ENABLING DATA RETRIEVAL: BY RANKING AND BEYOND

BY

CHENGKAI LI

B.S., Nanjing University, 1997
M.Eng., Nanjing University, 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, 2007

Urbana, Illinois

Abstract

The ubiquitous usage of databases for managing structured data, compounded with the expanded reach of the Internet to end users, has brought forward new scenarios of *data retrieval*. Users often want to express non-traditional fuzzy queries with soft criteria, in contrast to Boolean queries, and to explore what choices are available in databases and how they match the query criteria. Conventional database management systems (DBMSs) have become increasingly inadequate for such new scenarios.

Towards enabling data retrieval, this thesis first studies how to *fundamentally integrate ranking into databases*. We built RankSQL, a DBMS that provides systematic and principled support of ranking queries. With a new *ranking algebra* and an extended query optimizer for the algebra, RankSQL captures ranking as a first-class construct in databases, together with traditional Boolean constructs. We invented efficient techniques for answering *ad-hoc ranking aggregate queries*. RankSQL provides significant performance improvement over current DBMSs in processing ranking queries and ranking aggregate queries.

This thesis further studies how to *enable retrieval mechanisms beyond just ranking*. Our explorative study in this direction is exemplified by two novel proposals– One is to *integrate clustering and ranking* of database query results; the other is to support *inverse ranking queries* that provide ranks of objects in query context. Injecting such non-traditional facilities into databases presents non-trivial challenges in both defining query semantics and designing query processing methods. We extended SQL language to express such queries and invented partition- and summary-driven approaches to process them.

To my wife, Fen.

To my family.

Acknowledgments

I thank Kevin Chang, my advisor, for the wonderful training I had in my Ph.D. study. I was attracted into database research after taking CS511 Advanced Database Management Systems, which was taught by Kevin in Spring 2001. (The course was numbered as CS411 at that time.) It was only after taking the course that I realized the field of databases is much more exciting, profound, and practical as well than I thought. More importantly, Kevin's keen interest and enthusiasm in teaching and his pedagogical methods truly inspired me. It is the same type of enthusiasm that he devoted into advising helped in making me a true researcher. He has spent a tremendous amount of time training me in research, writing, and giving professional talks. He has been brilliant and insightful in research discussions. He taught me to shoot high and strive for perfection in doing research. His dedication, persistence, and sincerity in research deeply impressed me and have set up the high standards that I want to maintain during my career.

I am grateful to Ihab Ilyas. It has been a great experience to collaborate with him during the last 3 years. Ihab just graduated from Purdue when we started to work together. While Kevin generally leaves me to make progress at my own pace, Ihab always eagerly moves ideas forward with full enthusiasm. I have enjoyed much this combination in our collaboration. As a recent graduate who has made a successful start of his career, Ihab also becomes an immediate role model for me.

I want to thank my thesis committee members, Jiawei Han, Marianne Winslett, and Chengxiang Zhai, for their advices on my thesis research, for their support and encouragement in my job search, and for providing the inspiring environment and the diverse expertise of the Database and Information Systems Laboratