^n, \mathcal{F}, \{u_n\}, \theta_k)$$

$$= [\arg\min_{\mathcal{P}^1 \in \omega^p} C(\mathcal{P}^1, \mathcal{F}, \{u_1\}, \theta_k)] \cdots [\arg\min_{\mathcal{P}^n \in \omega^n} C(\mathcal{P}^n, \mathcal{F}, \{u_n\}, \theta_k)]$$

3.1.3.2 Global Scheduling

To enable cost-based search for the optimal predicate schedule $\mathcal{P}$, this section develops an analytic cost model, based on which we argue to focus on global predicate scheduling. We first develop a cost model for per-object cost, to identify the optimal sub-schedule $\mathcal{P}^i$ for each object $u_i$, which in turn identifies the complete best schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^q$: Given $\mathcal{P}^i = (p_{i1}, \ldots, p_{iq})$, OSch phase will probe each $p_{ij}$ in turn for $u_i$. Lemma 2 asserts that OSch
phase will perform probes in the shortest $j$-prefix such that $\mathcal{F}_{\mathcal{P}^i(j)}[u_i] < \theta_k$, or otherwise the entire $\mathcal{P}^i$. In other words, predicate $p_r$ will be evaluated (and thus incurs its per-probe cost $C_r$) only when $\mathcal{F}_{\mathcal{P}^i([r-1])}[u_i] \geq \theta_k$. As we must construct $\mathcal{P}^i$ before actually executing the probe, we compare the expected cost (without knowing predicate scores) in terms of the probabilities that the above inequality will hold, denoted as $Pr(\mathcal{F}_{\mathcal{P}^i([r-1])}[u_i])$. Our task is thus to identify $\mathcal{P}^i$ that minimizes:

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

(3.6)

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

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

Now the question is: In terms of benefit, can per-object scheduling be significantly “more optimal” than global scheduling to merit the overhead? As Equation 3.6 suggests, an optimal schedule depends on the “oracle” of some values unknown during optimization, e.g., $\theta_k$ and $\mathcal{F}_{\mathcal{P}^i([r-1])}[u_i]$. From an “information-theoretic” viewpoint, per-object scheduling will work better only if object-specific prediction of these values results in better oracles. (In extreme, scheduling will be truly optimal if both parameters can be exactly predicted without executing any probe, which is infeasible in practice.) Thus, to evaluate benefit, we observe whether the prediction of $\theta_k$ and $\mathcal{F}_{\mathcal{P}^i([r-1])}[u_i]$ benefit from object-specific information on $u_i$: First, the prediction of $\theta_k$ cannot benefit, as the parameter is determined per-query and thus not specific to each object by nature. Though the prediction can benefit from per-object invocation by leveraging more up-
to-date score information, it does not argue for per-object scheduling as global scheduling can similarly benefit from multiple invocations. Second, the prediction of $\mathcal{F}_{\mathcal{P}^g}[u_i]$ can benefit only if its already known scores $p[u_i]$, i.e., $p \in \mathcal{P}^g(r - 2)$, can predict unknown ones. Such prediction thus requires (1) pre-constructed information to correlate known scores to unknown ones, and (2) an incremental probe-by-probe invocation of the scheduler to take advantage of known scores in decision. However, it is often practically impossible to construct such object-specific correlation information, and also per-probe invocation potentially increases the scheduler overhead up to $nq$-fold. Because of these difficulties, while there have been several per-object schemes, their overhead rarely pays off, as our experiments empirically confirm (Section 3.4).

We thus decide to focus on finding a global schedule $\mathcal{P}^g$, which minimizes the cost across all objects in general. As we no longer discriminate objects in scheduling, our task is rephrased for a general object $u$ in the database as follows, which reduces the search space to $n!$.

$$E[C(\mathcal{P}^g, \mathcal{F}, \{u_1, \ldots, u_n\}, \theta_k)] = n \cdot \sum_{r=1}^{q} Pr(\mathcal{F}_{\mathcal{P}^g}[u] \geq \theta_k) \cdot C_r \tag{3.7}$$

Observe from Equation 3.7 that $C(\mathcal{P}, \text{Sch}(\mathcal{P})), \mathcal{F}, \mathcal{D}, k)$ can be exactly computed if the number of object $u \in \mathcal{D}$ that satisfies $\mathcal{F}_{\mathcal{P}^g}[u] \geq \theta_k$ is known. We thus define the aggregate selectivity $S^\theta_T(T)$ for a set of predicates $T$ as the ratio of database objects $u$ that “pass” $\mathcal{F}_T[u] \geq \theta$ (and thus will continue to be probed beyond $T$). (This selectivity notion, unlike its Boolean-predicate counterpart, depends on the aggregate “filtering” effect of all the predicates evaluated.) With the aggregate selectivity notion, we can formally state the exact probe cost of global schedule $\mathcal{P}^g$ as:

$$C(\mathcal{P}^g, \text{Sch}(\mathcal{P}^g)), \mathcal{F}, \mathcal{D}, k) = n \cdot \sum_{r=1}^{q} S^\theta_{\mathcal{P}^g}(\mathcal{P}^g(r - 1)) \cdot C_r.$$  

Our goal now is thus to identify the minimal-cost global schedule, among $n!$ possible ones. Toward the goal, we discuss how to estimate aggregate selectivity: Although pre-constructed statistics are often used in Boolean context, such requirement is unlikely to be realistic in our context, because predicates are either “dynamic” or “external”. Our scheduler can either estimate the selectivity by gathering statistics dynamically, or exclude the selectivity in scheduling decision.
selectivity by sampling online if feasible, or alternatively by exploiting other dynamic information, as follow:

**MPro with online sampling:** One way to gather statistics for estimating selectivities is to perform online sampling: At optimization time (i.e., before actual execution), the scheduler will sample a small number of objects and perform complete probing for their scores. We call such a scheme MPro-Sampling as they gather samples online that work as a small, representative database for predicate scheduling. While such sampling may add “unnecessary” probes, finding a good schedule can well justify this overhead (Section 3.4). In fact, some of the probes will later be necessary anyway (MPro can easily reuse those sampling probes).

Using the samples, we can estimate the selectivities with respect to the top-\(k\) threshold \(\theta_k\). The uniform sampling will select some \(k'\) top-\(k\) objects proportional to the sample size \(s\), i.e., \(k' = \lceil k \cdot \frac{1}{s} \rceil\). That is, the sampling transforms a top-\(k\) query on the database into a top-\(k'\) query on the samples. Thus \(\theta_k\) can be estimated as the lowest \(F\) score of the top-\(k'\) sampled objects. Aggregate selectivity \(F_{p_0(r-1)}[u] \geq \theta_k\) can then computed out of samples, by counting the ratio of sample objects satisfying \(F_{p_0(r-1)}[u] \geq \theta_k\).

To illustrate, suppose sampling results in the samples in Figure 3.4 for \(F = min(x, p_c, p_t)\). Assume that \(k' = 1\) for the sample size, and thus \(\theta_k = 0.3\) (the top-1 \(F\) score); let the relative costs \(C_{p_c} = 1\) and \(C_{p_t} = 3\). Our scheduler then compares the costs of potential schedules \(P_1^g = (p_c, p_t)\) and \(P_2^g = (p_t, p_c)\). To begin with, we compute the cost of \(P_1^g\), which is \(F_{\{z\}} \cdot C_{p_c} + F_{\{z, p_c\}} \cdot C_{p_t}\) according to Equation 3.7. Since all sampled objects satisfy \(F_{\{z\}} \geq 0.3\) and \(F_{\{z, p_c\}} \geq 0.3\), it follows that \(S_{p_0, z}^g(\{x\}) = S_{p_0, z}^g(\{x, p_c\}) = \frac{3}{2} = 1\). The cost of \(P_1^g\) is thus \(1 \cdot 1 + 1 \cdot 3 = 4\). Similarly, \(S_{p_0, z}^g(\{x, p_t\}) = \frac{1}{3}\) and thus the cost of \(P_2^g\) can be computed as \(1 \cdot 3 + 0.3 \cdot 1 = 3.3\). Consequently, the scheduler will select \(P_2^g\) over \(P_1^g\) (resulting in \(P_2 = (p_t, p_c)\) as in Example 8).

**Adaptive MPro:** Alternatively, when pre-execution sampling is not feasible, we can wait to activate the scheduler until OSch performs some necessary probes so that predicate scheduling can “adapt” according to the probes performed. We call such a scheme MPro-Adaptive, which consists of the following two phases: First, until accruing enough statistics to determine a desirable predicate scheduling, OSch phase keeps on performing a necessary probe on top-priority object \(u\). However, as predicate schedule has not been determined, we randomly decide
the next predicate \( p \) to probe for \( u \). As \( \rho[u] \) is evaluated from such probe, we add its score to the list \( \text{Scores}_p \). Second, once we acquire enough statistics (controlled by user-defined parameter as we will discuss in Section 3.4), we estimate selectivities from the \( \text{Scores} \) lists collected: In essence, we build samples (as in Figure 3.4) by corresponding each \( \text{Score}_p \) to a column of \( p \) scores in the table. However, note the difference from online sampling that, each list now evaluates different sets of objects. We thus arbitrary associate scores into objects, e.g., \( i \)th item of every \( \text{Score} \) list together form a virtual sample object \( s_i \), and identify the optimal predicate schedule \( \mathcal{P}^g \) over samples, just as demonstrated for online sampling. (The performance implication of building samples out of necessary probes will be further discussed in Section 3.4.) With \( \mathcal{P}^g \) identified, \text{OSch} phase can resume with respect to the schedule.

Alternatively, we can schedule greedily. In some scenarios, external parameters can “hint” the scheduler. For instance, when scoring function is weighted average, different weights indicate different impacts of each predicate toward the overall score. In particular, scheduler can take advantage of such hints to schedule the predicate with high impact but low per-probe cost first. Meanwhile, when such external information is not available, e.g., \( \mathcal{F} = \min \), this scheme degenerates to a simplistic scheduling in the increasing order of per-probe cost. We call such a scheme \text{MPr}_\text{EO} (for external parameters), which does not involve exponential scheduling.

These schemes will be empirically evaluated in Section 3.4, against the optimal scheduling and other existing schemes – our study shows they closely approximate the optimal scheduling with negligible net overhead.

### 3.2 Extensions and Scalability

We next discuss how probe-optimal algorithm \text{MPr} can be extended for several useful scenarios. As their full developments require extensive future works, this section only briefly discusses intuition: First, while we assume selection predicates so far, Section 3.2.1 generalizes to handling joins as well. Second, Section 3.2.2 extends \text{MPr} for approximate queries to allow trade-off between efficiency and accuracy. Third, Section 3.2.3 shows that \text{MPr} can be easily parallelized to exploit available resources with linear speedup. Finally, Section 3.2.4 analytically develops the scalability of \text{MPr}, showing that its cost growth is sub-linear in database size.
3.2.1 Fuzzy Joins

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

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

To illustrate this conceptual unification, consider the following query, which involves two relations house and park.

\[
\begin{align*}
\text{select } & h.\text{id}, s.\text{name} \text{ from } \text{house } h, \text{ park } s \\
\text{order by } & \min(\text{new}(h.\text{age}), \text{rating}(s.\text{name}), \text{close}(h.\text{zip}, s.\text{zip})) \\
\text{stop after } & 5
\end{align*}
\]  
\text{(Query Q3)}

The query uses a join predicate \( p_j = \text{close}(h.\text{zip}, s.\text{zip}) \) over pairs of \( h \) from house and \( c \) from park. For instance, suppose the relations, as sets of objects, are \( \text{house} = \{ a, b \} \) and \( \text{park} = \{ e, f \} \). The Cartesian product is thus \( \{ (a, e), (a, f), (b, e), (b, f) \} \). Note we use \( d_1, \ldots, d_4 \) to denote a Cartesian object that joins object \( d_i \) from relation \( r_i \), and we refer to \( d_i \) as the \( r_i \) dimension; e.g., \( (a, e) \) joins \( a \) and \( e \) as the house and park dimensions respectively. As example data, Figure 3.5 shows the predicate scores for each Cartesian object. For instance, since \( x \) is a selection over house (and similarly \( p_r \) over park), it only depends on the corresponding dimension; e.g., \( x[(a, e)] = x[(a, f)] = 0.9 \). In contrast, as a join predicate over both relations, \( p_j \) depends on both dimensions. With all Cartesian objects fully materialized, MPro can directly apply as if all predicates are selections.

However, while conceptually operating on Cartesian objects, MPro can incrementally materialize them. To incorporate such lazy materialization, we use a wildcard “*” for an unmaterialized dimension in a Cartesian object. For instance, in Figure 3.6, \( u = (a, *) \) has the first but not the second dimension materialized. We then explode unmaterialized dimensions on
<table>
<thead>
<tr>
<th>OID</th>
<th>$x$</th>
<th>$p_r$</th>
<th>$p_j$</th>
<th>$\mathcal{F}(x, p_r, p_j)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, e)</td>
<td>0.90</td>
<td>0.50</td>
<td>0.70</td>
<td>0.50</td>
</tr>
<tr>
<td>(a, f)</td>
<td>0.90</td>
<td>0.80</td>
<td>0.80</td>
<td>0.80</td>
</tr>
<tr>
<td>(b, e)</td>
<td>0.70</td>
<td>0.50</td>
<td>0.90</td>
<td>0.30</td>
</tr>
<tr>
<td>(b, f)</td>
<td>0.70</td>
<td>0.80</td>
<td>0.30</td>
<td>0.30</td>
</tr>
</tbody>
</table>

**Figure 3.5** Dataset 4 for join query $\mathcal{F} = \min(x, p_r, p_j)$.

<table>
<thead>
<tr>
<th>step</th>
<th>action</th>
<th>ceiling queue $Q$</th>
<th>output ($K$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>initialize $Q$ and $K$</td>
<td>$(a, e):0.90, (b, e):0.70$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>2.</td>
<td>explode $(a, *)$ for $p_r$ into $(a, e), (a, f)$</td>
<td>$(a, e):0.90, (a, f):0.90, (b, e):0.70$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>3.</td>
<td>probe $r_{ap_r}(a, e)$</td>
<td>$(a, e):0.90, (b, e):0.70, (a, f):0.50$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>4.</td>
<td>probe $r_{ap_r}(a, f)$</td>
<td>$(a, f):0.80, (b, e):0.70, (a, f):0.50$</td>
<td>$\emptyset$</td>
</tr>
<tr>
<td>5.</td>
<td>probe $r_{ap_r}(b, e)$</td>
<td>$(b, e):0.70, (a, e):0.50$</td>
<td>$(a, f):0.80$</td>
</tr>
<tr>
<td>6.</td>
<td>pop top complete objects from $Q$ into $K$</td>
<td>$\emptyset$</td>
<td>$(a, f):0.80$</td>
</tr>
<tr>
<td>7.</td>
<td>stop condition holds</td>
<td>$\emptyset$</td>
<td>$(a, f):0.80$</td>
</tr>
</tbody>
</table>

**Figure 3.6** Illustration of MPro for a join query.

demand. Recall that MPro will need an object $u$ only when it surfaces to the top of the ceiling queue $Q$ (Figure 3.2)—i.e., $u$ has the highest ceiling score, and thus $r_{ap_r}(u)$ is necessary (by Theorem 1) for the next predicate $p$. MPro can thus wait until this point to materialize the required dimensions (if not yet) for $p$. Consider $u = (a, *)$ and assume $p_r$ is to be probed next, which depends on the unmaterialized park dimension. Since the wildcard represents park = {e, f}, MPro will explode $u$ on this newly needed dimension to materialize $u$ into \{<a, e>, <a, f>\}.

Figure 3.6 illustrates MPro for evaluating $Q_3$, with search predicate $x$ and expensive predicates when $P^y = (p_r, p_j)$. MPro begins by initializing $Q$ with the $x$ output stream (of house objects), leaving the park dimension unmaterialized. MPro then explodes the top object $u = (a, *)$ for $p_r$, in order to perform necessary probe $r_{ap_r}(u)$. The resulting objects $(a, e)$ and $(a, f)$ will be inserted back to (the top of) $Q$. Since they share the same top ceiling score (as $u$), MPro will order them with the deterministic tie-breaker as Chapter 2 discussed. Suppose that $(a, e)$ becomes the new top object; $r_{ap_r}(a, e)$ will then be the next necessary probe. MPro will continue as usual and eventually output $(a, f):0.80$ (for house $a$ and park $f$) as the top-1 answer.
3.2.2 Approximate Queries

We next discuss a simple extension of Algorithm MPr0 for approximate evaluation. Such evaluation will return top-$k$ answers that are approximate, without guaranteeing all of them to be actually in the top-$k$ (in addition, their final scores may not be fully computed). Approximation is often an effective technique for speeding up query processing. In fact, given their inherent “fuzziness,” ranked queries may prove to be more pertinent (than Boolean queries) to approximate processing. In addition, ranked queries are often used in iterations, in which users formulate several (related or refined) queries to explore the result space. In such a query session, it is not critical to provide full accuracy in the initial iterations; instead, response time may be more important. Such trade-off will be especially useful for probe predicates, given their expensive nature.

Therefore, an approximate query system should allow principled trade-off between accuracy (of results) and cost (of processing). Let $A = \{a_1, \ldots, a_k\}$ be the approximate top-$k$ answers, while $K = \{u_1, \ldots, u_k\}$ are the correct ones. To control the approximation, we need to measure how $A$ approximates $K$. One can either control the rank range difference of $A$ and $K$ or the difference of member objects of $A$ and $K$. In our initial exploration, we found that the former measure is too pessimistic to save probes from the exact evaluation. Thus, in this section, we introduce the latter and define the closeness $\sigma$ as $\frac{|A \cap K|}{k}$. Note that $\sigma$ represents both of the standard metrics used extensively in information retrieval, namely precision $\frac{|A \cap K|}{|A|}$ and recall $\frac{|A \cap K|}{|K|}$, because $|A| = |K| = k$.

As a desirable feature, Algorithm MPr0 can be easily extended to process queries with controlled closeness. When approximation is desired, users specify both result size $k$ and closeness $\sigma$, which means at least $\sigma k$ answers must be correct. To enable this approximation, Algorithm MPr0 only needs a new stop condition (see Figure 3.2) as follows:

$$SC_{\text{app}}(\sigma) \leftarrow \text{“at least } \sigma k \text{ objects among the top-}k \text{ are completely evaluated”}$$

When $SC_{\text{app}}(\sigma)$ holds, MPr0 can stop and return the top-$k$ objects as approximate answers $A$. This new condition will guarantee the desired closeness: At least those complete objects in $A$ are correct top-$k$ answers, since their final scores are higher than the ceiling score of any object not in $A$. Note that our original $SC$ in fact corresponds to $SC_{\text{app}}(\sigma)$ when $\sigma = 1$ (which means
exact results), i.e., $\mathcal{S} = \mathcal{S}_{app}(1)$. Thus, by varying $\sigma \leq 1$, we will allow Algorithm MPro to stop earlier when there are less than $k$ complete objects on the top. (Note that the top objects will appear in both the output queue $\mathcal{K}$ and the top slots of the ceiling queue $\mathcal{Q}$.) To illustrate, consider our example top-2 query in Figure 3.3, if the desired closeness is $\sigma = 0.5$, MPro can actually stop at after probing $r_{ap_i}(a)$ (line 3), where 1 (as $0.5 \cdot 2$) out of top-2 is complete. This early stopping for $\sigma = 0.5$ thus saves 2 out of 4, or 50%, probes as compared to $\sigma = 1$.

### 3.2.3 Data-Partitioned Parallel Processing

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

As a basis, we first explore a rather straightforward extension of performing multiple necessary probes concurrently: Given a top-$k$ query, Section 3.1.2 observed that there are generally multiple necessary probes at any time during query processing. Note, as any of such necessary probes must be performed sooner or later, as they are required independent of other probes (Definition 1)- Thus, MPro can be straightforwardly parallelized to execute many necessary probes concurrently to speedup performance: That is, if the predicate subsystems (for evaluating probes) support concurrent probes (such as a multi-threaded local subsystem or Web server), the *probe-parallel* MPro can perform some or all the necessary probes (depending on the available concurrency level), update $\mathcal{Q}$ with all such probes, and continue to perform next batch of necessary probes. To illustrate, consider parallelizing our example in Figure 3.3 when $k = 2$. At line 1, MPro will execute all necessary probes $r_{ap_i}(a)$ and $r_{ap_j}(b)$ concurrently. Updating $\mathcal{Q}$ accordingly, MPro will still find $a$ and $b$ as the top-2 incomplete objects, and thus perform $r_{ap_i}(a)$ and $r_{ap_j}(b)$. MPro will then output $a$ and $b$ as the top-2 answers. Note that this probe-parallel algorithm performs the same set of necessary probes (as in Figure 3.3), although in a different order.

Observe that probe-parallel MPro, by performing nothing but necessary probes, maintains "probe minimality", which can be important in some application scenarios, such as probing web servers with minimal server overhead. However, more often, the goal of parallelization is achieving as much parallelism possible (even though probes performed may not be minimal).
Meanwhile, probe-parallel MP\textsuperscript{ro} is not suitable in achieving maximal parallelism as it, to maintain probe minimality, bounds parallelism to the number of incomplete objects among current top-\(k\) – which is initially \(k\) and decreases progressively to 0 as top-\(k\) are completed.

We thus prefer data-partitioned parallelization which partitions the database and processes chunks concurrently, if application scenario allows data partitioning. As Figure 3.7 shows, dpMP\textsuperscript{ro} (for data-parallel MP\textsuperscript{ro}) consists of two main steps: First, in data distribution, dpMP\textsuperscript{ro} will uniformly distribute the input database \(\mathcal{D}\) into \(c\) data chunks, each of which is of size \(1/c\) of \(\mathcal{D}\), for a given chunking factor \(c\). (Of course, this partitioning can be done off-line as it is query independent.) To maximize work distribution (or concurrency), we wish that top answers will uniformly come from different chunks. That is, the data distributor must ensure that the chunks be as “similar” to each other as possible, by identifying and distributing similar objects (that will perform similarly in queries) in \(\mathcal{D}\) to different chunks.

Second, during incremental merging, dpMP\textsuperscript{ro} will incrementally access and merge top answers from each \(\mathcal{D}_i\). Note such incremental access (i.e., \(I_i.next\)) can be supported by slightly extending MP\textsuperscript{ro}: This incremental extension, referred to as iMP\textsuperscript{ro}, is essentially the same as MP\textsuperscript{ro} in Figure 3.2, except now the ceiling queue is “persistent” during incremental access. Thus iMP\textsuperscript{ro} will initialize \(Q\) only at the very first next call. For any subsequent call, iMP\textsuperscript{ro} will continue to populate \(K\) with the next top answer just like MP\textsuperscript{ro}. Once top answers are collected from each \(\mathcal{D}_i\), dpMP\textsuperscript{ro} then uses a merging queue \(\mathcal{M}\) to sort them by their query scores. If \(u\) is the top of \(\mathcal{M}\), it has outperformed those still in \(\mathcal{M}\). dpMP\textsuperscript{ro} checks if \(u\) also outperforms all unseen objects by comparing it to the last top scores of every \(I_i\). (These top scores may have been output to \(K\) and not present in \(\mathcal{M}\) any more.) If so, \(u\) must be the overall top (of entire \(\mathcal{D}\)). Otherwise, some unseen objects may be better, and dpMP\textsuperscript{ro} will request parallel access to load new top objects from all \(\mathcal{D}_i\). Note that parallel loading will bring in more objects to \(\mathcal{M}\) in just one access time, saving some accesses that might be needed later.

Observe that dpMP\textsuperscript{ro} speeds up by distributing work to \(c\) parallel MP\textsuperscript{ro} “processors”: Consider a top-\(ck\) query over a database of size \(cn\), denoted \(DB(cn)\). First, as the database is chunked uniformly, the answers should distribute uniformly among the \(c\) chunks—thus each MP\textsuperscript{ro} processor only needs to find (around) \(k\) instead of \(ck\) answers. Further, the chunking reduces the database from \(DB(cn)\) to \(DB(n)\) for each processor. Putting together, dpMP\textsuperscript{ro} parallelizes the time of finding top-\(ck\) over \(DB(cn)\) into that of top-\(k\) over \(DB(n)\), reducing both
the database and retrieval sizes by $c$ times for each processor. As Section 3.2.4 will quantify, this reduction results in an overall $c$-times speedup.

### 3.2.4 Scalability

Lastly, based on necessary probes (Definition 1) which determine algorithm-independent optimal cost for expensive predicates given $P$, we study how much this required cost will be, and how it scales for larger databases. Note that $MPro$, as a probe-optimal algorithm (Theorem 2), executes exactly only necessary probes. Therefore, $MPro$ can be an effective vehicle for understanding minimal-probing costs. To this end, Section 3.4 will experimentally evaluate minimal-probing cost with respect to different queries ($i.e.$, $k$ and $F$) and databases ($i.e.$, how objects score under $F$). This section seeks to understand analytically how the minimal-cost scales, by addressing an important question: Given a top-$k$ query $F(x, p_1, \ldots, p_n)$, if it requires $m_i$ probes for each predicate $p_i$ over a database of size $n$, or $DB(n)$, how will the required necessary-probes increase when the database is scaled up $c$ times, $i.e.$, $DB(cn)$. As explained, we specifically study the scalability of $MPro$ to generally answer this question.

To focus on scalability, we assume uniform scaling, such that $DB(cn)$ will perform “statistically” similar to $DB(n)$—$i.e.$, the two databases differ in size but not nature, so that we can
isolate data size to study its effect. To allow analytical study, we approximate uniform scaling by simply replicating $DB(n)$ $c$ times to generate $DB(cn)$. We thus obtain an interesting result: If a database is uniformly scaled up $c$ times, MPr0 can retrieve $c$ times more top answers, with $c$ times more probe cost. While we leave a formal proof to Appendix, Section 3.4 experimentally verifies this result—over datasets that are only similar but not identical.

**Theorem 4 (Probe Scalability):** Consider a ranked query $\mathcal{F}(x, p_1, \ldots, p_n)$ and a schedule $\mathcal{P}$. Suppose that $DB(n)$ is a database of size $n$; let $DB(cn)$ be the database containing the $cn$ objects generated by replicating $DB(n)$ $c$ times. $C((\mathcal{P}, \text{OSch}(\mathcal{P})), \mathcal{F}, DB(cn), ck) = c \cdot C((\mathcal{P}, \text{OSch}(\mathcal{P})), \mathcal{F}, DB(n), k).

**Proof:** We show that $c \cdot C(k,n)$ both upper and lower bounds $C(ck,cn)$, and thus their equality holds. Note that for brevity of notation, we omit $(\mathcal{P}, \text{OSch}(\mathcal{P})), \mathcal{F}$ and $\mathcal{P}$ in the input lists.

**(1) upper bound:** Since MPr0 is a probe-optimal algorithm (Theorem 2), $C(ck, cn)$ is the minimal probe cost for finding the top-ck from $DB(cn)$, and the cost will be upper-bounded by that of any algorithms of this purpose.

To give such an upper bound, consider Algorithm dpMPr0$(\mathcal{F}, ck, \mathcal{P}, DB(cn), c)$ using $c$ as the chunking factor (Figure 3.7). How much probe cost will dpMPr0 pay? The data distributor in Step (1) will partition $DB(cn)$ into $c$ identical data chunks, i.e., $Di = DB(n)$. Every parallel access of top objects from all $I_i$ will thus return $c$ identical objects – it will take exactly $k$ such accesses to merge the desired $ck$ answers. That is, each $I_i$ effectively finds top-$k$ over its $DB(m)$ chunk using MPr0, paying probe cost as $C(k,n)$. Since dpMPr0 uses $c$ such components, it will pay the total cost as $c \cdot C(k,n)$.

**(2) lower bound:** Since every object $u$ in $DB(n)$ induces $c$ replicas $u_1, \ldots, u_c$ in $DB(cn)$, we show that any necessary probe on $u$ in MPr0$(k, DB(n))$ will induce $c$ cost-identical probes on $u$’s replicas in MPr0$(ck, DB(cn))$, and thus $C(ck, cn)$ is lower bounded by $c \cdot C(k,n)$. That is, we show that if $ra_p(u)$ is performed in MPr0$(k, DB(n))$, then $\forall i : ra_p(u_i)$ must be performed in MPr0$(ck, DB(cn))$.

Consider MPr0$(k, DB(n))$. Assume that $ra_p(u)$ is performed at some time $t$ during this processing. Since MPr0 is probe-optimal (Theorem 2), $ra_p(u)$ must be a necessary probe. Thus, at time $t$, by Theorem 1, $u$ is incomplete and its ceiling score $\mathcal{F}_{T_u}[u]$ is ranked among top-$k$. Note that $T_u$ covers every predicate preceding $p$ on schedule $\mathcal{P}$. 

40
Consider MPro$(ck, DB(cn))$. To derive a contradiction, assume that, for some $u_i$, $ra_p(u_i)$ is not performed. Let $T_{u_i}$ be the evaluated predicates of $u_i$ throughout MPro$(ck, DB(cn))$. As $u_i$ does not reach $p$ on $P$, $T_{u_i} \subseteq T_u$.

Since $u_i$ is incomplete, it cannot be in the top-$ck$ answers. Let the top-$ck$ answers that MPro$(ck, DB(cn))$ returns be $v_{11}, \ldots, v_{1c}, \ldots, v_{k1}, \ldots, v_{kc}$, where $v_{j1}, \ldots, v_{jc}$ are replicas of some object $v_j$ in $DB(n)$. The stop condition of MPro ensures that $\forall v_j : \mathcal{F}_{T_{u_i}}[u_i] < \mathcal{F}[v_j]$ or $\mathcal{F}_{T_{u_i}}[u_i] < \mathcal{F}[v_j]$. We show that, at time $t$ during MPro$(k, DB(m))$, $u$ cannot be ranked among top-$k$, a contradiction: At time $t$, $u$ has $T_u$ evaluated; suppose that every $v_j$ has $T_{v_j}$ evaluated. Since $T_{u_i} \subseteq T_u$ and $u$ and $u_i$ are identical, $\mathcal{F}_{T_u}[u] \leq \mathcal{F}_{T_{u_i}}[u_i]$. Also, by definition, $\mathcal{F}[v_j] \leq \mathcal{F}_{T_{v_j}}[v_j]$. It follows that $\mathcal{F}_{T_u}[u] < \mathcal{F}_{T_{v_j}}[v_j], \forall v_{j1}, \ldots, v_{jk}$, and thus $u$ cannot be ranked in the top-$k$, a contradiction.

Theorem 4 gives the scalability of necessary-probes in general as well as MPro in particular. This result enables several interesting observations. First, it shows that MPro has good data scalability, in which cost increase is sublinear in database size: The probes required for finding $k$ answers from $DB(cn)$ will be less than $c$ times that for finding the same number of answers from $DB(n)$: As the stop condition (SC in Figure 3.2) ensures, MPro will stop earlier and pay less probe cost for a smaller retrieval size, and thus $\mathcal{C}((P, OSch(P)), F, DB(cn), k) < \mathcal{C}((P, OSch(P)), F, DB(cn), ck)$. (Section 3.4 shows how necessary-probes increase over retrieval sizes.) With Theorem 4, we derive the sublinear growth of cost: $\mathcal{C}((P, OSch(P)), F, DB(cn), k) < c \cdot \mathcal{C}((P, OSch(P)), F, DB(n), k)$. Since complete probing (of the standard sort-merge framework) requires linear increase, MPro will scale better and reduce more probes for larger databases.

Second, MPro will enjoy good resource utility: If the computing resource scales up $c$ times, our framework can receive linear speedup by using $c$ concurrent MPro processors in parallel in dMPро. As Section 3.2.3 discusses, by reducing both the retrieval and database size for each processor, dMPро can reduce the probe cost for each processor from $\mathcal{C}((P, OSch(P)), F, DB(cn), ck)$ to $\mathcal{C}((P, OSch(P)), F, DB(n), k)$. As Theorem 4 asserts, the latter is $\frac{1}{c}$ of the former, which means a $c$-times speedup or linear to the resource increase.

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

3.3 Case Study: Integration with RDBMS for Post-Filtering Ranking

As a case study, this section discusses how to integrate MPro with a relational DBMS, namely PostgreSQL 7.3.2, to efficiently support post-filtering predicates. Section 3.3.1 overviews the current status of DBMS support for top-\(k\) queries. Section 3.3.2 then discusses the details of our implementation.

3.3.1 Overview

As discussed in Section 1, top-\(k\) queries naturally arise in many database applications. However, despite its wider applicability and expected performance payoff, DBMS support for top-\(k\) queries has been minimal. Though major DBMSs allow top-\(k\)-like query syntax, \emph{e.g.}, \texttt{ROWNUM} \(\leq k\) in Oracle or \texttt{limit} \(k\) in PostgreSQL, they only support simplistic heuristic optimization for top-\(k\) query processing, such as favoring pipelined query plans to optimize for early results.

Recently, there have been research efforts in processing top-\(k\) queries efficiently in relational context. Carey et al.\cite{carey2013, carey2014} discussed how the cardinality limit of \(k \ll m\) can be exploited by the query optimizer and executor subsystem of a typical relational DBMS. By leveraging relational query optimizer, their work optimizes top-\(k\) queries with Boolean filtering conditions in a single query optimization framework. However, their \texttt{Stop} operator incurs a complete probing when scoring function is not column value itself, \emph{e.g.}, \texttt{Query} \(Q_1\) with \(\mathcal{F} = \min\). Later, Chaudhuri et al.\cite{chaudhuri2017} proposed to support some limited scoring functions efficiently, by translating them into SQL range queries. However, their query model does not support arbitrary Boolean expressions as filtering condition.

In contrast, our goal is to support a general post-filtering ranking queries with any monotonic scoring function and arbitrary filtering conditions. Note that, our work complements previous works by (1) efficiently supporting arbitrary Boolean filtering conditions, while (2) implementing
MPro as Stop operator to support optimal evaluation of post-filtering predicates. Section 3.3.2 further discusses the details of our implementation design.

3.3.2 Design

As discussed in Section 3.3.1, major DBMSs allow a general post-filtering ranking queries by syntax, as follows:

```sql
select id from house
where zip=60603
order by min(new(age), cheap(price, size), large(size))
stop after 5
```

(Query Q4)

However, using PostgreSQL “as is” for Query Q4 incurs the prohibitive cost of complete probing. To illustrate, Figure 3.8 shows two possible execution plans for Query Q4: Depending on the availability of indices, IndexScan or SeqScan retrieves the tuples satisfying the filtering conditions. Then the scoring function \( F \) is evaluated for all the selected tuples. The Sort and Limit operators then follow to sort the tuples by \( F \) scores and return the first \( k \) results respectively.

![Figure 3.8](image)

(a) (b)

**Figure 3.8** Two possible execution plans generated by PostgreSQL to evaluate Q4.

To avoid this prohibitive cost of complete probing, we implement MPro in the Limit node to evaluate ranking condition with minimal probes. To minimize the change to the query engine in implementing MPro, we adopt the following two-phase approach: Stage 1 leverages query optimizer to evaluate filtering conditions efficiently and generate a sorted stream \( \mathcal{X} \) of filtered objects as an input for MPro. Stage 2 implements MPro to evaluate ranking predicates with minimal probes.

In Stage 1, we generate a sorted stream \( \mathcal{X} \) of tuples (1) selected by filtering conditions and (2) ordered by search predicate \( x \). For instance, assuming new is the only search predicate in
$Q_4$, $\mathcal{X}$ for $Q_4$ will be the stream of houses in Chicago, ordered by the score of new, which is equivalent to the results of the query below:

\begin{verbatim}
select id, new(age), price, size from house
where zip = 60603
order by new(age) desc           (transformed $Q_4$)
\end{verbatim}

To implement Stage 1 with minimal change, we simply modify query tree to correspond to the transformed query instead of the original query submitted by the user. The original, unchanged query optimizer then identifies the best query plan for generating sorted stream $\mathcal{X}$, based on the statistics and availability of indices. To illustrate, suppose we have indices on predicate new(age) and attribute zip. Figure 3.9 shows two possible query plans: In plan (a), we use sorted access on new(age) to produce the sorted stream and apply filtering conditions zip = 60603 on the fly to filter out disqualified tuples. Alternatively, plan (b) first selects houses in Chicago. As an index on base table for new(age) cannot be used after houses in Chicago are selected, we use Sort operator instead to order houses by the predicate score. Among the plans enumerated, the optimizer of PostgreSQL will make an informed decision based on the estimated cost of each plan: For instance, in this example, plan (b) will be favored when the filtering condition is highly selective. Note this decision is not trivial when queries have multiple filtering conditions with different selectivities and index availabilities – however, by leveraging relational query optimizer, we can exploit the statistics and functionalities of the underlying RDBMS in identifying the optimal plan for Stage 1.

Stage 2 then implements MPro in the Limit node to evaluate ranking conditions efficiently. Our experience found it straightforward to implement MPro, as its main data structure is a priority queue supported as in-the-core binary heap by PostgreSQL. The new Limit node keeps the same interface for interacting with other operators. However, when a tuple arrives at the node, it will be placed in the priority queue as an input of MPro until top-$k$ results are returned to the users.

Lastly, we discuss how predicate scheduling schemes in Section 3.1.3 can be implemented. Schemes that do not require pre-execution sampling (i.e., MPro-Adaptive and MPro-External) only requires additional routine (to determine predicate scheduling) or buffering (to store scores). However, pre-execution sampling can also be straightforwardly implemented by augmenting the base table by generating a random number in $[0:1]$ for each tuple and storing it as
the value of a new attribute `random`. Similarly to Stage 1 implementation, we manipulate query plan to select those satisfying the condition `random < p/100` to retrieve `p%` random objects.

3.4 Experiments

This section reports our extensive experiments for studying the effectiveness and practicality of Algorithm MPro. Our experiments in fact have a two-fold interpretation. First, our experiments specifically quantify the practical performance of Algorithm MPro. We extensively compare our predicate scheduling framework to other existing schemes and quantify actual response time over real-life databases. Second, recall that, we address probe minimization by principled scheduling optimization which consists of provably-optimal object scheduling and dynamically optimized predicate scheduling. Thus, in principle, MPro represents the “lower bound” performance, which we will empirically study over various configurations. To contrast, we adopt two interesting “upper bounds” costs: First, by completely probing every object, SortMerge defines the absolute upper bound cost of complete probing. Second, Algorithm TAZ [20] minimizes the number of objects evaluated (object minimization) by halting as soon as top-`k` results are identified. However, by always completely probing objects, its cost represents the upper bound cost of algorithms with object minimization.

We measured two different performance metrics. Consider query $Q = \mathcal{F}(x, p_1, \ldots, p_q)$. First, to quantify the relative probe performance, we measure how MPro saves unnecessary probes with the probe ratio metric. Suppose that MPro performs $m_i$ probes for each $p_i$. In contrast, SortMerge will require $n$ probes (where $n$ is the database size) for every probe predicate. The
overall probe ratio (in %) is thus \( \text{pratio}(Q) = \sum_{i=1}^{\gamma} \frac{n_i}{q_i} \), which is the sum of the predicate probe ratio \( \text{pratio}(p_i) = \frac{n_i}{q_i} \), i.e., \( \text{pratio}(Q) = \text{pratio}(p_1) + \cdots + \text{pratio}(p_\gamma) \).

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

Our experiments used both synthetic data as well as a “benchmark” database (with real-life data). For extensive study, Section 3.4.1 and 3.4.2 report “simulations” over data synthesized with standard distributions. To understand the performance of MPro in real-world scenarios, Section 3.4.3 experiments with a benchmark real-estate database (essentially the same as our examples). All experiments were conducted with a Pentium 2.4GHz PC with 512MB RAM.

### 3.4.1 Baseline Comparison

In this section, we study the lower-bound performance of MPro over various configurations, contrasted with upper-bound performances of SortMerge and TA\textsubscript{2}. Among three instantiations of MPro (Section 3.1.3), we use MPro-Adaptive in particular, as it performs best overall (and thus closer to the lower-bound performance) as Section 3.4.2 will discuss. All queries in this section are of the form \( \mathcal{F}(x, p_1, p_2, p_3) \) (as in Section 3.4.3). Our configurations are characterized by the following parameters: (1) Score distribution \textit{D}: the distributions of individual predicate scores, including the standard unif (uniform) and norm (normal) distributions as well as funif, a home-brew “filtered-uniform” distribution (see below). (2) Scoring function \( \mathcal{F} \): \textit{min}, \textit{avg} (average), and \textit{geavg} (geometric average). (3) Database size \textit{n}: 1k, 10k, and 100k.

\textbf{D: Score Distribution.} Figure 3.10(a) present the results with different score distributions, which characterize predicates. The left figure presents the results for normal distribution (with mean 0.5 and variance 0.16). The second distribution funif simulates “filtering” predicates, which are likely to be used in real-life queries that “filter out” a certain portion of data; e.g., the \textit{large} predicate in our benchmark queries (to be discussed in Section 3.4.3) qualifies 78% objects with zero scores (as their sizes are out of the desired range). We thus define funif(\( f \)) (for \textit{filtered uniform}) to simulate such predicates, where \( f \)\% objects score 0 and the rest are uniformly distributed. The right figure (Figure 3.10a) plots the cost for funif(75).
Figure 3.10 Different score distributions (a) and scoring functions (b) with $m = 1k$.

Observe that, in contrast to SortMerge which requires the same complete probing cost regardless of retrieval size, TA$_Z$ and MPo show desired “proportional” cost behavior by minimizing the objects evaluated. However, MPo significantly outperforms TA$_Z$ by short-circuiting some probes on objects evaluated. Observe that filtering (the right figure) makes MPo even more effective—For over $f = 75\%$ of objects that score the minimal-possible score 0, MPo can effectively short-circuit its evaluation as soon as its ceiling score reaches the final score 0.

**F: Scoring Function.** To understand the impact of scoring functions (which can form the basis of how a practical system may choose particular functions to support), we compare some common scoring functions: min (Figure 3.10a) and some representative average functions (Figure 3.10b), namely arithmetic average $avg : (x + p_1 + p_2 + p_3)/4$ and geometric average $gavg := (x \cdot p_1 \cdot p_2 \cdot p_3)^{1/4}$.

We found that min is naturally the least expensive, as it allows low scores to effectively decrease the ceiling scores and thus “filter” further probes. (In contrast, max will be the worst case, requiring complete probing.) Average functions are relatively less effective in filtering and thus adversely affect performance, especially that of TA$_Z$ which degenerates close to complete probing. The two average functions perform similarly, with $F = gavg$ being about 5% to 10% cheaper than $avg$.  

47
**Figure 3.11** Different database sizes $m$ with $(D=\text{norm}, F=\text{min})$.

**n: Database Size.** Figure 3.11 presents the performance evaluation for $m = 1k, 10k,$ and $100k$ when $F=\text{min}$ and $D=\text{norm}$. Figure 3.11 plots the probe ratios of MPr for different $m$ as lines with different styles, compared with those of TA$_Z$ plotted as the same lines but with markers. Observe that MPr scales more gracefully over $m$, compared to TA$_Z$. Further, it is interesting to observe that this evaluation confirms the scalability of MPr over $m$, studied in Section 3.2.4: Note the probe ratios are approximately the same if the relative retrieval size is the same, regardless of the database size. For instance, for $k = n \cdot 1\%$ (i.e., $k = 10, 100,$ and $1000$ for $n = 1k, 10k,$ and $100k$ respectively), the probe-ratios are all about $15\%$. Similarly, the probe-ratios for $k = n \cdot 0.1\%$ are about $8\%$. This observation is critical in evaluating the scalability of MPr over $n$, as users will probably interested in only the very top hits, regardless of the database size $n$. MPr will thus be relatively (compared to the baseline scheme) more efficient for larger databases, which shows its scalability over $n$.

Note that this “constant-ratio” observation also verifies Theorem 4. For $(k = 10, n = 1k)$, the probe cost is $8\% \cdot q \cdot n$ (where $q$ is the number of predicates). Now let $c = 10$; we observed that $(ck = 100, cn = 10k)$ costs $8\% \cdot q \cdot cn$, i.e., $c$ times more. That is, when the database is scaled up $c$ times, MPr retrieves $c$ times more top answers, with $c$ times more probe cost.

### 3.4.2 Predicate Scheduling Effectiveness

Recall from Theorem 2 that the optimality of MPr is conditional to the predicate schedule $\mathcal{P}$. This section thus evaluates the effectiveness of our predicate scheduling schemes (i.e., PSch), compared to theoretical optimal schemes and existing per-object scheduling schemes. To experiment with various aspects, we generate 100 random configurations varying the following three factors that affect predicate scheduling (as discussed in Section 3.1.3):
1. Scoring function $\mathcal{F}$: Though our schemes generally support any monotonic scoring function, we set $\mathcal{F}$ as a weighted average to accommodate comparison with existing schemes proposed for average scoring functions. In particular, we set $\mathcal{F} = \frac{x + w_1 \cdot p_1 + w_2 \cdot p_2 + w_3 \cdot p_3}{1 + w_1 + w_2 + w_3}$ and generate 100 random weights $(w_1, w_2, w_3)$, with $w_i$ in $[1:10]$.

2. Predicates: In this experiment, we model predicate distributions as $(\text{funif}(f_1), \text{funif}(f_2), \text{funif}(f_3))$, as such distributions have been practically observed in real benchmark queries (Section 3.4.3), while they can also be easily synthesized with varying filtering ratio $f_i$.

We generate 100 configurations of $(f_1, f_2, f_3)$, with $f_i$ in $[0:100]$.

3. Per-probe costs: We generate 100 configurations of $(C_1, C_2, C_3)$ with $C_i$ in $[1:10]$.

We quantify the effectiveness of predicate schedule $\mathcal{P}$, as the average cost of its corresponding optimal probe schedule $\mathcal{S} = (\mathcal{P}, O_P)$, over various configurations discussed above. In particular, the random configurations, of we consider the following predicate scheduling schemes in comparison:

**Optimal schedulers:** To quantify how closely schedulers approximate the optimal scheduling, we consider global- and object-optimal schemes (denoted as GOpt and OOpt respectively) which identify the optimal global and per-object predicate schedules. To guarantee the optimality, both schemes require complete knowledge of predicate scores: With such knowledge, GOpt can exactly compute selectivity and thus the probe cost $C(\mathcal{P}, \mathcal{F}, D, \theta_k)$ (in Eq. 3.7). GOpt then simply enumerates all $n!$ possible global predicate schedules and identifies the minimal-cost one. Similarly, with complete knowledge, OOpt can exhaustively enumerate $n!$ possible sub-schedules to identify the optimal sub-schedule $\mathcal{P}^i$ of every object $u_i$, which compositionally construct complete predicate schedule $\mathcal{P} = \mathcal{P}^1 \times \cdots \times \mathcal{P}^n$, as discussed in Section 3.1.3. Note, these schemes are only of theoretical interest for comparison, as obtaining complete knowledge is infeasible.

**Per-object schedulers:** We consider existing per-object schemes - Upper-Subset [5] and Upper-Rank (its unpublished follow-up [34]). Both schemes share the same object scheduling scheme as in MPro, but invoke predicate scheduler at each probe, i.e., $nq$ invocations: At each probe on object $u$, Upper-Subset determines the next predicate to evaluate, among a candidate “subset” of predicates for $u$, by picking the one with highest $\frac{w_i}{C_i}$. Assuming each predicate
$p_i$ is expected to score $e_i$ (which is typically set as 0.5), candidate subset is computed at every invocation, as the minimal-cost subset of unevaluated predicates which is expected to decreases the ceiling score below $\theta_k$. **Upper-Rank** later improves this scheme to further realize the benefit of per-object scheduling: Unlike **Upper-Subset** ordering predicates by global information $\frac{\mu_i}{e_i}$, **Upper-Rank** proposes a more sophisticated metric “rank” to leverage up-to-date per-object information in, not only identifying candidate subset but also ordering them.

**Our schedulers:** We consider all three predicate scheduling schemes discussed in Section 3.1.3, namely MPro-Sampling, MPro-Adaptive, and MPro-External. Note, for fair comparison, we design MPro-Sampling and MPro-Adaptive to use samples of the same size $s$: That is, predicate scheduling of MPro-Adaptive will be activated when $s$ scores for every predicate have been collected. Also, given $F$ as weighted average, we design MPro-External to schedule predicates in the decreasing order of $\frac{\mu_i}{e_i}$.

To compare the effectiveness of these schemes, Figure 3.12(a) compares the optimal probe costs of retrieving top $k = 100$ over $n = 10k$ objects, given the predicate schedules identified from the schemes above. In particular, as a cost metric, we extend probe ratio \textit{cost ratio} (relative to complete probing) to take different per-probe cost $C_i$ into consideration, \textit{i.e.},

$$\sum_{i=1}^{k} \frac{C_{q_i}}{\sum_{i=1}^{n} C_{m}}.$$  

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

**Global scheduling effectiveness:** Observe that MPro-Adaptive closely approximates GOpt by 1% margin. Note, in principle, MPro-Sampling and MPro-Adaptive should be as good as GOpt, only if provided with representative samples, as our cost model for predicate scheduling (Eq. 3.8) precisely models the actual probe cost.

**Local scheduling effectiveness:** Observe that, even though object-specific schedulings of Upper-Subset and Upper-Rank have potential to outperform GOpt and reach closer to OOkt, they fail to realize such potential, as it is non-trivial to obtain enough object-specific information (as discussed in Section 3.1.3). As a result, even a simplistic global scheduling of
MPRO-External achieves comparable performance to Upper-Rank. Meanwhile, Upper-Subset and Upper-Rank have to pay extra $nq$-fold computational scheduling cost (as discussed in Section 3.1.3), which is not reflected by cost ratio metrics used in the figure.

**Empirical performance analysis:** We next empirically analyze relative strengths of scheduling schemes: For closer observation, we first divide our experiment configurations into two groups of 50 configurations each—one group with rather similar predicate score distributions, i.e., $f$-ratio $= \frac{\max(f_i) - \min(f_i)}{\max(f_i)} \leq .75$, and the other with more contrasts, i.e., $f$-ratio $> .75$. Figure 3.12(b) compares the average cost ratios of schemes in the two groups—observe that schemes relying solely on external parameters (i.e., MPRO-External and Upper-Rank) relatively suffer when score distributions differ significantly and start affect scheduling decision. We next observe how MPRO-Sampling and MPRO-Adaptive compare: As observed in Figure 3.12(a) and (b), MPRO-Adaptive overall outperforms MPRO-Sampling by overlapping scheduling overhead with necessary probes (to perform anyway). However, note that, such overlap also implies the bias of sampling towards the objects with high ceiling scores, which may produce different turnouts: Intuitively, if predicates have positive correlation, objects with high ceiling score are likely to score high for unevaluated predicates also. As a result, samples will be biased towards the objects with high overall scores, which are objects of interest in top-$k$ processing. In contrast, when predicates are negatively correlated, sampling will be biased towards low-scored objects. Figure 3.12(c) illustrates their performance implication by plotting the average probe cost of MPRO-Adaptive (normalized to that of MPRO-Sampling as 100%) over the same 100 configurations, when $x$ and $p_i$’s have perfect negative correlation (i.e., correlation coefficient $\rho = -1$), no correlation ($\rho = 0$), or perfect positive correlation ($\rho = 1$): As expected, with none or positive correlation, MPRO-Adaptive outperforms MPRO-Sampling by hiding scheduling overhead and biasing sampling towards objects of interest. Meanwhile, negative correlation hurts scheduler performance, by biasing samples towards low-scored objects.

In summary, our experimental results show that our predicate scheduling schemes closely approximate the theoretical lower bound cost. In contrast, while per-object schemes have potential to outperform such bounds, existing schemes cannot realize such potential and only perform closely to global schemes with extra $nq$-fold computation overhead. Further, our
Figure 3.12 Scheduling for random configurations.

Q₁: select id from house
    order by min(nearby(zip, Chicago), numy(bedroom), cheap(price), large(size))
    stop after k
Q₂: select h.id, s.name from house h, park s
    order by min(nearby(zip, Chicago), numy(bedroom), cheap(price), close(h.zip, s.zip))
    stop after k
Q₃: select id from house
    where nearby(zip, Chicago) x,
    order by min(nearby(zip, Chicago), numy(bedroom), cheap(price), safe(zip))
    stop after k

Figure 3.13 Benchmark queries.

extensive experiments reveal relative strengths of various schemes: collecting actual predicate for scheduling decision is beneficial when distributions differ significantly. How to collect such scores also have performance implications: Biasing collection to necessary probes will have positive implication in the presence of positive or no correlation, or negative otherwise. We conclude the section by stressing that experiments in this section favor Upper-Subset and Upper-Rank by (1) restricting experiments to scenarios they are designed for and (2) excluding their \( nq \)-fold per-probe computational overhead from the metric. In fact, it is not straightforward to extend these schemes for a general class of monotonic function: Reference [34] proposes an ad-hoc extension for \( F = \min \), yet with no \( w_i \) to rely on, the scheduling degenerates into a simplistic scheduling by \( C_i \). In contrast, “principled” nature of our predicate scheduling (Section 3.1.3) enables general applicability over any arbitrary monotonic function.

3.4.3 Benchmarks over Real-Life Database

We next quantify how practical the performance of MPro is in real-world scenarios of accessing a real-estate system. To establish a practical scenario, we populated PostgreSQL with real-world dataset used in [12] (available at http://aim.cs.uinc.edu/). The dataset was extracted from www.realtor.com, in November 2002, by querying all the for-sale houses in
Illinois, resulting in \( n = 20990 \) houses with the attributes id, price, size, bedroom, bath, zip, and city. In addition, we constructed a second relation park of about 110 Illinois state parks, each with attributes name and zip. Among three instantiations of MPro, we used MPro-Sampling in particular (integrated with PostgreSQL as discussed in Section 3.3), as it represents worst-case optimizer scenario with extra online sampling overhead. Our goal is to evaluate the absolute performance and show its practicality, even in this worst scenario.

Our experiments considered a benchmark scenario of finding top houses around Chicago for a small family of four members. We thus created three benchmark queries, as Figure 6.11 (upper) shows. The queries use a search predicate \( \text{nearcity}(\text{zip}, C) \) which returns houses closest to \( C \) (e.g., \( C = \text{Chicago} \)) in order. All the others (\emph{roomy}, \emph{cheap}, \emph{large}, \emph{close}, and \emph{safe}) are probe predicates. Note that the three queries only differ in predicate \( p_3 \). In particular, \emph{large}, \emph{close}, and \emph{safe} represent increasingly more complex and expensive operations—\textit{i.e.,} simple user-defined functions, joins, and external predicates querying web sources.
For each query, we measured the probe ratios and the elapsed time. Figures 3.14(a)–(c) plot the probe ratios (the y-axis) with respect to different retrieval size $k$ (the x-axis), both logarithmically scaled. Each graph shows four curves for $pratio(p_1)$, $pratio(p_2)$, and $pratio(p_3)$, and their sum as the overall probe ratio. For instance, for $Q_3$ in Figure 3.14(a), when $k = 10$ (to retrieve top-10), $p_1$ requires 8.1% probes, $p_2$ 4.2%, and $p_3$ 0.06%, which sum up to 12.36%. In other words, since $n = 20990$ and $q = 3$, the ratios translate to 5100, 2644, and 40 probes (with a total of 7784) out of the $nm = 62970$ complete probes. Observe that the vast majority of complete probes are simply unnecessary and wasted— in this case 87.64% or 55186 probes. As also expected, the probe ratio (relative cost) is smaller for smaller $k$; e.g., for $k = 1$, the overall probe ratio is only 6%. Such “proportional cost” is crucial for top-$k$ queries, as users are typically interested in only a small number of top answers. In fact, Figure 3.14(b)–(c) only show the top range $k \leq 0.1\% \cdot n$ to stress this critical range of interest.

Figures 3.14(d)–(f) evaluates the “absolute” performance of our MP\textsc{ro} implementation, compared to the baseline scheme of using PostgreSQL “as is”. Recall from Section 3.3 that using PostgreSQL “as is” incurs complete probing as in Sort\textsc{Merge}. Referring to Figure 3.14(d), when $k = 10$, MP\textsc{ro} responded in 0.8 seconds, while the baseline scheme takes 2.8 seconds. Query $Q_3$ demonstrates even higher speedup, from 355 seconds to 2.85 seconds or 0.8% time (for $k = 10$). We observe that, when probe predicates are truly expensive as in $Q_3$ (with external predicate $safe$), our framework can be orders of magnitude faster than the baseline scheme. Such speedup can potentially turn an overnight query into an interactive one.
CHAPTER 4

Extending to Unified Cost-based Optimization for Top-k Queries

In the previous chapter, we studied how to minimize cost in limited scenarios where some sources support only random accesses (or “probes”). In contrast, this chapter discusses general top-k access scenarios scheduling arbitrary accesses (random, sorted, and potentially beyond), which complicate optimization with the “progressiveness” and “side-effect” of sorted accesses (Chapter 2). Sections 4.1-4.3 start with defining a comprehensive and focused space for arbitrary top-k algorithms. Section 4.4 then develops optimization schemes over this space. Section 4.5 discusses how our framework unifies and contrasts with existing algorithms. Section 4.6 reports our experiments.

4.1 Defining a Comprehensive Space

This section now tackles the first challenge of defining a “comprehensive” space for scheduling arbitrary accesses. Toward the goal, we ask: What are possible algorithms? Do we need to enumerate all such algorithms? Or can we identify “some” algorithms as a comprehensive space?

To understand a top-k algorithm constituting such a space, we begin with considering an example algorithm (as Example 9 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 4 (Figure 4.2) 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 $K = \{u_3:0.7\}$, i.e., $u_3$ is the top-ranked object with score $F[u_3] = 0.7$. 

55
**Figure 4.1** Cost comprehensiveness.

<table>
<thead>
<tr>
<th>OID</th>
<th>(p_1)</th>
<th>(p_2)</th>
<th>(F)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(u_1)</td>
<td>0.65</td>
<td>0.8</td>
<td>0.65</td>
</tr>
<tr>
<td>(u_2)</td>
<td>0.6</td>
<td>0.9</td>
<td>0.6</td>
</tr>
<tr>
<td>(u_3)</td>
<td>0.7</td>
<td>0.7</td>
<td>0.7</td>
</tr>
</tbody>
</table>

**Figure 4.2** Dataset 4

Recall that, as Section 1 mentioned, this thesis 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 \(ra_i(u_j)\). We note that the two types of accesses differ fundamentally in two aspects (as mentioned in Chapter 2):

- **side-effects**: Sorted access \(sa_i\) has side-effects; To illustrate, in Figure 4.2, 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 4.3 illustrates example algorithms, using a “sorted” list for each predicate—refer to Figure 4.3(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 4.3(a) on \(p_2\) represents Algorithm \(M_1\) performs a random access at time 2, for \(p_2\) on the pointed object \(u_3\).

**Example 9 (Example Algorithm)**: We illustrate, as Figure 4.3(a) shows, how Algorithm TA [19], a representative algorithm, processes \(Q_1\) in Figure 4.3(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., $F[u_3] = \min(0.7, 0.7) = 0.7$, 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., $p_1[u_1] \leq .7$, $F[u_1] = \min(p_1[u_1], p_2[u_1])$ cannot be higher than $F[u_3] = .7$.

As Example 9 shows one possible algorithm (i.e., $M_1$ 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 4.3: $M_2$ performs the same accesses as TA but one at a time and $M_3$ evaluates exhaustively using sorted accesses.

Different algorithms, by performing different accesses in different orders, incur different costs. For instance, $M_4$ can answer the same query, performing only a part of accesses that Algorithm TA performs (as we will see in Example 13). Our goal is thus to identify the cost optimal algorithm, among many possible algorithms. Toward the goal, we ask: How do we guarantee an algorithm space to include the optimal algorithm? To answer this question, we first formalize the notion of cost comprehensiveness (as Figure 4.1 illustrates): While a space $\Omega$ may not encompass the “universe” $U$ of all possible algorithms, it is comprehensive, with respect to some cost function, if any arbitrary algorithm $M$ in $U$ can find its “counterpart” $M'$ 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 $M'$ is at least as good as $M$. We formalize this notion of “comprehensiveness” below, which provides a foundation for finding such a space.
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 \( \mathcal{K} \); \( \mathcal{P} \leftarrow \mathcal{P} \cup \{A\} \);

\textbf{Figure 4.4} Algorithm “skeleton” \( \text{SEQ} \).

\textbf{Definition 2 (Cost Comprehensiveness):} A space \( \Omega \) is \textit{cost comprehensive} with respect to cost function \( \mathcal{C} \), if for any arbitrary algorithm \( \mathcal{M} \), there exists an algorithm \( \mathcal{M}' \in \Omega \) answering the same query with no more cost, \( i.e. \),

\[ \mathcal{C}(\mathcal{M}') \leq \mathcal{C}(\mathcal{M}). \]

\[ \square \]

In summary, while \( \mathcal{U} \) is our universe in principle, if some space \( \Omega \) satisfies cost comprehensiveness (Definition 2), it is sufficient to search in \( \Omega \) instead of \( \mathcal{U} \), \( i.e. \),

\[ \arg\min_{\mathcal{M} \in \mathcal{U}} \mathcal{C}(\mathcal{M}) = \arg\min_{\mathcal{M} \in \Omega} \mathcal{C}(\mathcal{M}) \text{ : What would then be such } \Omega \text{ for our objective of minimizing total access costs (Eq. 2.1)? As a key insight, observe Algorithm } \mathcal{M}_2 \text{ (Figure 4.3b) performing the exact same set of accesses as } \mathcal{M}_1 \text{ (Figure 4.3a) 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. 2.1), which aggregates the costs of all accesses. Generalizing the observation, just as } \mathcal{M}_1 \text{ has a sequential counterpart } \mathcal{M}_2 , \text{ any } \mathcal{M} \in \mathcal{U} \text{ has a sequential counterpart } \mathcal{M}', \text{ which performs the same accesses sequentially, such that } \mathcal{C}(\mathcal{M}') \leq \mathcal{C}(\mathcal{M}). \text{ 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\(^1\). More formally, we model such sequential algorithms in Figure 4.4: 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 \textbf{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 \( \mathcal{K} \).

\(^1\)While parallelization cannot contribute to our optimization objective of Eq 2.1 (or total resource usage), it may benefit elapsed time when sources can handle concurrent accesses, \( e.g. \), as Web sources typically do. Section 4.6 will show that such parallelization can successfully “build upon” our access minimization framework.
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 $\mathcal{SEQ}$, which we denote as $\mathcal{G}(\mathcal{SEQ})$, to search for the cost-optimal one $\mathcal{M}_{\text{opt}}$, with respect to the cost scenario, i.e.,

$$\mathcal{M}_{\text{opt}} = \arg\min_{\mathcal{M} \in \mathcal{G}(\mathcal{SEQ})} \mathcal{C}(\mathcal{M}).$$

(4.1)

While we have successful fulfilled our first objective, the “comprehensiveness” (as Section 6.1 motivated), we are now facing the challenge of the second objective: Is this new $\Omega = \mathcal{G}(\mathcal{SEQ})$ focused enough? Unfortunately, though comprehensive, $\mathcal{G}(\mathcal{SEQ})$ 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 are $m + m \cdot n = 500005$ different supported accesses to choose from. It is thus difficult to find an effective algorithm within $\mathcal{G}(\mathcal{SEQ})$: Our next goal is thus to refine the space to be as focused as possible, without compromising the comprehensiveness.

Towards the goal, it is critical to first decompose a top-$k$ query into “logical tasks” as building blocks, as Section 4.2 will develop, based on which we define a comprehensive and focused search space in Section 4.3.

### 4.2 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.

#### 4.2.1 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 \( D = \{u_1, \ldots, u_n\} \), any algorithm \( M \) must gather certain score information for each object \( u_j \), to determine the top-k. We can thus “compose” the work of \( M \) by a set of required scoring tasks, \( \{w_1, \ldots, w_n\} \). To define such tasks, let \( K = \{v_1, \ldots, v_k\} \) be the top-k answers (where each \( v_i \) represents some \( u_j \) from \( D \)). In this thesis, we assume applications require top-k answers to be completely evaluated, but our development can easily extend to partial evaluation. 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 \( u_j \in K \)) compute \( u_j \)'s complete score or else prove that it cannot score higher than \( v_k \) (the \( k^{th} \) answer).

**Definition 3 (Scoring Tasks):** Consider a top-k query \( Q = (F, k) \), with top-k answers \( K = \{v_1, \ldots, v_k\} \). The scoring task \( w_j \) for object \( u_j \) is:

1. for \( u_j \in K \): \( w_j \) must compute the final \( F[u_j] \) score; or
2. otherwise: \( w_j \) must indicate (by some partial scores) the maximal-possible \( F[u_j] \) score, tight enough to support that \( F[u_j] < F[v_k] \). \(^2\)

As a remark, note that these tasks are specified with given \( K \) (the top-k answers) and \( F[v_k] \) (the \( k^{th} \) 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 \( K \) and \( F[v_k] \) are known.

**Example 10 (Scoring Tasks):** Consider our running example \( Q_1 \) over Dataset 4 (Figure 4.2) \( D_4 = \{u_1, u_2, u_3\} \) (Figure 4.2): For \( k = 1 \), the answer is \( K = \{u_3\} \) with \( F[u_3] = .7 \) (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 \( u_3 \in K \), \( w_3 \) must gather all predicate scores—\( p_1[u_3] \) and \( p_2[u_3] \)—for computing \( F[u_3] \). Note \( w_3 \) can do so in various ways, e.g., by one sorted access \( s_{u_1} \) into \( p_1 \) (which hits \( u_3 \) and returns \( p_1[u_3] = .7 \)) and a random access \( r_{a_2}(u_3) \) (returning \( p_2[u_3] = .7 \)).

\(^2\)Recall that, to give deterministic semantics, we assume that there are no ties in \( F \) 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 only “truly comparable” algorithms returning the same results.

60
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 $\mathcal{F}[w_2] < \mathcal{F}[u_3] = .7$. To do so, $u_2$ can use, say, two sorted accesses $s_{a_1}$ into $p_1$, which return first $p_1[u_2] = .7$ 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 $\mathcal{F}[w_2] = \min(p_1[u_2], p_2[w_2])$, $\mathcal{F}[w_2]$ cannot be higher than $p_1[u_2]$, i.e., $\mathcal{F}[w_2] \leq .65 < .7.$ \hfill \q

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 $u_j$, i.e., 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 3, 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 inequity can be proved.

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

### 4.2.2 Identifying Unsatisfied Tasks

To focus query processing, it is critical to identify unsatisfied tasks to concentrate on.

However, during query processing, it is challenging to judge whether a task is satisfied, since $\mathcal{K} = \{v_1, \ldots, v_k\}$, which our task specification (Definition 3) 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 $\mathcal{K}$.
<table>
<thead>
<tr>
<th>OID</th>
<th>( p_1 )</th>
<th>( p_2 )</th>
<th>( F )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u_1 )</td>
<td>.65</td>
<td>( \leq .9 )</td>
<td>( \leq .65 )</td>
</tr>
<tr>
<td>( u_2 )</td>
<td>.6</td>
<td>.9</td>
<td>.6</td>
</tr>
<tr>
<td>( u_3 )</td>
<td>.7</td>
<td>( \leq .9 )</td>
<td>( \leq .7 )</td>
</tr>
</tbody>
</table>

**Figure 4.5** The score state of Example 11.

**Example 11 (Unsatisfied Tasks):** Consider \( Q_1 \) over Dataset 4 (Figure 4.2): Suppose, at some point, we have performed \( \mathcal{P} = \{ sa_1, sa_1, sa_2, ra_1(u_2) \} \), from which we 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 4.1), 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 4.5 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 \( F[u_1] \leq \min(.65, .9) = .65 \). Similarly, \( F[u_2] = .6 \) and \( F[u_3] \leq .7 \).

At this point, while we do not know what \( K \) will be (as Definition 3 requires), we can identify *at least* the scoring task \( u_3 \) for \( u_3 \) as unsatisfied, *no matter* what \( K \) is:

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

- if \( u_3 \notin K \): in this case, the top-1 is \( u_1 \) or \( u_2 \), with \( F \) scores of *at most* .65 and .6 respectively (Figure 4.5)—Thus, the top-1 score (i.e., \( F[u_k] \) in Definition 3) is at most .65.

Clearly, \( u_3 \) has *not* proved that \( F[u_3] \leq .65 \), since \( u_3 \) can score as high as .7.

As Example 11 hints, task \( w_j \) is unsatisfied, if \( u_j \) has “potential” to be in the top-\( k \) results \( K \). For such \( u_j \) (e.g., \( u_3 \)), regardless of what \( K \) 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 \( \mathcal{F} \mathcal{P}[u_j] \) 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., 
\[\mathcal{F}(x_1, \ldots, x_m) \geq \mathcal{F}(y_1, \ldots, y_m) \text{ when } \forall i: x_i \geq y_i;\] We thus can compute \(\mathcal{F}_\mathcal{P}[u_j]\) 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 4.1 discussed such “side-effects” of sorted accesses.) For instance, as Figure 4.5 shows, \(\mathcal{F}_\mathcal{P}(p_1, p_2)[u_1] = \min(p_1[u_1] = .65, \overline{p_2} = .9) = .65.\) Thus, formally, \(\mathcal{F}_\mathcal{P}(p_1, \ldots, p_m)[u_j] =
\[
\mathcal{F}\left(p_i = p_i[u_j] \quad \text{if } \mathcal{P} \text{ has determined } p_i[u_j] \quad \forall i \right) \\
          p_i = \overline{p_i} \quad \text{otherwise.}
\] (4.2)

Further, we focus on the current top-\(k\) objects by their potentials. Let \(\mathcal{K}_\mathcal{P} = \{v_1, \ldots, v_k\}\) be these current top objects ranked by their \(\mathcal{F}_\mathcal{P}\) scores. (To illustrate, in Example 11, \(\mathcal{K}_\mathcal{P} = \{u_3\}.\) 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 11 argued for \(u_3\) (an incomplete top-1), such \(v_j\) needs further accesses either way, by Definition 3: 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 \(v_1, \ldots, v_k\) 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., \(\mathcal{F}[v_j] = \mathcal{F}_\mathcal{P}[v_j]\). 2) Every object \(u_i \not\in \mathcal{K}_\mathcal{P}\), with the current ranking, has its maximal-possible score lower than the above exact scores, i.e., \(\mathcal{F}_\mathcal{P}[u_i] \leq \mathcal{F}[v_j]\). It follows that those \(v_j\) are the top-\(k\) answers, fully evaluated. Meanwhile, with these two observations, Definition 3 will declare all scoring tasks (either case) as satisfied. That is, checking from the “task” perspective, it is consistent to see that all tasks are fulfilled— thus query processing can indeed halt.

Theorem 5 states our results on identifying unsatisfied tasks.

**Theorem 5 (Unsatisfied Scoring Tasks):** Consider a top-\(k\) query \(Q = (\mathcal{F}, k)\) over \(\mathcal{D} = \{u_1, \ldots, u_n\}\). With respect to a set \(\mathcal{P}\) of performed accesses, let \(\mathcal{K}_\mathcal{P} = \{v_1, \ldots, v_k\}\) be the current top-\(k\) objects ranked by \(\mathcal{F}_\mathcal{P}[\cdot]\).

1. \(\forall v_j \in \mathcal{K}_\mathcal{P} \text{ s.t. } v_j \text{ has not been completely evaluated, its scoring task } w_j \text{ is unsatisfied.} \)
2. If all \( v_j \)'s are complete, then every scoring task \( w_j, \forall u_j \in \mathcal{D} \), is satisfied, and \( \mathcal{K}_p \) is the top-\( k \) results.

**Proof:** (1) If \( v_j \in \mathcal{K}_p \) 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 3, 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 \( \mathcal{F}_p[v_j] < \mathcal{F}[u] \), which in turn satisfy \( \mathcal{F}_p[v_j] < \mathcal{F}_p[u] \), as \( \mathcal{F}[u] \leq \mathcal{F}_p[u] \). Meanwhile, as \( v_j \in \mathcal{K}_p \), there are at most \( k - 1 \) objects \( u \) \( \mathcal{F}_p[v_j] < \mathcal{F}_p[u] \), a contradiction.

(2) If all \( v_j \)'s are complete, \( \mathcal{F}[v_j] = \mathcal{F}_p[v_j] > \mathcal{F}_p[u] \geq \mathcal{F}[u], \forall u \notin \mathcal{K}_p \), and thus \( \mathcal{K}_p = \mathcal{K} \).

With this, we can show that scoring task \( w_i \) is satisfied, for every \( u_i \in \mathcal{D} \).

\( \forall u_i \in \mathcal{K}_p = \mathcal{K} \): As every \( u_i \in \mathcal{K} \) has been completely evaluated, \( w_i \) is satisfied (by Definition 3, Case 1).

\( \forall u_i \notin \mathcal{K}_p = \mathcal{K} \): 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 3, Case 2).

We stress that Theorem 5 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 thus uniformly handle random and sorted accesses (and possibly beyond), despite the progressiveness and side-effects (Section 4.1).

### 4.3 Putting Together: Framework NC

This section develops a space that is both comprehensive and focused. Built upon our algorithm abstraction (Section 4.1) and task decomposition (Section 4.2), Section 4.3.1 first develops a framework NC which induces such an algorithm space. Section 4.3.2 then shows that the space induced is both comprehensive and focused.
Framework $\text{NC}(Q, \mathcal{D})$: Necessary Choices

Input: query $Q = (\mathcal{F}(p_1, \ldots, p_m), k)$, 
database $\mathcal{D} = \{u_1, \ldots, u_n\}$

Output: $K$, top-$k$ objects from $\mathcal{D}$ w.r.t. to $\mathcal{F}$

1) $\mathcal{P} \leftarrow \emptyset$; // accesses-so-far
2) $K_{\mathcal{P}} \leftarrow \{v_1, \ldots, v_k \mid \text{top-$k$ from } \mathcal{D} \text{ ranked by } \overline{\mathcal{F}}_{\mathcal{P}[]}\}$
3) while ($U \leftarrow \{v_j | v_j \in K_{\mathcal{P}}; v_j \text{ is incomplete}\}$)
4) $v_j \leftarrow \text{any object in } U$; // e.g., the highest-ranked
5) $N_j \leftarrow \{\text{sa}_i, \text{ra}_i(v_j) | p_i[v_j] \text{ is undetermined by } \mathcal{P}\}$;
   alternatives $\leftarrow N_j$;
6) Select access $A$ from alternatives; // access selection.
7) perform $A$; update $K_{\mathcal{P}}$; $\mathcal{P} \leftarrow \mathcal{P} \cup \{A\}$;
8) return $K = K_{\mathcal{P}}$;

Figure 4.6 Framework NC.

4.3.1 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 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 4.1). 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 4.2)-- that top-$k$ query processing is 
equivalent to fulfilling a set of (necessary and atomic) tasks $\{w_1, \ldots, w_n\}$. With this “task 
view,” during processing, when a set of accesses $\mathcal{P}$ has been performed, we can identify 
unsatisfied tasks, by Theorem 5. (When all tasks are satisfied, query processing can halt, as 
Theorem 5 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 $u_j$’s unevaluated predicates. (As Theorem 5 states, for such unsatisfied $w_j$, its object $u_j$ must be still incomplete.)

**Example 12 (Necessary Choices):** Continue our running example. Example 11 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 4.5 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 3). 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 11) has already one $sa_2$, the next $sa_2$ will return $u_1$ with score .8 (Figure 4.2). 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 $\{sa_2, ra_2(u_3)\}$. ■

**Definition 4 (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 $\mathcal{P}\}$. ■

Observe Figure 4.6 describing our framework NC: At each iteration, it identifies necessary choices, with Theorem 5 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 $\mathcal{F}_\mathcal{P}[\cdot]$. Some objects in $\mathcal{K}_\mathcal{P}$ may still be incomplete, which variable $U$ collects. As Theorem 5 specifies, there are two situations:

1. If $U = \phi$: As all top-$k$ objects are complete, Theorem 5 asserts no more unsatisfied tasks, which is thus the termination condition of NC: NC will break the while-loop (since $U = \phi$), and return $\mathcal{K}_\mathcal{P}$.
<table>
<thead>
<tr>
<th>step</th>
<th>$p_1$</th>
<th>$p_2$</th>
<th>$K_P$</th>
<th>alternatives</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>${u_3}$</td>
<td>$N_3={sa_1, sa_2, ra_1(u_3), ra_2(u_3)}$</td>
<td>$sa_1$</td>
</tr>
<tr>
<td>2</td>
<td>0.7</td>
<td>1</td>
<td>${u_3}$</td>
<td>$N_3={sa_2, ra_2(u_3)}$</td>
<td>$ra_2(u_3)$</td>
</tr>
</tbody>
</table>

Figure 4.7 Illustration of NC.

2. Otherwise: Since $U \neq \emptyset$, there are incomplete top-$k$ objects. Any such object $v_j$ corresponds to an unsatisfied task $w_j$, by Theorem 5. NC arbitrarily picks any such $v_j$ (say, the highest-ranked one), and constructs the necessary choices $N_j$ (by Definition 4) as alternatives for selecting further access.

Note that NC essentially relies on Theorem 5 to isolate a set of necessary choices. Theorem 5 enables an effective way to search for necessary choices, by maintaining $K_P$, 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) $K_P$ with some deterministic tie-breaking order (Section 4.1). 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 effectively $\mathcal{F}[u_i] > \mathcal{F}[u_j]$.

Observe that, at each iteration, there may be multiple incomplete $v_j$ in $K_P$. We stress that NC can simply choose any such $v_j$ to proceed. Each $v_j$ designates an unsatisfied task $w_j$– Any such $w_j$ must be further carried out, and is thus “equally” necessary (Section 4.2.1). More precisely, an unsatisfied task $w_j$ will induce a set $N_j$ of necessary choices– with a desired completeness property– As Section 4.3.2 will discuss, with this completeness, any $N_j$ can guarantee the comprehensiveness of NC. Example 13 illustrates how NC works.

Example 13 (Framework NC): Figure 4.7 shows the execution of algorithm $M_4$ (Figure 4.3d) that NC can generate: Initially, at Step 1 (Figure 4.7), as all the maximal-possible scores tie as 1.0, $K_P$ is set to $\{u_3\}$ (by the highest OID, our tie-breaker), which induces alternatives $= N_3$. According to NC, $M_4$ then Select an access, $sa_1$ in this case, among the alternatives, which returns $p_1[u_3] = .7$ (see Figure 4.2) and lowers $p_1$ to .7.

At Step 2, as all the maximal-possible scores tie as .7, $u_3$ remains as the top in $K_P$. However, $u_3$ now induces a smaller $N_3$, with accesses only for its unevaluated predicate $p_2$. $M_4$ then
Select \( ra_2(u_3) \), which returns \( p_2[u_3] = 0.7 \) and completes \( u_3 \) with \( \mathcal{F}[u_3] = 0.7 \). Since \( K_P \) with \( u_3 \) as the top-1 is now fully complete, according to NC, \( M_4 \) will halt, with total accesses
\[
P(M_4) = \{s_{a_1}, ra_2(u_3)\}.
\]

### 4.3.2 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. In particular, we define a set of alternatives as complete with respect to “accesses-so-far” \( P \), if any algorithm \( M \) that has performed \( P \) must also perform at least one access from alternatives. Thus, SEQ is trivially complete: If \( P \) is not sufficient to determine query answers, any algorithm having done \( P \) must continue with at least one more access—which by definition must be among all supported accesses.

In fact, while NC focuses on a much smaller alternatives, it is still comprehensive. To see why, note that NC identifies a set of necessary choices \( N_j \)—which, by Definition 4, contains all accesses that can contribute to the unsatisfied task \( w_j \). Since \( w_j \) is necessary (Section 4.2.1), at least one access in \( N_j \) must be further executed, or \( w_j \) cannot be satisfied and thus the query cannot be answered—(For instance, for \( N_3 \) in Example 12, if neither \( s_{a_2} \) nor \( ra_2(u_3) \) is executed after \( P \), \( w_3 \) will remain unsatisfied.) Thus, \( N_j \) is complete, with respect to accesses-so-far \( P \). This completeness holds for the necessary choices of any unsatisfied task—since any such must be fulfilled, sooner or later.

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 2), as Theorem 6 below states. With this guarantee, it is sufficient to search only within NC for an optimal algorithm.

**Theorem 6 (NC Comprehensiveness):** For any algorithm \( M_1 \) with an access cost \( C_1 \) with respect to the cost model \( C \) (Eq. 2.1), there exists an algorithm \( M_2 \) in \( \mathcal{G}(\text{NC}) \) with cost \( C_2 \), such that \( C_2 \leq C_1 \).
Proof: Consider any query processing by $M_1$ (for some query $Q$ over database $D$). We will show the generality of NC by constructing an algorithm $M_2$ in Framework NC for the same processing, such that $M_2$ costs no more than $M_1$. Let $P_1$ be the total accesses that $M_1$ has performed, i.e., $P(M_1) = P_1$. Since $M_2$ follows the interactive framework (Figure 4.7), let $P^j_2$ be the accesses of $M_2$ before the $j^{th}$ iteration; initially, $P^1_2 = \phi$. Similarly, let alternatives$^j$ be alternatives of $M_2$ at $j^{th}$ iteration.

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

- $L_1$: $P^j_2 \subseteq P_1, \forall j$.
- $L_2$: alternatives$^j \cap (P_1 - P^j_2) \neq \phi, \forall j$.

Note that, by $L_1$, algorithm $M_2$ incurs no more access than $M_1$, when $M_2$ halts at some iteration $j$ (denoted as $M^j_2$): $P^j_2 \subseteq P_1$. Note, this immediately implies that $C(M^j_2) \leq C(M_1)$ as well, because our cost function (Eq. 2.1) is “monotonic” to accesses performed: If $M_1$ performs more times of every kind of access than $M^j_2$, then $M_1$ will have an overall higher cost, i.e., $P(M^j_2) \subseteq P(M_1) \implies C(M^j_2) \leq C(M_1)$. To complete the proof, we now show by induction that $L_1$ and $L_2$ hold; we will also specify the “behavior” of $M_2$ for each iteration, to show how it can be constructed in the NC framework.

- $j = 1$: $L_1$ is trivial, since initially $P^1_2 = \phi$.

Consider $L_2$: We note that, by definition of the Framework NC, alternatives$^j$ is “complete” - that any algorithm (like $M_1$) that has performed $P^j_2$ must have performed ‘in addition’ some access $A$ among alternatives$^j$. Thus, as $M_1$ has performed $P^1_2$ (trivially, since $P^1_2 = \phi$), it must have performed access $A \in$ alternatives$^1$ in addition. That is, $A$ is in both alternatives$^1$ and $P_1 - P^1_2$, and thus $L_2$ holds.

- $j = k$: As the induction hypothesis, assume for $j = k$, the lemmas hold.

What should algorithm $M_2$ do in each iteration? We now construct $M_2$ for iteration $k$: If $M_2$ exhausts $P_1$, which provides enough information to answer $Q$, $M_2$ halts right before this iteration. Otherwise, NC requires that $M_2$ select one access from alternatives$^k$ to continue: We will let $M_2$ choose an access $A^k$ that is also in $P_1 - P^k_2$. Such $A^k$ must exist by $L_2$, i.e., $A^k \in$ alternatives$^k \cap (P_1 - P^k_2)$. 
• $j = k + 1$: First, $L_1$ 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.

Second, $L_2$ holds: By $L_1$ (just proven 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 $\text{alternatives}^{k+1}$, $\mathcal{M}_1$ must have performed, ‘in addition’ to $\mathcal{P}_2^{k+1}$, some access $A \in \text{alternatives}^{k+1}$. That is, $A$ is in both $\text{alternatives}^{k+1}$ and $\mathcal{P}_1 - \mathcal{P}_2^{k+1}$, and thus $L_2$ 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 4.4 will discuss optimization techniques for finding $\mathcal{M}_{\text{opt}}$ such that, further refining Eq. 4.1:

$$\mathcal{M}_{\text{opt}} = \arg\min_{\mathcal{M} \in \mathcal{G}(\text{NC})} C(\mathcal{M}).$$

(4.3)

### 4.4 Search: Dynamic Optimization

In this section, we discuss how to actually optimize top-$k$ queries, using Framework NC in Section 4.3. 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. 4.3. 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 than the space of arbitrary algorithms, $\mathcal{G}(\text{NC})$ is still too large for exhaustive search. We thus design a suite of “systematic” heuristics to reduce the space. Similarly, Boolean query optimization relies on systematic heuristics for effective search, such as focusing only on linear joins.

2. **Search**: Within the space identified, we design effective optimization schemes focusing search on promising algorithms. Similarly, Boolean optimization focuses its search on plans enumerated in particular ways, *e.g.*, by dynamic programming.
3. **Cost estimation**: As a ground to compare algorithms in the space, the optimizer must be able to estimate the cost of each algorithm. Our cost estimation extends the insight of its Boolean counterpart, as we will discuss in Section 4.4.3.

### 4.4.1 Space Reduction

While NC help optimization by inducing a focused algorithm space, it is still 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” heuristics. These heuristics contribute in two ways: First, they reduce the space significantly, while still retaining 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 heuristics for optimization:

First, we choose to focus only on **SR** algorithms (for sorted-then-random), which perform all $sa_i$ on predicate $p_i$, if done at all, before any any $ra_i(\cdot)$. Lemma 3 states that, for any top-k algorithm, we have its **SR**-counterpart gathering the same score information, with no more cost.

**Lemma 3 (SR-counterpart):** For any algorithm $M_1 \in \mathcal{G}(NC)$, there exists its **SR**-counterpart $M_2$ with no more cost, i.e., $\mathcal{C}(M_2) \leq \mathcal{C}(M_1)$.

**Proof:** We prove by constructing **SR**-counterpart $M_2$ of $M_1$ with no more cost. Let $\mathcal{P}_1^i$ be the total accesses that $M_1$ has performed on $p_i$, i.e., $\mathcal{P}(M_1) = \sum_i \mathcal{P}_1^i$. That is, $\mathcal{P}_1^i$ should be sufficient for collecting the same information as $M_1$ on predicate $p_i$. We thus construct $M_2$ 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 $sa$ in $\mathcal{P}_1^i$ and then every $ra$ 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 proceeding sorted access. Consequently, $\forall i: \mathcal{P}_2^i \subseteq \mathcal{P}_1^i$, and thus $M_2$ terminates as early as $M_1$, if not earlier, i.e., $\mathcal{C}(M_2) \leq \mathcal{C}(M_1)$. ■

Lemma 3 allows us to reduce our plan space NC by focusing only on the subset of **SR** algorithms, i.e., **SR**-subset. However, how good is this heuristics? Will we miss the actual optimal algorithm, $M_{opt}$ by such reduction? By Lemma 3, we can conclude that the reduction has no “loss” of optimality as long as the **SR**-counterpart of $M_{opt}$ is still in NC— a property

71
<table>
<thead>
<tr>
<th>step</th>
<th>$\overline{p_i}$</th>
<th>$\overline{p_j}$</th>
<th>$K_p$</th>
<th>alternatives</th>
<th>Select</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>1</td>
<td>1</td>
<td>${u_3}$</td>
<td>$N_3 = {sa_1, sa_2, ra_1(u_3), ra_2(u_3)}$</td>
<td>$sa_1$</td>
</tr>
<tr>
<td>2.</td>
<td>0.7</td>
<td>1</td>
<td>${u_3}$</td>
<td>$N_3 = {sa_2, ra_2(u_3)}$</td>
<td>$sa_2$</td>
</tr>
<tr>
<td>3.</td>
<td>0.7</td>
<td>0.9</td>
<td>${u_3}$</td>
<td>$N_3 = {sa_2, ra_2(u_3)}$</td>
<td>$sa_2$</td>
</tr>
<tr>
<td>4.</td>
<td>0.7</td>
<td>0.8</td>
<td>${u_3}$</td>
<td>$N_3 = {sa_2, ra_2(u_3)}$</td>
<td>$ra_2(u_3)$</td>
</tr>
</tbody>
</table>

**Figure 4.8 Illustration of SR/G heuristics.**

**Procedure** $Select$ (alternatives, $\Delta, H$):
if $\exists sa_i \in$ alternatives such that $\overline{p_i} > \delta_i$:

$A \leftarrow sa_i$;
else if $\exists ra_i(u) \in$ alternatives such that $p_i = next(u, H)$:

$A \leftarrow ra_i(u)$;

**Figure 4.9 Select with SR/G heuristics.**

we call “SR-inclusion”. We believe $SR$-subset reduction is at least a good heuristics with little “loss” of optimality, as $SR$-inclusion does hold in our empirical observations, though we don’t have a formal proof.

Second, we assume that $m$ on every object follows the same “global” order $H$. That is, when multiple random accesses exist in alternatives, we follow some particular order $H$ (given by the optimizer; See Section 4.4.2) to choose which to perform. To illustrate, supposing necessary choices are alternatives $= \{ra_i(u_1), ra_j(u_1)\}$ given $H = (p_i, p_j)$, we pick $ra_i(u_1)$ first as the next unevaluated predicate of $u_1$ is $p_i$ according to $H$, which we denote as $next(u_1, H) = p_i$.

This heuristics has been first studied in [12] (which focuses only $m$ “probes”, unlike our general optimization). As [12] reported, such global scheduling achieves comparable optimization result, while significantly reducing the complexity.

By focusing on the above two heuristics, we propose Framework NC with SR/G ($SR$-subset and Global scheduling) heuristics. These heuristics customize the $Select$ routine of NC as Figure 4.9 shows: Now the selection is more focused, guided by two parameters $\Delta = (\delta_1, \ldots, \delta_m)$ and $H = (p_1, \ldots, p_m)$, which will be determined by the optimizer (Section 4.4.2). In essence, $Select$ chooses $sa$ whenever there exists $sa_i$ which hasn’t reached the suggested depth $\delta_i$, i.e., $\overline{p_i} > \delta_i$. Otherwise, it performs $m$ in alternatives, by picking the next unevaluated predicate (according to $H$). Example 14 illustrates how these heuristics actually
work with our running example. (For the sake of presentation, NC from here on refers to the framework with SR/G heuristics.)

**Example 14 (SR/G heuristics):** Consider our running example $Q_1$ on Dataset 1: Figure 4.8 illustrates how SR/G heuristics 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 $sa$ 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, 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 heuristics 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 $(\Delta_{opt}, \mathcal{H}_{opt}) = \arg\min_{\Delta, \mathcal{H}} \mathcal{C}(\Delta, \mathcal{H})$.

4.4.2 Search

Toward identifying the optimal algorithm $(\Delta_{opt}, \mathcal{H}_{opt})$, we first approximate the problem by identifying $\Delta_{opt}$ and $\mathcal{H}_{opt}$ in turn:

- **$\Delta$-optimization:** We first identify the optimal depth $\Delta_{opt}$, with respect to some initial schedule $\mathcal{H}_0$, i.e.,

$$\Delta_{opt} = \arg\min_{\Delta} \mathcal{C}(\Delta, \mathcal{H}_0).$$

- **$\mathcal{H}$-optimization:** We then identify the optimal scheduling $\mathcal{H}_{opt}$ with respect to $\Delta_{opt}$ identified.

For $\mathcal{H}$-optimization, we use predicate scheduling framework for MPro (Section 3.1.3) as a building block. Thus, in this section, we focus on $\Delta$-optimization: As Example 15 will illustrate, $\Delta$-optimization is specific to runtime factors, e.g., score functions, predicate score distributions, and cost scenarios.
Example 15 (Δ—Optimization Possibilities): To illustrate, we continue Example 14 with a different depth configuration $\Delta_2 = (0.8, 1)$. In fact, $\Delta_2$ generates the algorithm illustrated in Figure 4.7: 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 4.8), a focused configuration $\Delta_2 = (0.8, 1)$ requires only two accesses (Figure 4.7). 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 $\Delta$—space, i.e., $m$-dimensional space of $\delta_1 \times \ldots \times \delta_m = [0 : 1]^m$. We first discuss an exhaustive search scheme Naïve, which will be used as a baseline for comparison (Section 4.6).

We then enhance the scheme with more informed (either query-driven or generic) search.

(Scheme Naïve:) Naïve simply explores the whole $\Delta$—space by meshing it into a finite set of grid points $P$. Then, for every grid point $p \in P$, it estimates the cost (See Section 4.4.3) of every algorithm $(\Delta_p, \mathcal{H}_0)$ and identifies the minimal-cost configuration among them. Though simple, Naïve obviously suffers from scalability and performance limitations, especially when $\Delta$—space explodes for large $m$. We thus enhance Naïve to systematically focus on a promising subset of $P$, as follow.

(Scheme Strategies:) Strategies enhances Naïve approach by applying query-driven “strategies” in the search for $\Delta_{opt}$. As illustrated in Example 15, a particular scoring function often implies a particular “best strategy” to narrow down search, e.g., parallel configurations for $avg$ and focused configurations for $min$. Thus, Scheme Strategies focuses its search on some configurations corresponding to the given strategy.

(Scheme HClimb:) As an alternative to query-specific Strategies scheme, one can apply a generic informed search to enhance Naïve scheme. For instance, one can apply hill climbing scheme: From a random point, HClimb simply searches towards its neighboring configuration with less estimated cost, until it reaches the minimum. The scheme is typically enhanced with multiple random starting points, to avoid being stuck at the local minimum.

In particular, our experiments in Section 4.6 will adopt HClimb as an optimization scheme, which is evaluated to be the most effective from our experiments.
4.4.3 Cost Estimation

Finally, we discuss how to estimate the cost of algorithms in $\Delta$-space. To motivate, recall the cost estimation for Boolean queries: First, optimizer estimates the selectivity of each predicate using some “statistical samples”, e.g., histograms. Second, it then estimates their “aggregate” effect, from which the overall cost can be computed: The aggregated effect is computed “analytically” in Boolean queries, as predicates are composed by the known set of relational operators, e.g., $\sigma$ or $\Join$. For instance, in a simple conjunctive query, the aggregate selectivity is simply the product of selectivities, assuming predicate independence.

For top-$k$ queries, we extend the same intuition in the following ways: First, we generalize Boolean selectivity into the selectivity of probabilistic score “distributions”, which can be similarly estimated from statistical samples. Second, we estimate the aggregate selectivity of predicates, which is challenging for top-$k$ queries: As predicates are aggregated by arbitrary function $F$, the aggregate effect cannot be quantified by analytic composition as in Boolean optimization, but only by “simulation runs”: Simulation is essentially a mimic of the actual execution on sample objects. In particular, we perform a simulation run on the samples, transforming a top-$k$ query on the database into a top-$k'$ query on the samples. The retrieval size $k'$ is determined in proportional to the sample size $s$, i.e., $k' = \lceil k \cdot \frac{s}{\bar{n}} \rceil$.

In principle, samples can be obtained from online sampling, or built offline (e.g., based on a priori knowledge on predicate score distribution.) However, when samples are unavailable or too costly to obtain online, one can generate “dummy” samples based on the “assumed” distribution (e.g., uniform)—Though such samples cannot represent actual score distributions, they help optimize for other important aspects, such as $F$ or $k$. While our optimizer will certainly benefit from accurate samples, Section 4.6 will implement our optimization framework using dummy samples, to validate our framework in the worst case scenario.
4.5 Unification and Contrast

With general optimization, NC should in principle unify algorithms for specific scenarios—We thus study how NC\(^3\) in fact unifies specific algorithms, by generating similar behaviors, and further contrast them, by identifying those “ungeneralizable” behaviors.

As middleware algorithms generally assume no-wild-guesses [19], we first describe how NC handles 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 \(\mathcal{F}[\text{unseen}] = \mathcal{F}(\overline{\mathcal{P}}_1, \ldots, \overline{\mathcal{P}}_m)\). This virtual object needs special handling, as Figure 4.10 shows with query \(Q_1\): First, initially all objects are unseen, so NC now initializes \(\mathcal{K}_P\) 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 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).

### 4.5.1 Algorithm TA

We now observe how NC adapts to TA scenarios. As Figure 2.2 summarized, TA aims at access scenarios where sorted and random access have uniform unit costs, i.e., \(\frac{\overline{\mathcal{P}}_r}{\overline{\mathcal{P}}_s} \approx 1\). In brief, TA works as follows:

---

3For notational simplicity, we use NC interchangeably as an abstract framework and as the optimal algorithm generated.
• Perform sorted accesses on \( m \) predicates in parallel, or \textit{equal-depth}^4. As an object \( u \) is seen from any sorted access, perform \( ra_j(u) \) \textit{exhaustively} for every unevaled predicate \( p_j \) to compute its final score \( F[u] \). Add \( u \) to \( \mathcal{K} \), if it is one of the \( k \) highest so far.

• Let threshold \( \theta_{TA} = F(p_1, \ldots, p_m) \). As soon as \( \mathcal{K} \) has \( k \) objects with scores no less than \( \theta_{TA} \), stop and output \( \mathcal{K} \).

In essence, TA can be characterized by three “behaviors”: (1) \textit{equal-depth-sorted-access}: At each iteration it performs sorted accesses to all predicates. (2) \textit{exhaustive-random-access}: It then does exhaustive random accesses on every seen object. (3) \textit{early-stop}: It terminates as soon as the stop condition \( \forall v \in \mathcal{K} : F[v] \geq \theta_{TA} \) is satisfied. So, would NC adapt to uniform scenarios by dynamic optimization and generate similar behaviors?

\textbf{Unification:} In “symmetric” cases (which will be clear later), which TA’s behaviors are “optimized for”, NC will indeed generate TA: We illustrate with a scenario \( S_1 \) with scoring function \( F = \text{avg}(p_1, p_2) \), 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 4.11(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, .83). To compare, the figure also marks the depth TA reaches (by an oval) at (.84, .84).^5

Observe that the two algorithms are indeed almost identical: (1) Both perform \textit{equal-depth-sorted-access} up to similar depths. (2) By accessing the same depths, they will both see the same set of objects—Since NC does not use exhaustive random access, it will only perform less random accesses than TA, \textit{e.g.}, NC slightly outperforms TA (by 1%) in

\footnote{Note that the \textit{depth} of sorted access, in this context of TA, refers to the \textit{number} of objects accessed, instead of the score reached.}

\footnote{This figure can be viewed or printed in color, for better visibility.}
Figure 4.11(a). (3) The output \( K \) of NC shares the same *early-stop* condition as TA: Since \( \mathcal{F}[\text{unseen}] = \theta_{TA} \) (by definition) and \( \text{unseen} \notin K \), it follows that 
\[ \forall v \in K : \mathcal{F}[v] \geq \mathcal{F}[\text{unseen}] = \theta_{TA}. \]

**Contrast:** However, NC contrasts with TA by being able to adapt: Even among uniform scenarios, in the “asymmetric” cases, TA’s characteristic behaviors cannot adapt well.

1. *Equal-depth-sorted-access* is not desirable, in scenarios when the optimal depth is not equal across predicates—*e.g.*, for \( \mathcal{F} = \min(p_1, p_2) \), *focused* sorted access is more effective (Example 15).

2. *Exhaustive-random-access* is not desirable—As contrasted above, by scheduling both sorted and random accesses, NC performs less random accesses.

3. *Early-stop* is not desirable, if performing *deeper* sorted access can trade those random accesses to follow and thus reduce the total cost—*i.e.*, trade-off exists between deeper sorted accesses and more random accesses.

In fact, in such scenarios, NC will adapt beyond TA and thus generate a rather different algorithm. To contrast, Figure 4.11(b) shows scenario \( S_2 \) with \( \mathcal{F} = \min \) (and otherwise the same as \( S_1 \)). Observe NC and TA differ significantly—NC focuses sorted accesses \( p_1 \) with \( \Delta_{opt} = (0.80, 1) \), while TA performs equal-depth sorted access up to \((0.88, 0.88)\). Observe also their cost difference is significant as well—NC saves access cost by 30% from TA, by focusing sorted accesses.

For a closer observation, Figure 4.12 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 4.12(a) first considers scenario \( S_1 \), which is rather favorable to TA (as explained above). Overall, both algorithms behave similarly by performing sorted accesses up to similar depths, but TA performs significantly more random accesses due to its exhaustive random access and early stop behaviors. (In comparison, NC indeed goes deeper in sorted accesses to trade random accesses.) To contrast, as “asymmetric” cases, Figure 4.12(b) considers scenarios that introduce *asymmetry*, one at a time, to \( S_1 \) as follow:

- **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 accesses 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 accesses on $p_2$ (which is more “selective” to distinguish objects by scores).

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

### 4.5.2 Algorithm MP$	ext{Pro}$

We next compare NC with MP$	ext{Pro}$ developed for a simpler $r$-only scenario (similar to Upper-Subset [5]). Consider $\mathcal{F}(p_1, \ldots, p_m, p_{m+1}, \ldots, p_n)$. We distinguish two types of predicates: While $p_{m+1}, \ldots, p_n$ are simply *ordinary* (or “indexed”) predicates, the other group $p_1, \ldots, p_m$ are *expensive* predicates—They are “probe only” or *m-only*.

- $\forall p_i \in \{p_1, \ldots, p_m\}$: $p_i$ supports $m$ only with unit cost $cr_i$; thus $cs_i = \infty$.
- $\forall p_i \in \{p_{m+1}, \ldots, p_n\}$: $p_i$ supports both $m$ and $sa$, with unit cost $cr_i$ and $cs_i$ respectively.

MP$	ext{Pro}$ aims at minimizing $m$’s, or per-object “probes,” on expensive predicates. In brief, it works as follows:

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

---

![Graph](image_url)

**Figure 4.12** Comparison of TA and NC.
2. Sort all objects $u$ by their maximal-possible score $F[u]$ (with respect to its evaluated predicates). Let $K_P$ be the current top-$k$ objects.

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

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

**Unification:** As MPro aims at minimizing random accesses, i.e., $m$’s, 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 $F[\text{unseen}] = F(1, \ldots, 1, \overline{p_{m+1}}, \ldots, \overline{p_n})$, 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 $K_P$, and stopping when $K_P$ completes. However, for such $v_j$, MPro constructs a set of $P_j$—with only $m$ 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 $sa$ and $m$ (line 5, Figure 4.6). As these probe-only predicates do not support $sa$’s, such “special restrictions” are naturally captured by the cost-based framework simply by setting $cs_i = \infty$. The optimizer (Section 4.4) will then algorithmically “ignore” the $sa$’s, by configuring the $sa$ depths as $\Delta$: $(\delta_1=1, \ldots, \delta_{m}=1)$—i.e., no $sa$ 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, both algorithms differ fundamentally for the “ordinary” predicates with both $sa$ and $ra$. 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 $p_{m+1}, \ldots, p_n$, MPro will suffer all the contrasts as Section 4.5.1 discussed: equal-depth $sa$ into these predicates, exhaustive-$m$, and early-stop.

**Remark:** Although correctly unified for the “probe-only” special-case, NC has generalized beyond MPro significantly—by generally handling arbitrary accesses, or in particular, both $sa$ and $m$. Such a generalization is non-trivial—Essentially, As Section 2.2 identified, $sa$ is fundamentally different with its progressiveness and side-effects. By limiting to only $m$’s, MPro removes the difficult issues of defining a complete framework: To illustrate, consider a $ra$-only setting, when some $v_j \in K_P$ is incomplete, and thus with further accesses, say, $P_j = \{ra_1(v_j), ra_2(v_j)\}$. To contrast, in its $sa$-also counterpart, by adding $sa$’s, the further accesses are $N_j = \{sa_1, ra_1(v_j), sa_2, ra_2(v_j)\}$. We identify two different properties:

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

2. **$sa$ progressiveness:** In $ra$-only, for $v_j$ just picked, given a 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 $sa$-also, it is not clear what such schedule is, or what to schedule exactly: As every $sa$ can repeat for progressive accesses, there are generally an infinitely number of such schedules; e.g., for $N_j$: $(sa_1, sa_2, ra_2(v_j), ra_1(v_j))$, $(sa_1, sa_1, sa_2, ra_2(v_j)$, $ra_1(v_j))$, etc.

Thus, $ra$-only essentially reduces the optimization issue (to a “barebone” of finding $\mathcal{H}$): Properties 1 and 2 together univocally determine a required probe on a particular $v_j$ for a particular $p_i$. (Such required probes are the notion of necessary probe in [12].) In contrast, for
<table>
<thead>
<tr>
<th>retrieval size ($k$)</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of predicates ($m$)</td>
<td>2</td>
</tr>
<tr>
<td>predicate score distribution</td>
<td>uniform</td>
</tr>
<tr>
<td>database size ($n$)</td>
<td>10000</td>
</tr>
</tbody>
</table>

**Figure 4.13** Default setting for synthetic dataset.

our general **also** framework, neither holds true. Our notion of **necessary choices** handles both $m$ and $sa$; it essentially hinges on its completeness to guarantee a general space (Theorem 6), **regardless** of side-effects and progressiveness.

### 4.6 Experiments

This section reports our experiments. Although Web scenarios are our immediate focus, our access minimization framework is clearly applicable to any middleware settings where accesses dominate the cost. Our goal is thus two-fold: First, in Section 4.6.1, we quantify the absolute performance of NC over actual Web scenarios to validate its practicality. Second, in Section 4.6.2, we study its performance over a wider range of middleware settings, by simulating over extensive synthetic scenarios. As baselines for comparison, we use TA and CA [19], which are the most representative top-$k$ middleware algorithms. Our experiments were conducted with a Pentium III 933MHz machine with 256M RAM, using our implementation of NC in Python.

#### 4.6.1 Real Web Scenarios

This section studies the “absolute” performance of NC over real-life Web sources. As Web sources typically handle concurrent accesses, we first develop parallelization of NC. We then report the experiments in Section 4.6.1.2.

#### 4.6.1.1 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 $\mathcal{K}_P$ 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 $\text{Select}$ (Figure 4.9) picks a sorted access $sa_i$ as the next access, when the “last-seen” score $\overline{p_i}$ from the preceding $sa_i$ has not reached the suggested depth $\delta_i$, i.e., $\overline{p_i} > \delta_i$. Note the up-to-date $\overline{p}$ is not known until all outstanding sorted accesses complete.

However, as waiting to get the exact $\overline{p}$ defeats the whole purpose of concurrent accesses, we continue with an estimated $\overline{p}$ instead, by computing its expected decrement $D_i$: Assuming $d_i$ is the expected decrement of $\overline{p}$ 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 [19]: At each iteration, TA performs sorted accesses to all $m$ predicates in parallel and completely evaluates the objects seen (up to $m - 1$ random accesses per each of the $m$ objects seen). Consequently, by issuing such accesses asynchronously, TA can overlap up to $m + m(m - 1) = m^2$ accesses. To fairly compare with these other algorithms, our experiments next will bound the concurrency of NC using their natural concurrency, i.e., setting $c = m^2$.

### 4.6.1.2 Results

This section evaluates NC over actual Web sources. In particular, we experiment with the “travel-agent” scenarios $Q_1$ and $Q_2$ (Examples 1 and 2) as our benchmark queries. To more extensively evaluate, we run each query combined with different scoring functions, i.e., $\text{min}$ and $\text{avg}$. We use the real Web sources suggested in Figure 1.1 to access the restaurants and hotels in Chicago (by issuing an additional Boolean selection “city=Chicago”). As these sources allow sorted access only in a small chunk of “batches” (e.g., per page of 25 objects), we regard a batch access as a single sorted access. For simplicity, the predicates are evaluated by linearly normalizing the corresponding attribute value into $[0:1]$, e.g., $\text{rating}$ for a two-star hotel in the five-star rating system will be evaluated as $\frac{2}{5} = 0.4$. 

83
As metrics, we use both the total access cost and actual elapsed time, to capture two different performance aspects— the resource usage and processing time respectively. The total access cost is measured by adding up the latency of all accesses (as in Eq. 2.1), while the elapsed time simply measures the processing time (including local computation and optimization time). Note, in the presence of concurrency, the elapsed time is typically shorter, by overlapping some high-latency accesses. Using these metrics, we compare NC with TA. Note CA is not applicable for our benchmark queries, as it requires a ratio $h$ of sorted versus random access costs, fixed for all predicates (i.e., $h = cr_i/c_{si}, \forall p_i$). As Figure 1.1 shows, this ratio is either varying (for $Q_1$) or 0 (for $Q_2$). (In Section 4.6.2, we will compare with CA over synthetic settings where $h$ is clearly defined.)

Figures 4.14(a) and (b) compare the total access cost of TA and NC for query $Q_1$, when $F=min$ and $F=avg$ respectively. Observe that NC significantly outperforms TA, as retrieval size $k$ (x-axis) increases: For instance, when $k = 500$ in Figure 4.14(a), the access cost (y-axis)
of NC is 42 seconds, which saves 80% from that of TA (192 seconds). In fact, NC outperforms TA by dynamically adapting to this scenario with expensive random accesses: In particular, NC performs deeper sorted accesses than TA, to trade random accesses with cheaper sorted accesses.

Similarly, Figures 4.14(c) and (d) compare the total access cost for query Q2. Again, the dynamic optimization enables NC to outperform TA significantly, e.g., when k = 10 in Figure 4.14(c), NC saves up to 66% from the access cost of TA. However, in contrast to Q1, 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 15), while TA still performs sorted accesses to every predicate with no adaptation. Note that, these observations are consistent with our remarks in Section 4.5.

Finally, Figure 4.15 compares the elapsed time of Q1 and Q2, when $F=\text{min}$. Comparing the access cost and elapsed time of Q1 (i.e., Figure 4.14a and Figure 4.15a), 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 (which has inherent parallelism). The same observation holds true for Q2 as well, except when the suggested depth of NC is too shallow to benefit from concurrency: 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. Meanwhile, TA exploits concurrency to overlap three parallel sorted accesses and performs as well in terms of the elapsed time.

4.6.2 Synthetic Middleware Scenarios

While Section 4.6.1 validates NC in real Web scenarios, some questions remain: Will NC perform as well, in a wider range of middleware access scenarios? Will the access costs saved justify the computational optimization overhead of NC, even when accesses are not as expensive as Web queries? To answer these questions, this section studies the performance of NC in a broader context of middlewares, by simulating over extensive synthetically generated scenarios.

In particular, we study the performance of NC over 1000 synthetic middleware scenarios, by varying the followings (over the default setting in Figure 4.13): We randomly generate the
unit cost of access $cr_i$ 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 $cr_i > cs_i$.) Further, we use $wavg$ (weighted average) as scoring function, which is commonly used in top-$k$ experimental settings. To simulate different relative significance of predicates, we randomly generate weight $w_i$ in $[1:100]$ for each configuration.

Figure 4.16(a) compares the total access cost (as the total number of units) of the three algorithms. First, we observe their average costs over all 1000 random configurations: Even as a general-purpose algorithm, NC still outperforms TA and CA by 17% and 14%. Second, for closer observation, we divide the 1000 settings into the three groups in which each of the three algorithms performs the best: In particular, into groups of 44, 218, and 738 configurations in which TA, CA, and NC performed the best respectively. Observe that, NC outperforms TA and CA in the majority of settings, i.e., in 74% (738/1000), and adapts as the second best in the rest 26% of settings.

Since the metric in Figure 4.16(a) does not reflect the computation overhead of NC, we further analyze the trade-off of the saving and overhead. In particular, as Figure 4.16(a) reports, overall, NC achieves a saving of $116210-100730=15480$ units from CA. We then measure the average computational overhead. Since the saving of access cost will have different impacts depending on the actual time unit $t$, Figure 4.16(b) compares this saving and overhead, with respect to different unit time $t$. (Note we compare only with CA, as it outperformed TA in Figure 4.16a.) 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.
CHAPTER 5

Integrating with Intuitive Rank Formulation

In the previous chapters, we studied processing algorithms to efficiently evaluate user ranking function $\mathcal{F}$. While such quantitative global ranking function $\mathcal{F}$ is amenable to processing algorithms, or “DB-friendly”, it is far from trivial for everyday user to formulate such function, representing how she would evaluate each every object into a numeric score. Rather, we believe an effective retrieval system should be more “user-friendly” so that they can formulate ranking in a more qualitative and intuitive way. In this chapter, we thus develop an intuitive rank formulation front-end and discuss how to integrate such front-end with processing algorithms. Section 5.1 motivates and describes the architecture of this integrated framework. Section 5.2 presents the component technologies. Section 5.3 demonstrates the efficiency and effectiveness of this framework using real-life queries and datasets.

5.1 Overview: Bridging Rank Formulation and Processing

This section motivates and introduces our approach—Our goal is to seamlessly integrate user-friendly rank formulation with DB-friendly rank processing. As explained above, such “mix” is critical for enabling soft queries for data retrieval.

So far, we have discussed rank processing techniques, given a global and quantitative ranking function $\mathcal{F}$ (i.e., score-based ranking view). This view is 1) amenable and 2) expressive to enable effective query processing: First, such a ranking function is amenable to all existing rank processing algorithms (e.g., as the Query $Q_1$ in Example 1). Second, it is simple yet expressive, by determining a global ordering with a single formula. (Such score-based models have served IR well, e.g., the $tf/idf$ scoring function for ranking.)
While this score-based view is expressive and efficient, formulating such ranking functions is challenging to users. It is far from trivial for the user to articulate how she evaluates each and every object into an absolute numeric score, that is, to express her preference by defining the soft predicates and function. Note that, unlike typical relational queries usually formulated by application developers or DB administrator, common users for data retrieval tasks are ordinary people. Thus, to accommodate such users, the formulation of rank criteria must be essentially supported—without which ranking is not usable.

To enable effective rank formulation, we believe the framework should be both intuitive and exploratory: First, preference often stems from relative ordering without explicit absolute scores. Thus, while scoring is an underlying “computational machinery” to capture a desirable preference, explicit scoring is non-intuitive and overly-demanding to most users. To be intuitive, the framework should allow users to specify only relative ordering or partial orders (but not absolute scores)—it is up to the system to infer the underlying ranking function from a few given examples. Second, ranking often requires context knowledge—of what objects are available in the database to be ranked. However, data retrieval is inherently exploratory; users are exploring an unfamiliar database for what they want, and thus such context knowledge is often lacking. Thus, the framework should present what are available in the database, and let users focus on only those presented. These examples on one hand serve as a “guided” tour of \( \mathcal{D} \) and on the other hand provide a sufficient context for user interaction.

Together, both requirements lead us to pursue an interactive “rank-by-examples” paradigm for rank formulation—Consequently, the critical ability of “inference by examples” (for finding the implicit ranking criteria) clearly suggests a machine learning approach. With interactive sampling and labeling of training examples, our “learning machine” will infer the desired ranking function. However, unlike a conventional learning problem of classifying objects into groups, our learning machine must learn a global ranking function \( \mathcal{F} \) that outputs a ranking score of each data object so that it is adoptable in the score-based ranking view for efficient processing. The learning machine must also learn from partial orders to provide the intuitive formulation. Additionally, the dynamic nature of online querying poses strict constraints on response time and user intervention—this ranking function must be learned instantly with minimal user intervention.
Putting together, we develop the RankFP framework, aiming at integrating a “front-end” for learning-based rank query formulation to a “back-end” for score-based rank query processing. As Figure 1.2 illustrates, first, with the “iterative learning” front-end (at the top), our RankFP framework supports users to formulate queries in an exploratory (as the system iteratively shows database sample objects) and intuitive (by specifying only partial ordering on those samples) process. Second, with the score-based rank processing back-end (at the bottom), our framework supports integrated query processing to return ranked answers efficiently. Section 5.2 will present the techniques for each component.

Note that, unlike typical document retrieval tasks, users in data retrieval tasks are often willing to perform many iterations to further refine the ranking functions; A document retrieval task usually ends as soon as the user finds a few satisfying documents. However, users in data retrieval tasks often want to retrieve every possible candidate before they make decisions. For instance, users searching for for-sale houses or digital camcorders do not easily finish their tasks by retrieving a few good samples. Instead, they usually retrieve every possible candidate that fits their preferences before they make decisions on purchase; Spending more time on refining the ranking function is eventually likely to save the total time for performing the data retrieval task.

1. Learning machine randomly selects $l$ number of samples $S_1$ from a candidate set $\mathcal{D}(|S_1| = l \ll |D|)$.

2. User expresses her preference by ordering the $l$ samples (providing $R^*_{S_1}$, the order $R^*$ on the samples $S_1$).

3. Learning machine
   (a) using SVM techniques (Section 5.2.2), learns a function $F_i$ from $R^*_{S_1}$, where $i$ denotes the number of iterations,
   (b) using SVM top sampling (Section 5.2.3), intelligently selects another $l$ number of samples $S_{i+1}$ such that $S_{i+1}$ maximizes the benefit of learning of $F_{i+1}$, and
   (c) temporarily save $R^*_{F_{i+1}} = \text{the order on } S_{i+1} \text{ generated by } F_i$. (This will be used for evaluating function $F_i$ at Step 5.)

4. User expresses her preference by ordering the $l$ samples (providing $R^*_{S_{i+1}}$).

5. Learning machine computes $\tau(R^*_{S_{i+1}}, R^*_{F_{i+1}})$, and if the $\tau$ is higher than a threshold $\theta$, print out top results according function $F_i$ and exit. Otherwise, repeat from 3.

**Figure 5.1** The front-end rank formulation: “learning-till-convergence”
**Rank Formulation:** The rank formulation module (Figure 1.2, top) iteratively interacts with the user to learn the desired ranking function. This process operates in *rounds*, as Figure 1.2 illustrates (in the top) and Figure 5.1 shows in details. In each round, the learning machine selects a sample $S$ of a small number of $l$ objects (for $l \ll |D|$; e.g., $l = 5$ in our study). The user orders these examples by her desired ranking $R^*$; thus she “labels” these examples as training data. The learning machine will thus construct a function $\mathcal{F}$ from the training examples so far; let $R^F_S$ be the induced ranking over the latest sample $S$. At convergence, i.e., when $R^F_S$ is sufficiently close to $R^*$ (i.e., when $\mathcal{F}$ is accurate on $S$), the learner will halt and output $\mathcal{F}$ as the learned ranking function. (In particular, we measure such convergence with the Kendall’s $\tau$ metric, the most widely used measure for similarity between two orderings such as $R^*$ and $R^F_S$ [30, 29, 35].) This “learning-till-convergence” mechanism seems rather simple and elegant to satisfy our goal of intuitive and exploratory query formulation. (Section 5.2.2 and 5.2.3 will present the techniques for the rank formulation.)

**Rank Processing:** The rank processing module (Figure 1.2, bottom) carries out the learned function $\mathcal{F}$ for online query processing over the entire database. We will later present how to process this learned function.

### 5.2 The RankFP Framework: Enabling Rank Formulation and Processing Online

In this section, we present the techniques for realizing the RankFP framework (Figure 1.2). First, Section 5.2.1 starts with developing how we “connect” score-based ranking view, which is effective for processing back-end, with classification view, effective for learning front-end. Second, Section 5.2.2 then investigates SVM as the learning machine (Step 3a in Figure 5.1). Finally, Section 5.2.3 develops techniques to enable rank formulation and processing to be “online”, e.g., selective sampling for effective online learning with minimal user intervention (Step 3b in Figure 5.1).
5.2.1 Duality of Ranking and Classification View

As argued in Section 5.1, the score-based ranking model, viewing ranking as induced by a ranking function $F$, is amenable and expressive for query processing. For now, let’s assume the rank function $F$ is linear, i.e., $F(f_1, \ldots, f_m)[u] \equiv w_1 \cdot f_1[u] + \cdots + w_m \cdot f_m[u]$. (We will generalize to nonlinear ranking in Section 5.2.2.) Let the weights $\vec{w} \equiv (w_1, \ldots, w_m)$ be the weight vector (which is what the learner will infer). Also, let the “features” $\vec{f}_i \equiv (f_1, \ldots, f_m)[u_i]$ be the feature vector of data object $u_i$. In our learning framework, these features are simply the attributes of each database tuple (e.g., price or size). We can thus write $F_w(\vec{f})[u_i] = \vec{w} \cdot \vec{f}_i$, which maps an object $u_i$ to its score $F[u_i]$ by weighting its various features. As our hypothesis, suppose there exists such a ranking function $F$ that is consistent with the desired ranking $R^*$. Our ranking problem is thus to induce an order of objects by comparing their scores, and our goal in rank formulation is to find such an $F_w$ (or the weight vector $\vec{w}$) such that:

$$ u_i \geq_{R^*} u_j \iff F_w(\vec{f})[u_i] \geq F_w(\vec{f})[u_j] $$  \hspace{1cm} (5.1)

$$ \iff \vec{w} \cdot \vec{f}_i \geq \vec{w} \cdot \vec{f}_j $$  \hspace{1cm} (5.2)

We denote $u_i \geq_{R} u_j$ or $(u_i, u_j) \in R$ when $u_i$ is ranked higher than $u_j$ according to an ordering $R$.

To automatically infer ranking function $F_w$ by applying a machine learning method as a formulation front-end, we rewrite Eq.(5.2) into the following Eq.(5.3).

$$ \vec{w}(\vec{f}_i - \vec{f}_j) \geq 0 $$  \hspace{1cm} (5.3)

Now, the learning problem becomes a binary classification problem on pairwise ordering: Let $\vec{d}_{ij} \equiv \vec{f}_i - \vec{f}_j$ be the feature-difference vector between $u_i$ and $u_j$. Then it is formulated as the following binary classification problem that determines if $(u_i, u_j) \in R^*$.

$$ (u_i, u_j) \in R^* \iff F_w(\vec{d}_{ij}) \geq 0 \iff \vec{w} \cdot \vec{d}_{ij} \geq 0 $$  \hspace{1cm} (5.4)

Thus, our rank formulation problem can be formulated as the following problem: Let $R^*$ be a ranking over database $D$. Given training data or partial orders as a set $\{(u_i, u_j, y_{ij})\}$, where
$u_i \in \mathcal{D}$, $u_j \in \mathcal{D}$, and $y_{ij} = +1$ if $(u_i, u_j) \in R^*$ or else $y_{ij} = -1$, our goal is to learn a function $\mathcal{F}_w$ for classifying every pair of objects $(u_i, u_j)$ from $\mathcal{D}$ with respect to $R^*$, as Eq.(5.4) defines. Thus, we formally develop the duality of the classification and ranking view through the following theorem.

**Theorem 7 (duality):** Let $R^* = (\overrightarrow{f_1}, \overrightarrow{f_2}, \ldots)$ be the global ordering determined by pairwise function $\mathcal{F}$ such that $\mathcal{F}(\hat{d}_{ij}) \geq 0$ for every $(\overrightarrow{f_i}, \overrightarrow{f_j})$ pair satisfying $i < j$. Then, $\mathcal{F}(\overrightarrow{f_i}) > \mathcal{F}(\overrightarrow{f_j})$ if and only if $i < j$.

**Proof:** From Eq.(5.3) and (5.4),

$$\forall d_{ij} \in R^*, \mathcal{F}_w(d_{ij}) > 0 \Leftrightarrow \mathcal{F}_w(\overrightarrow{f_i} - \overrightarrow{f_j}) > 0 \Leftrightarrow \overrightarrow{w} \cdot (\overrightarrow{f_i} - \overrightarrow{f_j}) > 0 \Leftrightarrow \overrightarrow{w} \cdot \overrightarrow{f_i} > \overrightarrow{w} \cdot \overrightarrow{f_j} \Leftrightarrow \mathcal{F}_w(\overrightarrow{f_i}) > \mathcal{F}_w(\overrightarrow{f_j}).$$

This duality tells that $\mathcal{F}$ - a classifier function learned from the pairwise difference vectors - can be used for the global ranking function generating a score per object, which allows us to seamlessly integrate with existing relational database.

### 5.2.2 Incorporating SVM Learning for Rank Formulation

Our formulation of duality (Theorem 7) enables us to adopt any linear binary classification method (e.g., Perceptron, Winnow, SVM, etc)

1. for learning ranking function $\mathcal{F}$. Among those, *Support Vector Machines* or SVMs [44, 6, 14] have been recently most actively developed in the machine learning community, as they have shown to demonstrate high generalization performance by the margin maximization property

2. That is, they learn an accurate classification function that generalizes well beyond training data. (Generalization performance denotes the performance of the learned function on “unseen” data.)

Applying SVM on Eq.(5.4) becomes essentially equivalent to the solution proposed in [24], which uses SVM for ordinal regression. In this section, we provide an intuitive interpretation of the solution by presenting Corollary 1 and 2 and Remark 1, which explains how SVM can also improve the generalization of ranking. That justifies, our framework, by adopting SVM as the learning machine, can learn $\mathcal{F}_w$ that is concordant with the given training data (i.e., partial orders from $R^*$) and also generalizes well to rank unseen data with respect to $R^*$.

---

1. Given a set of (positive and negative) training points, a linear binary classifier will find the weight vector $\overrightarrow{w}$ (and thus the ranking function $\mathcal{F}_w$), which defines a “hyperplane” separating the positive and negative examples.

2. SVMs compute the classification boundary of the highest margin that is the distance between the boundary and the closest data points (i.e., support vectors) in the feature space.
5.2.2.1 SVM Classification

Let us first overview SVM classification. Suppose there exists such a function $\mathcal{F}_w$ that Eq.(5.4) holds for some partial ordering $R' \in R^s$, then we can rescale $\overrightarrow{w}$ such that the following Eq.(5.5) holds for that partial orders.

$$\forall (u_i, u_j) \in R' : \overrightarrow{w} \cdot \overrightarrow{d}_{ij} \geq 1 \tag{5.5}$$

For instance, let us say that the smallest output from the function of Eq.(5.4) is 0.01 for all $\overrightarrow{d}_{ij} \in R'$, and say that the vector outputting 0.01 is $\overrightarrow{d}$. Then we can rescale $\overrightarrow{w}$ such that $\overrightarrow{w} \cdot \overrightarrow{d} = 1$ and $\overrightarrow{w} \cdot \overrightarrow{d} > 1$ for all other vectors $\overrightarrow{d}$. (We will not actually rescale it, but in order to understand the SVM's margin maximization property, it is crucial to understand that there exists $\overrightarrow{w}$ that Eq.(5.5) holds if there exists $\overrightarrow{w}$ that Eq.(5.4) holds for a set of vectors $\forall \overrightarrow{d}_{ij} \in R'$.) The particular feature-difference vector $\overrightarrow{d}_{ij}$ for which Eq.(5.5) is satisfied with the equality sign (e.g., $\overrightarrow{d}$ in the above example) are called support vectors. Thus, in SVM classification, support vectors are the data objects closest to the decision boundary ($\overrightarrow{w} \cdot \overrightarrow{d} = 0$), because the decision function $\mathcal{F}_w(\overrightarrow{d})$ for the support vectors $\overrightarrow{d}$ returns the smallest possible value (= 1).

The margin denotes the distance from the support vector $\overrightarrow{d}$ to the decision boundary ($\overrightarrow{w} \cdot \overrightarrow{d} = 0$) in the feature space which is formulated as the following Eq.(5.6) since $\mathcal{F}(\overrightarrow{d}) = 1$.

$$m = \frac{\mathcal{F}(\overrightarrow{d})}{||\overrightarrow{w}||} = \frac{1}{||\overrightarrow{w}||} \tag{5.6}$$

SVMs compute a function $\mathcal{F}_w$ of the highest margin $m$ by minimizing $||\overrightarrow{w}||$ in Eq.(5.6). Figure 5.2 illustrates an example of the margin-maximized decision boundary (i.e., $\overrightarrow{w} \cdot \overrightarrow{d} = 0$) of the SVM binary classification in a two-dimensional feature space. Each data pair $\overrightarrow{d}$ is represented by ‘o’ or ‘+’ according to its class (e.g., $\mathcal{F}(\ell^+) \geq 1$ and $\mathcal{F}(\ell^-) \leq 1$). The data pairs on the dotted lines are the support vectors. SVM computes the boundary that separates the two groups of data and also maximizes the margin, i.e., the distance between the boundary and the support vectors in the feature space.

5.2.2.2 SVM for Ranking

In our ranking problem, from Eq.(5.2), we see that a linear ranking function $\mathcal{F}_w$ projects data vectors onto a weight vector $\overrightarrow{w}$. For instance, Figure 5.3 illustrates linear projections of four
Suppose $\mathbf{f}$ is an SVM function in $P^n_m$ generated from paired orders according to $\mathbf{m}$. Suppose $\mathbf{f}_1$ and $\mathbf{f}_2$ are the support vectors of the function $\mathbf{f}$, represented by the data points that are closest to each other in ranking.

**Proof:** Let $\mathbf{f}(\mathbf{d}) = \mathbf{w} \cdot \mathbf{d} = 0$. Then, the support vector where $\mathbf{f}(\mathbf{d}) > 1$ or $\mathbf{f}(\mathbf{d}) < -1$, $m = 1 / ||\mathbf{w}||$.

**Corollary 1:** Suppose $\mathbf{f}$ is an SVM function in $P^2_m$ generated from paired orders according to $\mathbf{m}$, that is, ordered by the geometric distance of the two vectors produced on $\mathbf{f}$.

For instance, in Figure 2.3, the ranking difference of $(\mathbf{f}_1, \mathbf{f}_2)$ is $1/2$. The ranking difference of two vectors $(\mathbf{f}_1, \mathbf{f}_2)$ can be derived by the geometric distance of the two vectors produced on $\mathbf{f}$.

The ranking difference of two vectors $(\mathbf{f}_1, \mathbf{f}_2)$, where the same ordering $\mathbf{f}_1$ and $\mathbf{f}_2$ make the same ordering for the four vectors such that vectors $\mathbf{f}_1$ and $\mathbf{f}_2$ are ordered by the geometric distance of the two vectors produced on $\mathbf{f}$, are $1$, $1/2$, and $1/3$ in a two-dimensional feature space. Both $\mathbf{f}_1$ and $\mathbf{f}_2$ make the same ordering $\mathbf{f}_1$ and $\mathbf{f}_2$ in a two-dimensional feature space.
smallest possible value for all data pairs \( \forall (u_i, u_j) \in R' \). Thus, its ranking difference according to \( \mathcal{F}_{\bar{w}} \left( = \frac{w'(\bar{f} - \bar{f}')}{\|w\|} \right) \) is also the smallest among all data pairs \( \forall (u_i, u_j) \in R' \). 

**Corollary 2:** The ranking function \( \mathcal{F} \), generated by the SVM, maximizes the minimal difference of any data pairs in ranking.

**Proof:** SVM computes a function that maximizes the margin, and the margin is formulated as \( \frac{\langle \bar{w}, \bar{x} \rangle}{\|\bar{w}\|} = \frac{w'(\bar{f} - \bar{f}')}{\|w\|} \), which denotes, from the proof of Corollary 2, the minimal difference of any data pairs in ranking.

**Remark 1:** Our framework, adopting SVM as the learning machine, generates a ranking function of high generalization performance.

**Rationale.** Consider the two linear ranking functions \( \mathcal{F}_{\bar{w}_1} \) and \( \mathcal{F}_{\bar{w}_2} \) in Figure 5.3. Although the two functions make the same ordering \( R \) for the four vectors such that \( \bar{f}_1 >_R \bar{f}_2 >_R \bar{f}_3 >_R \bar{f}_4 \), as we intuitively think, \( \bar{w}_1 \) generalizes better than \( \bar{w}_2 \) because the minimal difference of two projected vectors in \( \bar{w}_1 \) (i.e., \( \delta_1 \)) is larger than that in \( \bar{w}_2 \) (i.e., \( \delta_2 \)). From Corollary 1 and 2, SVM computes the weight vector \( \bar{w} \) that maximizes the minimal difference in ranking. Thus, our framework, adopting the SVM as the learning machine, generates a ranking function of high generalization performance.

**5.2.2.3 Incorporating Nonlinear Ranking**

We have discussed so far the framework assuming ranking function \( \mathcal{F} \) is linear. Conceptually, linear ranking functions consider the sum of only individual attributes since the weight is assigned to only each individual attribute (e.g., \( \mathcal{F}_{\bar{w}}(d) = w_1d_1 + w_2d_2 \) when \( d \) is a two-dimensional vector such as price and size for a house data). Though our experiment results report the vast majority, i.e., over 90%, of user preferences are learned accurately with linear ranking functions, we also need to support nonlinear ranking function to support complex preferences. Complex preferences in this context means the preference considering beyond individual features (e.g., \( \mathcal{F}(\bar{d}) = w_1d_1 + w_2d_2 + w_3d_1d_2 \) thus not being able to be expressed by linear functions.

SVMs support the “kernel trick” for nonlinear classification [44], which can be exploited for formulating a nonlinear ranking function: Eq.(5.2) becomes
\( \tilde{w} \cdot \Phi(f_i) > \tilde{w} \cdot \Phi(f_j) \iff \sum c_k K(f_k, f_i) > \sum c_k K(f_k, f_j) \) (Refer to [24] for details.) However, as the function gets more complex and expressive (as a nonlinear function usually does), its generalization performance tends to decrease and thus requires more training data to be learned accurately. This fact is explained in the Bias-Variance Tradeoff in classification [44]. It is also nontrivial to optimize nonlinear kernel parameters in an online environment. Thus, we leave the seamless integration of nonlinear ranking into the framework as future work.

### 5.2.3 Satisfying Online Requirements

While duality (Theorem 7) enables to apply learning machines for rank formulation, the dynamic nature of our online querying framework poses very strict constraints on (1) user intervention and (2) response time. First, for online rank formulation, we will discuss how to learn effectively with minimal user intervention, Second, for online rank processing, we will discuss how to evaluate ranking efficiently.

#### 5.2.3.1 Online Formulation: Top Sampling

Toward the goal of online formulation, we develop the top sampling technique, which on one hand, provides an exploratory interface to users and on the other hand, further enhances the learning performance of SVM (Step 3(b) in Figure 5.1). Particularly, this technique minimizes the user intervention required to achieve the user-specified accuracy.

SVM selective sampling has been studied in the context of binary classification in order to minimize the amount of human labors of labeling training data to achieve a certain accuracy [40, 43, 10]. Our top sampling technique extends it for learning ranking function and also providing an exploratory interface. The key idea of the top sampling is that at each round the learning machine selects the samples that (1) are most ambiguous in ranking such that the ordering on the samples maximizes the “degree” of learning and (2) are also highly ranked so that users can focus on the samples of their interests.

The SVM selective sampling for binary classification is based on a similar idea: At each round, a learning machine selects the samples that are closest to the classification boundary such that they are the most difficult to classify, and the user labels the selected samples which accumulates as the training data. Since an SVM classification function is represented by support vectors that are closest to the boundary, this simple idea quickly identifies the
support vectors and thus significantly reduces the total number of training (labeled) data to achieve a high accuracy.

Similarly, our framework in Step 3(b) of Figure 5.1 needs to select most ambiguous samples according the ranking function learned so far. From Corollary 1, the support vectors are the most ambiguous samples for ranking. Thus, the user’s feedback on those samples will maximize the “degree” of learning by quickly inducing the support vectors. At the same time, our framework needs to provide an exploratory interface to users. Our top sampling technique satisfies both requirements.

Initially, our framework starts with randomly selected \( l \) number of samples (Step 1). After a user provides the ordering on the samples \( R^*_S \) (Step 2), the learning machine includes the feature-difference vectors \( \tilde{d} \in R^*_S \) in the training set, and build \( \mathcal{F}_i \) from the training set using SVM techniques (Step 3a).

In Step 3(b), the learning machine selects another \( l \) number of samples \( S_{i+1} \) that are most difficult to rank among them, which we is formulated as:

\[
\arg \min_{\tilde{S}_{i+1}} C(S_{i+1})
\]  

(5.7)

where \( C(S_{i+1}) \) (i.e., the cost function of \( S_{i+1} \)) is the sum of the ranking difference of every data pair in \( S_{i+1} \), i.e.,

\[
C(S_{i+1}) = \sum_{\forall (\tilde{u}_j, \tilde{u}_k) \in S_{i+1}} |\mathcal{F}_i(\tilde{u}_j - \tilde{u}_k)|
\]  

(5.8)

Since in our framework we use every data pair \((\tilde{u}_j, \tilde{u}_k) \in S_{i+1}\) as training data, the data subset \( S_{i+1} \) that minimizes the cost function \( C(S_{i+1}) \) of Eq.(5.8) are most difficult to rank among them. Thus, based on the same principle as the original selective sampling for binary classification (i.e., the data close to the decision boundary are likely to be the support vectors of the boundary.), this technique effectively reduces the total number of samples to achieve a certain accuracy.

However, a direct optimization of Eq.(5.8) requires \( |D|C_l \) times of SVM function evaluations at each round, as it needs to evaluate every possible \( l \) number of data in the datasets, which
would delay the response time intolerably long. The optimized samples also could hardly serve as an exploratory samples since it does not consider the user’s preference.

Thus, our top sampling selects the top $l$ samples ranked according to the function learned in the previous round, that is, select $L = \{u_1, ..., u_l\}$ such that $\mathcal{F}(u_i) \geq \mathcal{F}(u_j)$ for $u_i \in L$ and $u_j \notin L$. As the data in the set $L$ is contiguous in ranking, the cost function of Eq.(5.8) for $L$ is likely smaller than that for an randomly chosen $l$ samples, and thus it approximates the SVM selective sampling technique for ranking. Our experiment in Section 5.3 shows that the top sampling achieves a high accuracy more quickly than random sampling.

The top sampling is efficient as well as exploratory: (1) Selecting such set $L$ requires only a single scan of the dataset, which evaluates the function only $|D|$ times while the direct optimization evaluates $|D|C_l$ times. (2) It selects highly ranked samples according to the function learned from the user’s feedback. Thus it naturally provides users with possible candidates which are unseen but highly likely preferred.

5.2.3.2 Online Processing

As a next step, we carry out “classification” by the learned function $\mathcal{F}$ over the entire database. Toward the goal of online processing, as data retrieval scenarios naturally situate in large databases, such processing should be efficient and scalable. However, such development has been clearly lacking, as machine learning methods did not have to address this “processing” aspect, as their objective is usually optimizing the accuracy.

As a major contribution towards online processing, recall that, our development of duality (Theorem 7) enables us to transform a pairwise classification function a global ranking function, and thus enables us to avoid evaluating the pairwise classification function for all object pairs. That is, once the pairwise classification function is learned we can conveniently use it as per-object ranking function $\mathcal{F}$ as well, according to the duality property, to adopt existing rank processing algorithms developed for a global per-object ranking function.

Similarly, the top sampling can be supported by rank processing algorithms, which enables a uniform support for both query processing and sample selection. To illustrate, at $i$-th iteration, our top sampling scheme is essentially $k$ objects ordered by $\mathcal{F}_i$.

However, note existing algorithms cannot be adopted “as is”: While the learning rank formulation front-end can learn any arbitrary ranking function, all existing rank processing
supports only the monotonic functions. Toward this goal of supporting any arbitrary ranking functions learned, we develop a rank processing back-end that no longer relies on the monotonicity assumption in Chapter 6.

5.3 Experimental Evaluation

This section reports our extensive experiments for studying the usability (or “user-friendliness”) and efficiency (or “DB-friendliness”) of our RankFP framework: First, for usability, we used Kendall’s τ measure [30, 29, 35], which is used widely to measure the similarity of the two orderings, i.e., the ideal ordering $R^*$ and the ordering generated by our system $R^F$. Second, for efficiency, we measured absolute response time. Our experiments were conducted with a Pentium 4 2GHz PC with 1GB RAM.

**Implementation.** We use $\nu$-SVM$^3$ (included in LIBSVM$^4$). As for the size of samples $l$, we set it as 5, which gave fairly good results among all values. Note deciding $l$ is a trade-off problem: Small $l$ requires more iterations, while large $l$ makes ordering non-trivial.

**Data Set.** We perform our experiments on a real-estate system (as Example 1 introduced) with real-life house dataset used in [12].$^5$ This data set is extracted from realtor.com and contains all the for-sale houses in Illinois, resulting in $N = 20990$ objects for relation house, each with attributes id, price, size, beds, baths, zip and city. In addition, we translated the zip code into the coordinates of latitude and longitude to support the notion of closeness between two locations.

First, to evaluate the framework extensively, we synthetically generate queries with various complexity and measure the performance in Section 5.3.1. We then evaluate the framework with real-life queries, collected from our user study, in Section 5.3.2.

---

$^3$ $\nu$-SVM employs a semantically meaningful soft margin parameter $\nu$ [41, 9]. An intuitive setting of $\nu$ normally works well. We fixed $\nu = 0.1$ for our experiment.

$^4$ http://www.csie.ntu.edu.tw/~cjlin/libsvm

$^5$ http://aim.cs.uiuc.edu/readme.html
5.3.1 Experiment on Synthetic Queries

First, we evaluate the framework using synthetic queries, expressed in scoring-based model, as shown in Figure 5.4: Query 1 shows a scenario of finding a house in a big city Chicago, while Query 2 shows a scenario of finding a house in a small city Urbana.

**Query 1:**

```
SELECT id FROM house WHERE city="Chicago"
ORDER BY average(a,b,c,d):
  a = cheap(price), b = large(size),
  c = many1(beds), d = many2(baths)
```

**Query 2:**

```
SELECT id FROM house WHERE city="Urbana"
ORDER BY average(a,b,c,d):
  a = cheap(price), b = large(size),
  c = many1(beds), d = many2(baths)
```

---

**Figure 5.4 Preference queries**

Queries in Figure 5.4 show that the user is interested in cheap and large houses having many beds and baths. Further, predicate definitions in Figure 5.5 illustrates the user’s preference in more details: For instance, according to definitions in *Predicates 1*, user is willing to trade 200 square footage in size (which decreases score of $b$ by 20) for having one more bedroom (which increases score of $c$ by the same amount). *Predicate 1* can be formulated as the following linear ranking function:

$$
F = 100 - 0.001 \times \text{price} + 0.1 \times \text{size} + 20 \times \text{beds} + 20 \times \text{baths}
$$

(5.9)

To contrast, definitions in *Predicates 2* (Figure 5.5) show more complex preferences: For instance, from the function $\text{many1(beds)}$ $c$, we can observe that the user penalize much for the houses with less than three bedrooms, while giving the same score for all houses with more than five bedrooms. As a result, queries using Predicates 2 will not be “linearly rankable”.

To evaluate the framework with queries with various degrees of complexity, we mix and match two queries and two sets of predicate definition, i.e., Q1+P1, Q2+P1, Q1+P2, and Q2+P2. For each combination, we generate user feedbacks, i.e., partial orderings, based on the given query combination. We then measured the accuracy of the ranking function learned and
response time at each round, with both random sampling and the top sampling. Table 5.1 summarizes the results. We highlight our observations as follow:

- For the linearly rankable queries (i.e., Q1+P1 and Q2+P1 in Table 5.1), the top sampling technique generates noticeably higher performance from the second round. Figure 5.6 illustrates the performance difference between the random sampling and the top sampling at each round. The performance of the first round does not make much difference because the samples in the first round can only be randomly selected in both cases. However, as rounds go on, the top sampling technique quickly achieves higher performance with fewer number of samples. References [40, 43] observe similar behaviors in binary classification problems.

- Observe that, the response time is similar for Q1 and Q2 when random sampling is used. In contrast, top sampling is proportional to database size. For instance, as there are more houses in Chicago than in Urbana, top sampling is less efficient in Q1 than in Q2. Thus, supporting an efficient top sampling using indexing would be an interesting future work.

- For the non-linearly rankable queries (i.e., Q1+P2 and Q2+P2), Gaussian kernel performs better than linear kernel. However, top sampling with Gaussian kernel does not perform noticeably better than random sampling.

5.3.2 Experimental on Real Queries

In this section, we evaluate the framework in more realistic settings. Ten ordinary users tested our system with their own house preferences from which 100 real queries were collected. Note that, in this user-study setting, the perfect ordering \( R^* \) the user intended remains unclear. (It is infeasible for each user to provide a complete ordering \( R^* \) on hundreds or thousands of houses.) Thus, the accuracy of the ranking function at this iteration is evaluated against the partial ordering specified by user in the next iteration. That is, the accuracy of ranking function \( F_i \) learned at \( i^{th} \) iteration is measured by comparing the similarity of user’s partial ordering on \( S_{i+1} \) at the next iteration and the ordering generated by \( F_i \), i.e., \( R_{S_{i+1}}^{F_i} \).
Table 5.1 Performance results (averaged over 20 runs). RAN: random sampling; TOP: top sampling; Q: query; R: # of rounds (l = 5); Acc: accuracy (%); Time: average response time (Sec.); Kernel: the kernel of higher accuracy

This measure approximates the generalization performance of ranking functions well as $S_{i+1}$ is not a part of training data for learning $\mathcal{F}_i$. Further, using this evaluation method, we can also acquire fair evaluations from users since the users are not aware of whether they are providing feedback or evaluating the functions at each round.

However, this measure severely disfavors top sampling: Intuitively, top sampling will be most effective for learning if user’s ordering on $S_{i+1}$ is not what is expected from the previous section, $R^{\mathcal{F}}_{S_{i+1}}$. We thus use random sampling for the user study reported in this section.

However, note that, top sampling is expected to be effective in practice, as most real queries are linearly rankable (as Figure 5.7 will show) and top sampling is effective with linear ranking function (as discussed in Section 5.3.1).

Overall Result

Figure 5.7 shows the distribution of user preferences generating over 90% and 100% accuracy respectively per each iteration. (Note that the accuracy here is an approximation which is
computed over five random samples.) Observe from Figure 5.7 that linear kernel reaches 90% accuracy mostly within the second iteration, and 100% accuracy within the third iteration. We deduce the following observations from the analysis of the experiments.

- For the real preferences, our rank query framework formulates an accurate (i.e., accuracy ≥ 90%) ranking function in a couple of communications with a user (i.e., iterations ≤ 2) quickly (i.e., response time ≤ 10 millisec. with random sampling) and automatically (without any parameter turning in the processing). For instance, in the rank query on the houses in Chicago where |D| ≈ 1500, users are provided over 90% accurate ranked list by ordering just two 5 samples among the 1500 houses. In other words, our framework provided over 90% accurate pairwise orders on 2248500 (= 1500 C2) house pairs from just 20 (= 2 × 5 C2) pairwise orders. From our experiments on the synthetic queries, top sampling can even further improve the accuracy.

- The accuracy is scalable to the database size |D|. For instance, Table 5.1 shows that 20 pairwise orders surprisingly generated higher accuracy in a larger |D| (i.e., houses in Chicago) than in those in Urbana. It can be explained by the fact that attributes of the Urbana houses are less variant, e.g., sharing mostly the same zip code, which provides less clues to learning machine.

Illustration of a Running System

To qualitatively demonstrate our framework, Figure 5.8 shows a screen shot of our framework that we used in user study. At Round 1, the system showed randomly selected five samples of houses. Each house shows its price, # of beds and baths, size, and location (i.e., latitude and longitude extracted from the zip code). We have not performed any normalizations of the attribute values. Assume that the user prefers cheap and large houses of many beds and baths. Based on this criteria, a user ordered the five houses as 5 > R 2 > R 4 > R 3 > R 1 as shown in Figure 5.8. From this partial orders, our system generated the SVM model which can be represented as the following ranking function:

\[ \mathcal{F}_1 = -0.00040567886125586483 \times \text{price} \]
$$+0.0004819932549453127 \times \text{beds}$$

$$+0 \times \text{baths}$$

$$+0.041829246236619627 \times \text{size}$$

$$-0.00025355851829348932 \times \text{latitude}$$

$$+7.4244894504004128e-05 \times \text{longitude}$$

$$-1.17129e-11$$

Observe that, the ranking function $F_1$ from the first iteration captures that the user prefers houses of low price, many beds, and large size. In contrast, the weights of baths and latitude are very low, which implies, from the first round, these attributes do not contribute much in distinguishing desirable objects. (Note, attributes are not normalized, e.g., the values of price and size are large compared to the others.)

At Round 2, the system showed another randomly selected five samples, and the user ordered them as $1 > R^*, 2 > R^*, 3 > R^*, 4 > R^*, 5$, which turned out to be the same as the orders generated by $F_1$ of linear kernel. (The expected accuracy of linear kernel after Round 1 showed 100% in Figure 5.8.) Thus, our system printed out the top ten results according to the ranking functions of linear kernel and RBF kernel respectively as shown in Result of Figure 5.8. The top house ranked by the linear function (i.e., PRICE:59000, BEDS:8, BATHS:2, SIZE: 1080, ...) is large and of many beds while low in price.
Predicates 1:

cheap(price) a:
\[ a = (100,000 - \text{price}) \times 0.001 \]

large(size) b:
\[ b = \text{size} \times 0.1 \]

many1(beds) c:
\[ c = \text{beds} \times 20 \]

many2(baths) d:
\[ d = \text{baths} \times 20 \]

Predicates 2:

cheap(price) a:
\[
\begin{align*}
\text{IF} \ \text{price} > 100,000 \ \text{THEN} \\
\quad a &= (\text{price} - 100,000) \times 0.002 \\
\text{ELSE} \\
\quad a &= (100,000 - \text{price}) \times 0.001
\end{align*}
\]

large(size) b:
\[ b = \text{size} \times 0.1 \]

many1(beds) c:
\[
\begin{align*}
\text{IF} \ \text{beds} < 3 \ \text{THEN} \\
\quad c &= \text{beds} \times (-50) \\
\text{ELSE IF} \ \text{beds} < 5 \ \text{THEN} \\
\quad c &= \text{beds} \times 10 \\
\text{ELSE} \\
\quad c &= 50
\end{align*}
\]

many2(baths) d:
\[
\begin{align*}
\text{IF} \ \text{baths} < 2 \ \text{THEN} \\
\quad d &= \text{baths} \times (-50) \\
\text{ELSE IF} \ \text{baths} < 3 \ \text{THEN} \\
\quad d &= \text{baths} \times 10 \\
\text{ELSE} \\
\quad d &= 30
\end{align*}
\]

Figure 5.5 Definitions of fuzzy predicates

![Graph 1](image1)

![Graph 2](image2)

Figure 5.6 Performance convergence of two sampling techniques on the linearly rankable queries. TOP: top sampling; RAN: random sampling; X-axis: # of rounds; Y-axis: accuracy
CHAPTER 6

Supporting Non-monotonic Ranking Function

While we observe from Chapter 5 that user ranking functions can be quite arbitrary, e.g., Figure ??, existing processing algorithms focus on supporting ranking functions that are monotonic with respect to sorted accesses: That is, existing algorithms derive “stop condition” for efficient processing, relying on the assumption that the orderings materialized (i.e., sorted accesses) all correlate positively toward the overall ranking. Considering data retrieval is inherently ad-hoc, in that sensible orderings depend on the specific user, it is often infeasible to expect such orderings to be pre-materialized, in which case existing algorithms incur exhaustive accesses.

To support these non-monotonic functions efficiently, Section 6.1 discusses how we support ranking queries without relying on the monotonicity assumption, by developing a fresh perspective of abstracting rank processing as a score optimization problem of finding “highest-scored” objects in the “data space”. Sections 6.2 and 6.3 demonstrate how this framework can be implemented for middleware sources. Section 6.4 develops a “collective I/O” techniques to further optimize the access costs. We evaluate our framework in Section 6.5.

6.1 Framework AsOpt: Top-k As Score Optimization

In this section, we propose our rank processing back-end framework AsOpt, abstracting query answering as an optimization problem. First, we motivate the framework discussing why such an abstraction is essential in our integrated data retrieval system (Section 6.1.1). Second, we discuss how we realize such an abstraction as a systematic framework (Section 6.1.2). Lastly, we demonstrate the generality of our proposed framework, showing how it can unify (and generalize beyond) the existing top-k algorithms (Section 6.1.3).
6.1.1 Motivation

As Chapter 2 overviewed, all existing rank processing algorithms rely on the assumption that the ranking function $F$ is monotonic with respect to sorted access. Such a restriction turns out to be too restrictive, in our integrated data retrieval system, where any arbitrary ranking function can be learned from the formulation front-end. To illustrate, Figure 6.1 visualizes an example rank function learned from the front-end: This user (say, Amy) looks for a house that price and size are just right for her, e.g., price around $300k and size around 2000 sqft. Meanwhile, she is likely to negotiate her preference on price if very spacious houses (e.g., large enough for future family growth) are available. Based on this preference, Figure 6.1 plots the scores ($z$-axis) of houses characterized by price ($x$-axis) and size ($y$-axis). Observe the houses score high when price is around $300k and size around 2000 sqft, or alternatively, when they are very spacious, which explains another “peak”.

Observe this function is non-monotonic with respect to any of attribute indices. As a result, unless there exist user-specific sorted accesses materializing her preference on price and size, which is unlikely, rank processing algorithms developed in previous chapters can be very inefficient—As they take advantage of sorted accesses only when they correlate positively to the overall function, these algorithms will not be able to take advantage of any access structure and will end up performing “exhaustive” accesses (incurring prohibitive access cost).

In this section, we thus extend rank processing to perform “focused” accesses, taking advantage of access structures, even when the given ranking function is non-monotonic. We address this need by rethinking rank processing with a fresh view— that a top-$k$ query is essentially a “score optimization” problem optimizing ranking function $F$ over database $\mathcal{D} = \{u_1, \ldots, u_n\}$. That is, our task is to identify the $k$ objects $\mathcal{K} = \{u_1, \ldots, u_k\}, \forall u_i \in \mathcal{D}$ such that $F(u_i)$ scores are the maximum.

Our abstraction thus views processing top-$k$ queries as searching the score space (Figure 6.1) and retrieving the $k$ high-scoring objects (e.g., those near the high-scoring peaks). To the best of knowledge, our work is the first abstracting query answering itself as an optimization. (To clarify potential confusion, not query optimization.) This abstraction is useful as it enables to use existing optimization schemes, e.g., hill climbing, for query answering. However, more
Figure 6.1 Example score space.

importantly, as such schemes generally optimize for any arbitrary function, this abstraction naturally enables to support any arbitrary learned ranking functions.

6.1.2 Framework

While appealing in idea, realizing this optimization abstraction for query answering can be challenging: First, recall that, query answering optimizes $\mathcal{F}$ over a discrete set of data objects $\mathcal{D} = \{u_1, \ldots, u_n\}$, while function optimization problems typically assume a well-defined and continuous data space. Second, accessing objects in $\mathcal{D}$ and obtaining their attribute values incur non-trivial cost in return, e.g., I/O (local source) or communication cost (external source).

These critical differences prohibit us from applying existing optimization schemes “as is”. To illustrate, consider applying hill climbing for query answering—Starting from a random starting point, hill climbing iteratively chooses the next object to evaluate, until the utility (i.e., $\mathcal{F}$ score) of the next object no longer improves, i.e., the search has reached ached a local optimum. To retrieve top-$k$ results, we can repeat this whole process $k$ times, keeping requesting the next top-1 for $k$ times. In typical function optimization, both identifying the next object and evaluating its utility are inexpensive in-memory operations, e.g., randomly generating an object in the continuous data space and computing $\mathcal{F}$. However, in query answering, making sure the next object belongs to $\mathcal{D}$ and obtaining its attribute values from $\mathcal{D}$ require accesses to $\mathcal{D}$, e.g., disk I/Os. Obviously, this naive scheme incurs prohibitive access
cost—In our toy evaluation of 1000 uniformly random objects, evaluating $k = 100$ requires to access the majority of $\mathcal{D}$ (60% in average).

We thus naturally ask: Can we take advantage of function optimization schemes to survey the score space, without this prohibitive access cost? Such a survey will be essential in generating “focused accesses” selectively retrieving only the high-scoring objects. To address this dilemma, we propose a “two-phase” approach, decoupling global function optimization and local object search: First, the function optimization phase optimizes the ranking function, over “virtual” continuous space. As such a virtual space, we use the domain of $\mathcal{D}$, i.e., a cross product of the set of values in $\mathcal{D}$ for each attribute. To illustrate, continuing Amy’s example (Figure 6.1), we survey the shape $\mathcal{F}$ over the virtual space of domain($\mathcal{D}$) and identify a likely optimal house for her, e.g., ($300k, 2000\text{sq ft}$). However, note this is only a “virtual” optimum, since we cannot verify whether it exists in $\mathcal{D}$, until we actually access $\mathcal{D}$. For this purpose, the second phase of object search follows—Being informed of the virtual optima identified from the first phase, we can now perform a focused exploration of $\mathcal{D}$, to efficiently retrieve the top-$k$ results.

Observe that these two phases in fact complement each other—The first phase surveys the landscape of the function to inform the second phase, while the second phase completes the query answering with retrieving the actual optima that exist in $\mathcal{D}$. Figure 6.2 illustrates how our two-phase framework works in a toy example—To find top-3 results among 10 objects (Figure 6.1), the first phase will first identify two virtual optima (marked by ‘+’), followed by the second phase performing a focused retrieval (e.g., dotted rectangles in Figure 6.2) of top-3 actual objects around the virtual optima. As many sources efficiently support such a focused
Framework AsOpt($\mathcal{F}, k, D, H$): Top-$k$ answering as optimization

Input:
- $\mathcal{F}$: scoring function
- $k$: retrieval size, i.e., to return top-$k$ answers.
- $D$: input database.
- $H$: statistics (if any) on $D$ (e.g., histograms).

Output: $K$, the top-$k$ answers with respect to $\mathcal{F}$.

Procedure:
(1) Function Optimization Stage: surveys the shape of $\mathcal{F}$ and identifies likely optima.
   // implements genetic algorithm, while other function optimization schemes can generally apply.
   initialize $\mathcal{G}$, $K$ $\leftarrow \{}$; $Q$ $\leftarrow \{}$
   // initialize generation $\mathcal{G}$ with random objects. initialize $K$ for outputs and $Q$ for virtual optima.
   while (true):
     $\mathcal{G}$.crossover()
     $\mathcal{G}$.mutation()
     $\mathcal{G}$ $\leftarrow$ $\mathcal{G}$.select() // select high-scoring objects to the next generation.
     if converged($\mathcal{G}, \mathcal{G}$): break // stop if generations converge.
     $Q$.insert($\mathcal{G}, \mathcal{F}$) // insert the final generation to the queue $Q$ prioritized by their $\mathcal{F}$ scores.

(2) Object Search Stage: retrieves actual top-$k$ objects.
   // implements hill climbing, while other schemes can generally apply.
   while (true):
     $u$ $\leftarrow$ $Q$.top() // the highest-scoring point.
     $K$.insert($\text{evaluate}(u, H), \mathcal{F}$) // retrieve and evaluate the high-scoring data object(s); insert them to $K$.
     // note this performing evaluate every iteration is only conceptual (ref: Section 6).
     neighbors $\leftarrow$ $\text{explore}(u, H)$ // explore neighbors around $u$.
     $Q$.insert(neighbors, $h$) // insert promising neighbor(s) to $Q$.
     if $K[k-1] \geq \mathcal{F}(Q, \text{top})$: return $K$ // stop if top-$k$ in $K$ score higher than unevaluated objects.

Top-$k$ Output: return the top-$k$ objects in $K$ in order.

Figure 6.3 Framework AsOpt.

retrieval, e.g., selection queries by attribute values, this framework can avoid exhaustive
evaluation for non-monotonic functions.

Figure 6.3 now develops this idea into our two-phase framework AsOpt: The first phase
surveys the shape of $\mathcal{F}$ over domain($D$)– While any function optimization schemes generally
apply, we illustrate this phase using our implementation of genetic algorithm scheme. Genetic
algorithm imitates biological processes in finding virtual optima: Starting from an initial pool
of random objects, at each iteration, known as generation, high-score objects survive to the
next generation with high probability (selection). Further, search adds new objects to
generation by applying crossover and mutation operations: Crossover generates a promising
offspring by exchanging parts of high scoring objects. For instance, Figure 6.4 illustrates how
exchanging attribute values of objects $u$ and $v$ generates a larger and cheaper offspring $w$.
The high-scoring objects will survive over generations until generations eventually converge
(e.g., until the difference of scores of two generations is within user-specified parameter $\Delta$.)

111
Once the first phase surveys the score space and identifies the virtual optima, e.g., \( v_1 \) and \( v_2 \) in Figure 6.2, the second phase follows to generate focused accesses, in order to retrieve actual top-k results that exist in \( D \) (e.g., dotted rectangles in Figure 6.2). For this purpose, we illustrate this second phase using our implementation of hill climbing scheme. At each iteration, starting from the highest-scoring object \( u \) (as prioritized in the queue \( Q \)), we **evaluate** the actual objects around \( u \) using source-specific accesses (e.g., objects in the dotted rectangles in Figure 6.2). This operation can take advantage of source-specific information, e.g., histogram \( H \), if exists. We then **explore** around \( u \) to identify neighboring objects to those just accessed and store the promising neighbors (determined by some heuristics function \( h \)) in the priority queue \( Q \), as potential next moves. It terminates when no potential next moves in \( Q \) are expected to beat the current top-k results. Note, the implementation of a focused retrieval, more specifically, **evaluate** and **explore** operations (shown in **bold** in Figure 6.3) are source-specific. For instance, in Web sources, query forms support selection queries by attribute values, which can efficiently focus the retrieval only to the promising region (e.g., dotted rectangles in Figure 6.2).

### 6.1.3 Unification and Generalization

This section now demonstrates how framework AsOpt (Figure 6.3), by supporting arbitrary functions, unifies (and generalizes beyond) the existing ranking algorithms. That is, while not intended, existing algorithms can cast as an instance of framework AsOpt, which enables to understand existing works in a unified way and identify the challenges in generalizing beyond...
them. In particular, this section illustrates with a representative top-\( k \) algorithm each for middlewares [19] and database scenarios [4].

We begin with how TA [19] can cast as an instance of AsOpt: As overviewed in Section 1, TA builds on the monotonicity of \( \mathcal{F} \) over sorted accesses. We thus illustrate TA with a monotonic query example: Suppose we have sorted accesses on raw attributes size and bed, and our ranking function is monotonic with respect to these accesses, for instance, \( \mathcal{F} = \text{size} + \text{bed} \).

We now illustrate how Algorithm TA casts as an instance of a two-phase optimization: Note, the monotonicity assumption far simplifies both the first and second phase. First, with this assumption, the first phase of function optimization is rather trivial—It is implicit from the assumption that the virtual optimum lies where the values of the attributes size and bed are the largest (i.e., an upper right corner, as marked by ‘+’ in Figure 6.5). Second, starting from this virtual optimum, TA implements explore operation (Figure 6.3) of identifying potential next moves using sorted accesses: Figure 6.5 illustrates TA performing sorted accesses on size and bed to identify object \( u \) and \( v \) (near the virtual optimum) as the next moves. TA then evaluate these objects using random accesses retrieving their attribute values to compute \( \mathcal{F} \).

These iterations of explore and evaluate continue until top-\( k \) objects evaluated thus far score no less than the upper bound score of unevaluated objects. Note, this “stop condition” relies on the monotonicity assumption: As Figure 6.3 illustrates, at any given point, the score of virtual object \( o \) at the intersection of sorted accesses cannot score less than unevaluated objects in the shaded area that are smaller than \( o \) in both size and bed values—Thus, the current top-\( k \) results scoring higher than \( o \) are guaranteed to be accurate. Observe, this
reliance of TA to the monotonicity assumption, both in function optimization and object search, makes it impossible for TA to support non-monotonic ranking functions.

Similarly, we now discuss how reference [4] can cast as a special case of our framework. In contrast to TA bases object search on sorted and random accesses (specific to middleware sources), reference [4] bases on Boolean range queries that are efficiently supported by attribute indices. However, their scenarios are restricted to finding top-\( k \) “nearest neighbors” from the user-specified query point \( q \) and distance function \( \mathcal{F} \). Consequently, the first phase is again trivial, with the virtual optimum \( q \) explicitly specified. The second phase is simplified as well, as they assume \( \mathcal{F} \) is symmetric to the given optimum \( q \)– Object search phase thus boils down to find a search distance \( d \) from \( q \), such that all top-\( k \) tuples lie within the distance \( d \) from \( q \), i.e., within a symmetric circle centered at \( q \) with radius \( d \). Due to this assumption, this scheme cannot apply to arbitrary ranking functions, potentially asymmetric with multiple optima (as in Figure 6.2).

In contrast, our framework generalizes beyond these works and supports any arbitrary ranking functions. Our contribution is two-fold, tackling the challenges no existing works have yet addressed:

- **Explicit function optimization**: We propose an explicit function optimization phase that removes reliance on the known virtual optimum (given implicitly [19] or explicitly [4]).

- **General object search**: We propose a general object search phase that removes reliance on restrictive assumptions (e.g., monotonicity [19] or symmetry [4]).

### 6.2 Implementation: Using Boolean Selection Queries

In this section, we discuss how framework AsOpt can be implemented, or more specifically how *evaluate* and *explore* operations can be implemented, leveraging traditional Boolean selection queries, as efficiently supported by many sources using “user-generic” access structures (e.g., attribute indices).

Note, such operations naturally benefit if there exists any source-specific information or statistics, e.g., distribution of data objects. In particular, we study how we can leverage the most common form of such statistics– a histogram. Formally, a histogram \( H \) on relation \( R \) is
essentially a set of buckets \( H = \{ b_1 : (r_1, f_1), \ldots, b_m : (r_m, f_m) \} \), where each bucket \( b_i \) is
defined by a hyper-rectangle \( r_i \) within the domain of \( R \), and each frequency \( f_i \) is the number
of tuples in \( R \) that lie within \( r_i \). The buckets are disjoint and their union covers the entire
domain of \( R \). Note, histogram is not essential to our framework. When no a priori
information exists, we can use a “dummy” histogram built with an arbitrary assumption (e.g.,
uniform distribution): While the accuracy of histogram helps to improve efficiency, it does not
affect accuracy (Section 6.5).

We now build evaluate and explore operations using this histogram \( H \): First, to develop
evaluate generating a range query, given the top object \( u \) from \( Q \), we use histogram \( H \) to
find a bucket \( b_i : (r_i, f_i) \) containing \( u \) and query the respective range \( r_i \). Second, to develop
explore identifying the next moves, we use \( H \) to identify the unvisited neighboring buckets—
Among them, hill climbing will choose the “most promising” one, according to some heuristic
function \( h \), e.g., \( F \) score of the bucket centroid. The most promising move identified will be
stored in \( Q \) so that it can be evaluated in later iterations.

Figure 6.6 illustrates how this implementation actually works, using a toy example— Initially
(Figure 6.6a), queue \( Q \) stores virtual optima \( v_1 \) and \( v_2 \) (as marked by ‘+’) with their scores,
as identified from the first phase. First, we evaluate the highest scoring optimum \( v_1 \), by
generating a range query (shaded region in the Figure 6.6b)— Such a query, generated by
identifying the bucket containing \( v_1 \), retrieves the actual data objects residing in the range,
e.g., object \( a \) (as marked by ‘o’). The evaluated data objects will then be inserted into \( K \).
Next, we explore the neighboring buckets in Figure 6.6(b) as potential next moves and identify the one with the highest centroid score (Figure 6.6c). This centroid will then be inserted into $Q$, so that this range can be revisited in a later iteration. Framework A3Opt keeps repeating these iterations, until top-$k$ objects are identified. We keep picking the next most desirable object from $Q$ (e.g., $b_i$) and evaluate it (Figure 6.6d), until top $k = 2$ objects (e.g., objects $a$ and $b$) score higher than the expected scores of unevaluated buckets remaining in $Q$.

Note this scheme is approximate, as the seemingly desirable next move (determined by heuristics $h$) may prematurely converge to a low-scoring local optimum. However, in our empirical evaluation (Section 6.5), this hill climbing object search has achieved extremely high accuracy, e.g., the average score deviation of our results from the exact ones were less than 0.1%. Such an approximation, trading accuracy for efficiency, is especially effective in ranked query with fuzzy semantics.

6.3 Implementation: Dealing with Middleware Source Restrictions

This section now discusses how framework A3Opt can apply to middleware sources. While they often support attribute range queries, some techniques developed in Section 6.2 may not apply, due to the following common restrictions that: (1) sources may be remotely located and (2) the given query interface imposes additional query constraints. We thus study these restrictions and discuss how they impact on developing object search for middleware sources. The first constraint in supporting middleware scenarios comes from the fact that data objects may lie in a remote source. In such case, accessing data object thus incurs non-trivial overhead (e.g., communication cost). As a result, building histograms up-front is prohibitively expensive, if not impossible.

As a low-cost alternative, we can avoid this prohibitive initial build-up cost, applying the “self-tuning” histogram idea [1]: We start with an initial histogram with any information or assumption on attributes (e.g., assuming uniform distribution). We then progressively refine the histogram as queries are issued. In particular, we can use the actual selectivity of range selections performed, as a free feedback to improve the histogram.
Figure 6.7 Example query interface.

More specifically, as each selection query is performed, we identify the buckets that overlap with the query, based on which we estimate the result size. We then compute the estimation error, i.e., $\text{actual result size} - \text{estimated size}$, and refine accordingly– When the error is positive (i.e., overestimation), the bucket frequencies need to be lowered, proportionally to the contribution each bucket makes to the result size estimation: The contribution of bucket $b_i$ can be computed as the product of its frequency and the overlapped fraction with the given query, i.e., $f_i \cdot \frac{\text{size}(b_i)}{\text{size}(Q)}$. Similarly, when the error is negative, the bucket frequencies are raised proportionally. In addition to refine the bucket frequencies, we can refine the bucket boundaries as well– To get the ideal partitioning, buckets with similar frequencies are merged and buckets with high frequencies are splitted.

The second constraint in supporting middleware scenarios stems from the query interface. To illustrate, consider the interface of a Web middleware source (www.realtor.com) as shown in Figure 6.7: This interface enables our framework to formulate range queries, as we did in Section 6.2, but with the following interface-specific restrictions:

1. The city and state (or the zip code) of the location “must” be specified;

2. The query predicate is pre-formulated. For instance, selection condition on bed can only be specified in the format of $\text{bed} \geq b$ and city in the format of $\text{city} = c$. Further, query values can be restricted as well, e.g., query values for bed can only be chosen from those provided in the drag and drop menu, i.e., $b \in \{1, 2, 3, 4, 5\}$.


<table>
<thead>
<tr>
<th>templates</th>
<th>bound-free</th>
<th>bound-constant</th>
<th>optional-free</th>
<th>optional-constant</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>zip</td>
<td>state</td>
<td>city</td>
<td>state, bed, bath, price</td>
</tr>
<tr>
<td>$T_2$</td>
<td>city</td>
<td>state</td>
<td>zip</td>
<td>bed, bath, price</td>
</tr>
</tbody>
</table>

**Figure 6.8** Query constraints of realtor.com

**Figure 6.9** Illustration of object search for a web source.

We thus further classify attributes by the following taxonomy:

- **bound vs. optional**: The bound attribute must be specified in the query, while the optional attribute may not be specified.

- **constant vs. free**: The constant attribute can only be queried with values chosen from a set of constants, while the free attribute does not have any constraint in query values.

In Section 6.2, we did not consider any of these restrictions— all attributes thus essentially fall into a single category of optional-free. In contrast, Example 16 illustrates how attributes in our example interface (Figure 6.7) can be classified into four different categories.

**Example 16**: The interface in Figure 6.7 can query over the six attributes, city, state, zip, price, bed, and bath. This query form requires to specify either zip, or the pair of city and state. This form can thus be understood as two alternative query “templates” $T_1$ and $T_2$: $T_1$ requires zip as bound and the rest as optional, while $T_2$ requires city and state as bound and the rest as optional. Observe, in either template, attributes state, price, bed, and bath are required to use query values among the ones specified in the menu (*i.e.*, constant attributes),
while the rest are free attributes. Table 6.8 summarizes the query constraints of these two query templates.

We now develop object search for middleware sources: While the search essentially builds on range queries, as in Section 6.2, we extend it to accommodate bound and constant attributes as well, unlike Section 6.2 supporting only optional-free attributes.

First, to accommodate bound attributes, object search has to generate range queries that specify all the bound attributes. For instance, in $T_1$ (Figure 6.8) where zip is bound, object search should not generate range queries excluding this attribute, e.g., $Q: (\text{price} > 100k) \land (\text{bed} \geq 1)$. Note, such a “unprocessable” query can be generated when histogram $H$ only partitions with respect to price and bed. Thus, to enforce bound attribute restrictions, we simply require histogram to partition on (at least) all bound attributes, so that each bucket corresponds to a range query specifying all bound attributes.

Second, to accommodate constant attributes, object search should generate range queries that use the query values specified in the interface. Recall that, in our interface example (Figure 6.7), attribute bed can only be queried in the form of $\text{bed} \geq b$, such that $b \in \{1, 2, 3, 4, 5\}$. Such restriction can be supported when histogram boundary reflects the value constraints: For instance, histogram can partition attribute bed into five buckets representing five possible values $\{1, 2, 3, 4, 5\}$. To maintain these “constraint-aware” bucket boundaries, self-tuning merging/splitting operations to refine bucket boundaries will not be applied to these constant attributes (and only bucket frequencies will be refined).

Lastly, recall that query predicates for attributes are pre-fixed. For instance, bed can only be queried in the form $\text{bed} \geq b$. Consequently, in implementing evaluate operation, this restriction needs to be enforced. For instance, a bucket $b$ corresponding to range $q: (\text{zip} = 61810) \land (\text{bed} = 2)$ cannot be queried by itself, but rather needs to be transformed into a “processable” alternative $q'$ subsuming $q$, e.g., $q' = (\text{zip} = 61810) \land (\text{bed} \geq 2)$.

Obviously, this restriction results in significant redundant retrieval: In previous example, the retrieval of houses with three or more bedrooms will be simply wasted. More significantly, this restriction may even result in repeated retrieval: For instance, when the next move targets one-bedroom houses, it will generate a query, e.g., $(\text{zip} = 61810) \land (\text{bed} \geq 1)$, that overlaps
significantly with the preceding query $q'$. (We will later discuss how this repeated retrieval can be avoided, applying collective I/O techniques in Section 6.4.)

Figure 6.9 contrasts object search for middleware sources with object search we developed in Section 6.2—First, this histogram supports bound attribute zip, by partitioning with respect to this attribute—A generated range thus naturally specify this attribute. Second, this histogram supports constant attribute (e.g., bed) by deciding bucket boundaries according to the fixed query values, in contrast to arbitrary bucketing for size in Figure 6.6. Lastly, object search support predicate restriction, e.g., of querying bed in the form of bed $\geq b$. For instance, when we evaluate the highest scoring optimum $v_1$, we generate a range query to cover not only the bucket itself that contains $v_1$, but neighboring buckets with more bedrooms as well (shaded region in the Figure 6.6b), to accommodate query predicate restriction. The search will continue until top-$k$ objects thus far score higher than unevaluated buckets.

### 6.4 Discussion: Collective I/O

We now develop a simple extension to object search schemes developed in Section 6.2 and 6.3, to further reduce the access cost. Recall that, in the proposed schemes, a range query is performed at each iteration (i.e., direct I/O). In contrast, we may delay the execution and “collect” I/O requests over multiple iterations: This idea of collective I/O has been explored in the context of optimizing I/O requests in multi-processor environment. The insight is to collect the requests across different processors and combine into an I/O efficient schedule, by reordering or merging the requests, for the given application (e.g., [26]).

This idea can be successfully applied for object search to further reduce the access cost. To develop a concrete scheme, we first study two major factors that determine the overall cost:

First, there is per-query overhead incurring at each query, e.g., rotational delay. Second, there is per-object overhead, proportional to the amount of data retrieved, such as communication cost of retrieving results. However, the impact of these factors towards the overall cost widely varies across scenarios: In querying remote sources, per-object overhead (e.g., communication cost) far outweighs per-query overhead (e.g., local computation at the server). In contrast, when querying local sources, per-object cost is rather inexpensive (e.g., disk block transfer) and thus per-query overhead is relatively more significant.
There are two representative techniques developed in collective I/O literatures. First, *merging* technique merges two or more consecutive requests to reduce the number of I/O requests (as identified as a cost factor above.) Figure 6.10(a) illustrates this technique, which is effective regardless of cost characteristics— It is obviously profitable to give a single I/O request for three consecutive blocks than to give three separate requests (and thus three-fold saving in per-query costs) for the same amount of data. Second, *sieving* technique extends merging technique to coalesce non-contiguous requests (Figure 6.10b). This technique thus has tradeoff: While it reduces the number of requests, it may result in redundant retrieval. Consequently, this technique needs to be applied only when “cost effective”, i.e., when the saving in per-query overhead outweighs the cost of redundant retrieval.

In this section, we propose framework AsOpt-Collective, extending AsOpt to support collective I/O. In particular, we collect range queries across multiple iterations and combine into an I/O efficient schedule, applying merging and sieving. While such extension can achieve speedup, it also slightly complicates our framework. For one thing, how do we decide when to stop collecting range queries and start executing them? Recall that our stopping condition (Figure 6.3) requires up-to-date $K$, i.e., the scores of objects from preceding range queries. However, these scores can be only known when such queries are actually performed, which defeats the purpose of delaying and collecting preceding queries. We address this dilemma, by approximating the scores of objects in $K$ for deciding whether to stop or not. In particular, we
estimated $\mathcal{K}$ by sampling—That is, for each bucket $b_i: (r_i, f_i)$, we generated $f_i$ random samples that fall within the hyper-rectangle $r_i$ and updated $\mathcal{K}$ with the scores of the samples. Once the stopping condition is satisfied, we stop collecting queries and generate I/O-efficient scheduling, by merging contiguous queries, or sieving non-contiguous queries if “cost effective”. Figures 6.10(c) and (d) illustrate how the notion of “contiguous” queries can be straightforwardly extended for hyper-rectangles: Rectangle $r_i$ and $r_j$ are contiguous if the minimum bounding rectangle (MBR) of the two is in fact the union of $r_i$ and $r_j$. In this case, merging the two range queries into its MBR will not result in any redundant retrieval.

Similarly, we define sieving of $r_i$ and $r_j$ as cost effective, when the saving in per-query overhead outweighs the cost redundant retrieval (e.g., the cost of retrieving objects in the shaded region in Figure 6.10d).

The most straightforward way to execute these merging/sieving over the $M$ queries collected is to generate all possible groupings and choose the best. However, such procedure incurs prohibitive cost of $O(2^M)$. We thus propose a greedy scheme—We first eliminate queries subsumed by others (to eliminate repeated retrieval observed in Section 6.3). Then, we first merge all contiguous query pairs, and then keep on greedily sieving the most cost-effective non-contiguous pairs with minimal redundant retrieval until no more cost-effective sieving can be done.

6.5 Experiments

This section reports our experiments, quantifying the accuracy and efficiency of AsOpt using a benchmark query over a real-life middleware sources. (To contrast, we compare with the same query over a local DBMS as well.)
In quantifying the efficiency and accuracy of framework AsOpt, we use the following metrics: First, to evaluate efficiency, we quantify the relative access cost of AsOpt to that of naive approach performing a sequential scan of the entire data (framework Exhaustive), both in terms of response time and the number of tuples touched (numtuples). Second, to evaluate accuracy, we use the two metrics of score deviation and rank deviation, measuring how the top-k answers from AsOpt $A = \{a_1, \ldots, a_k\}$ deviate from the exact results $K = \{c_1, \ldots, c_k\}$ in terms of score and ranking respectively:

- **Score deviation:** The score deviation metric quantifies how the scores of $A$ deviate from the exact results $K$. That is, $sdev(A, K) = \sum_{i=1}^{k} \frac{sdif f(a_i)}{k}$, when

$$sdif f(a_i) = \begin{cases} 0 & \text{if } a_i \in K \\ F(c_k) - F(a_i) & \text{otherwise.} \end{cases}$$

This metric is in the range of 0 to 1, with 0 being the most desirable (i.e., perfect match).

- **Rank deviation:** The rank deviation metric quantifies how the actual rank of $a_i$, which we denote as $rank(a_i)$, deviates from those in $K$. That is,

$$rdev(A, K) = \sum_{i=1}^{k} \frac{rdif f(a_i)}{N k},$$

$$rdif f(a_i) = \begin{cases} 0 & \text{if } a_i \in K \\ rank(a_i) - k & \text{otherwise.} \end{cases}$$

This metric is also in the range of 0 to 1, with 0 being the most desirable (i.e., perfect match).

Observe, these measures complement the standard precision measure, i.e.,

$$precision(A, K) = \sum_{i=1}^{k} diff(a_i),$$

when $diff(a_i) = \begin{cases} 1 & \text{if } a_i \in K \\ 0 & \text{otherwise.} \end{cases}$

In contrast to precision scoring $0 \forall a_i \notin K$, score and rank deviation measures can further differentiate them with fuzzy degree of deviation (in score and ranking respectively).

All our experiments were conducted with a Pentium III 933 MHz machine with 256M RAM. Our implementation of AsOpt and histograms adopts Python programming language. As we base our object search on multi-dimensional range queries, we build histograms and index structures to efficiently support such queries: For histograms, we implement a multi-dimensional Equi-Depth histogram [36], generated by recursively partitioning the data domain, one dimension at a time, into buckets enclosing the same number of tuples, e.g., the
larger of 20 and the number of distinct values in this evaluation. For access structure, we build a multi-dimensional B+-tree index whose search key is the concatenation of multiple attributes (as reference [4] similarly concluded using their automatic tuning wizard).

We now study the performance of AsOpt over real-life sources, both Web sources and local databases. Our experiments considered a benchmark scenario of finding top houses around Chicago using a Web source, e.g., www.realtor.com (an authoritative real-estate source.) We also consider a scenario of querying a DBMS, namely PostgreSQL 7.1.3, populated with the houses extracted, by querying all the for-sale houses in Chicago (resulting in \( n = 19148 \) houses with the attributes price, bed, and bath). As a benchmark query, we create one looking for a cheap two-bedroom house within budget (e.g., 200k), while she may negotiate her preference on price for a three-bedroom house (Figure 6.11).

Figures 6.12(a) and (b) compare the total cost of AsOpt and Exhaustive of evaluating this benchmark query over databases, in terms of response time and numtuples respectively.

Observe that AsOpt significantly outperforms Exhaustive: For instance, when \( k = 100 \) (x-axis)
in Figure 6.12(a), the response time (y-axis) of AsOpt is only 10% of that of Exhaustive performing a sequential scan. (Note both axes are logarithmically scaled.) Observe also that, framework AsOpt-Collective implementing our collective I/O techniques (Section 6.4) outperforms AsOpt- Collective I/O techniques enable the cost saving of (up to) 7% (of the response time of Exhaustive). Further, from Figure 6.12(b), we can observe that AsOpt-Collective performs both merging and sieving—When \( k = 1000 \), AsOpt-Collective performs redundant retrieval (e.g., of 1.2% or 143 more objects than AsOpt), which suggests sieving was performed. Note, despite this cost of redundant retrieval, the overall response time of decreases by saving per-query costs (e.g., rotation cost and multiple index traversals), which are relatively significant in database scenarios (as Section 6.4 discussed.)

Meanwhile, Figures 6.12(c) and (d) quantify the accuracy of AsOpt, using the score and rank deviation metrics discussed above. As both AsOpt and AsOpt-Collective touch similar number of tuples (Figure 6.12b), their accuracy nearly coincide in both score and rank deviation measures. Observe also that the results from AsOpt are of comparable quality to the exact results: When \( k = 10 \) in Figure 6.12(c), the scores of our results deviate only by 0.0016 in
average from those of the exact results. Similarly, in Figure 6.12(d), we can observe that the average rank deviation is extremely low as well, e.g., 0.00003 when \( k = 100 \).

We perform the same set of experiments over a Web source as well, as reported in Figure 6.13. For now, we assume we have an accurate histogram (as in databases)—Extensive evaluations with histograms of varying accuracy will soon follow. Observe from Figure 6.13(a) that, in this scenario with high access cost, framework AsOpt demonstrates a more dramatic speedup—For instance, when \( k = 10 \), framework AsOpt saves the response time by 96%, which turns an 30-min query into one with a 1-minute turnaround. Observe also that, our collective I/O extension again significantly saves the overall cost, e.g., up to 10% of the cost of Exhaustive (or 3 minutes) when \( k = 100 \). This saving mostly stems from avoiding repeated and redundant retrieval (Section 6.3).

Meanwhile, it is interesting to observe from Figure 6.13(b) that, framework AsOpt-Collective does not perform any sieving in this scenario. That is, framework AsOpt and AsOpt-Collective retrieve the exact same amount of data objects. The reason is due to its cost characteristic, i.e., per-object overhead \( \gg \) per-query overhead (Section 6.4): In this scenario, sieving is generally discouraged, as the potential benefit of decreasing per-query overhead (e.g., server computation) is significantly outweighed by the extra retrieval cost (involving communication cost).

In Figures 6.13(c) and (d), we evaluate the accuracy of framework AsOpt over Web sources–Observe, the accuracy of AsOpt and AsOpt-Collective coincide in both score and rank deviation measures, by retrieving the exact same set of tuples. Similarly to database
scenarios, framework AsOpt achieves high accuracy—The accuracy is perfect when \( k = 10 \) and 100. When \( k = 1000 \), the score and rank deviation are .01 and .005 respectively.

Lastly, to quantify the practicality of supporting Web scenarios where histogram building/maintenance is significantly expensive, we evaluate framework AsOpt with dynamically refined histograms of varying quality. Recall from Section 6.3 that, framework AsOpt-Collective starts with an initial histogram \( H_0 \), generated under some assumption, e.g., uniform distribution. Whenever a selection query is executed, the histogram is refined based on its selectivity—For convenience, we here notate refined histogram after \( m \) selection queries as \( H_m \). Figure 6.14(a) evaluates the cost of AsOpt-Collective with histogram of varying qualities, in particular with \( H_0, H_{10}, \) and \( H_{50} \), i.e., refined after 0, 10, and 50 (uniform) randomly generated selection queries respectively. For comparison, we also plotted the cost when histogram is accurate (as in Figure 6.13). Note, even with a dummy histogram \( H_0 \), framework AsOpt-Collective performs comparably to AsOpt. However, as the number of preceding selection queries increase, the performance also increases—Observe that, only after uniformly random 50 queries, the refined histogram is comparable to an accurate one. While we evaluated with uniform query distribution, the convergence will quicken if query have localities, that is, if few ranges are heavily queried (e.g., Zipfian query distribution).

Meanwhile, observe from Figure 6.14(b) that, the quality of histogram does not affect the accuracy of results.
CHAPTER 7

Related Work

This chapter discusses recent research efforts that are related to the work presented in this thesis.

7.1 Rank Processing

Existing work on modeling and processing rank queries has been divided into two major paradigms, depending on how query conditions are represented and combined. As discussed, one of such paradigms is score-based model, where query condition is represented as fuzzy predicate mapping each data object into an absolute numerical score, e.g., \( \text{new}[a] = 0.9 \).

Predicate scores of an object are then combined by a user-specified mathematical function such as \( \min \) or \( \text{avg} \). Within the context of score-based model, as we overviewed as preliminaries (Chapter 2), existing works have studied processing algorithms either on top of \([13, 28]\) or inside \([7, 8]\) relational databases, and also in middleware \([5, 12, 19]\) access scenarios. Alternatively, ordering-based model \([31, 32]\) has been recently proposed where predicate is replaced by partial ordering. That is, instead of defining a predicate evaluating each and every object into an absolute score, user specifies relative ordering only for some objects such as “I like house \( a \) better than \( b \) in terms of age”. While such alleviation makes partial-ordering model more “intuitive”, it compromises “expressiveness” in return: First, as orderings are specified only for some (likely minority of) objects, it is not clear how to rank “unspecified” objects among themselves and also over other “specified” objects. Second, due to the lack of absolute quantification, conflicts in multiple orderings cannot be resolved. To illustrate, suppose a user prefers house \( a \) over \( b \) in terms of age, while preferring \( b \) over \( a \) in terms of size. While such trade-off situations are common in real-life queries, \( a \) and \( b \) cannot
be differentiated in the ordering-based model, since the absolute degree of the two conflicting preferences cannot be compared as in score-based model. (To illustrate, reference [33] reports experiment results where 36k objects tie as the top in an anti-correlated dataset of 1 million objects.)

7.2 Rank Learning

For the “user-friendly” rank formulation, we adopt machine learning approach, in particular SVM [44], to learn a quantitative ranking function from a qualitative feedback. SVM has proven highly effective in classification [44, 6, 14]. Reference [24] developed an SVM ordinal regression method, and reference [29] adopted it to optimize search engines. We adopts it here to enable soft queries on relational database systems. References [10, 40, 43] studied SVM selective sampling techniques to improve the learning performance and also provide an interactive retrieval framework. However, they are limited to binary classification (e.g., of whether the image is relevant or not) and thus do not generalize to our rank learning. Our top sampling method enhances the rank learning performance and also provides exploratory samples converging to the final top-k results.
CHAPTER 8

Conclusions

8.1 Summary

Today, the explosion of internet usage has provided users with access to information in an unprecedented scale. This explosion has made the data retrieval problem of finding relevant data more important than ever. Due to the large scale of data, such retrieval benefits from ranked answers, enabling to focus only on a few top results.

This thesis thus studies the problem of supporting ranked queries in the context of “middleware” (e.g., over multimedia subsystems or Web sources). In particular, this thesis presents techniques to support this ranked data retrieval both efficiently and effectively. As the first goal, we pursue to develop efficient algorithms retrieving top-k results with minimal access costs. However, this task is challenging, due the heterogeneous nature of data sources (with varying access capabilities and costs). We address this challenge by developing a cost-based optimization framework, first for an important special case of supporting expensive predicates [12], and then for arbitrary access scenarios [27]. By dynamic search over some space of algorithms, cost-based optimization is general across virtually all cost scenarios, yet adaptive to the specific one at runtime. As a critical component, we develop a comprehensive yet focused algorithm space for top-k queries and study optimization schemes to efficiently identify the optimal one in $\Omega$. This approach naturally complements existing algorithms: (1) By enumerating a comprehensive space of algorithms, our framework not only unifies existing algorithms but also encompasses a larger and more complete space. (2) By dynamic search over such space, our cost-based optimization systematically optimizes with respect to runtime parameters unlike existing algorithms.
As the next goal, we pursue to develop a rank formulation framework to assist users to effectively express their information needs. In particular, we believe, for ranking to be intuitive, the system should allow users to specify it in relative ordering (and not absolute scores), and provide context knowledge of what objects are available in the database to be effectively formulated. In particular, to address these challenges, we pursue an interactive “rank-by-examples” front-end [46]– With interactive sampling and labeling by users, our front-end infers the desired ranking expression. While the integration of learning front-end and processing back-end sounds like a happy marriage, in fact, it is not trivial to incorporate learning machine (e.g., SVM) for rank formulation: Unlike a conventional learning problem of classifying objects into groups, we need to learn a ranking function $F$ for inducing an desired ordering of all objects. Consequently, this thesis develops how to “connect” these conflicting views of ranking (for processing back-end) and classification (for learning front-end), and integrates them into a systematic framework RankFP. Further, we study how to extend rank processing to support any arbitrary ranking functions, as learned from the formulation front-end, by abstracting top-$k$ query answering as an optimization problem.

8.2 Future Work

We now identify future tasks in data retrieval.

8.2.1 Categorization

As an alternative mechanism to ranking enabling users to focus on top results, a data retrieval system can support categorization, such that users can focus only on relevant categories. However, categorization for ad-hoc data retrieval presents some unique challenges that are not addressed in the existing approaches taken by search engines (e.g., Google) or static product catalog (e.g., Amazon): Existing approaches create the category structures a priori, and at runtime, they simply place results under the appropriate categories. As such static categorization is independent of the user’s ad-hoc information need, the distribution of items in the categories is susceptible to skew: some groups can have a very large number of items and some very few. For example, a search on databases on Amazon.com yields around 34000 matches out of which 32580 are under the “books” category. These 32580 items are not
categorized any further and the user is forced to go through the long list to find the relevant items, which defeats the purpose of categorization.

It would be thus interesting to develop techniques to dynamically generate a categorization structure that is most effective in satisfying the information needs of the specific user. A systematic framework for such categorization would require (1) an analytic cost model quantifying the effectiveness of categorization structures and (2) efficient algorithms dynamically identifying the most effective categorization structure.

For summer internships at Microsoft Research, we have investigated this problem [11] by developing a model quantifying the effectiveness of categorizations and algorithms that efficiently compute the most effective structure. Based on this analytical cost model, we could estimate the effectiveness of categorizations using the aggregate knowledge of user behaviors (as collected from MSN House and Home site) and develop an efficient search find the most effective structure. There are in fact, many more interesting future directions to be pursued: For instance, how can we similarly estimate the effectiveness of categorizations from data-inherent information, for applications where workloads are infeasible to get? Or, how can we categorize with respect to a combination of multiple correlated attributes, as we similarly explored in the context of data clustering [39]? Further into the future, how can ranking and categorization systems be integrated?

8.2.2 Fuzzy Joins

Another interesting direction is to support joins in ranked retrieval: Fuzzy joins are essential in rank queries, just as they are in relational queries, to associate objects across relations (e.g., houses and parks as in Query 3) by some fuzzy conditions (e.g., closeness). However, despite its importance, fuzzy joins have been only studied per-application basis, e.g., joins by textual similarity [15] or spatial similarity [25]. Our goal, in contrast, is to provide the unified support for fuzzy joins in general.

To motivate, this section overviews existing fuzzy join applications: In data integration context, reference [15] has applied the fuzzy measure of document similarity to join two similar textual entries across sources, possibly referring to the same entity (e.g., “John Smith” and “John A. Smith”). In particular, to efficiently locate highly similar pairs, they have exploited an inverted index structure to map terms to the tuples containing them. Similarly,
in spatial query context, reference [25] has proposed an efficient spatial join algorithm which retrieves spatial object pairs in the order of their distance, taking advantage of general spatial indices (e.g., R* tree).

Observe that, access methods observed from fuzzy join applications, e.g., inverted and spatial indices, distinguish themselves from the sorted access traditionally assumed in top-k works. To illustrate, consider a spatial join retrieving close house and park pairs. In contrast to sorted access retrieving house and park pairs in the order of close scores, spatial indices will require an anchor object (e.g., a house) to retrieve its high-scored pairs (e.g., closest parks to the given house). We characterize such access as hybrid, as it starts with a random access to an anchor object, followed by sorted accesses to its join pairs with the high predicate scores.

Our initial study presented in Section 3.2.1 proposes a simplistic solution assuming only random accesses for join predicates. In contrast, as the unified and efficient framework for fuzzy joins, we can extend NC to generally schedule sorted and hybrid join accesses as well. While NC should straightforwardly handle new type of access as well, this extension will have implications on cost model and dynamic optimization strategies, which we leave as future works.
REFERENCES


AUTHOR’S BIOGRAPHY

Seung-won Hwang received her B.S. degree in Computer Science from Korea Advanced Institute of Science and Technology in 1998. She started her graduate studies at the University of Illinois at Urbana-Champaign in 1998 and received her M.S. degree in Computer Science in 2000, for her work on I/O optimization techniques. Seung-won continued her Ph.D. studies after her Master’s degree. For her Ph.D. thesis, she proposed and developed novel techniques to support intuitive ranking formulation and efficient rank processing. In the summers of 2003 and 2004, Seung-won worked as a research intern at Microsoft Research, working on automatic categorization techniques for query results, and in the summers of 2000 and 2002, she worked as a research intern at Berkeley National Laboratory, working on clustering techniques for high-dimensional scientific data.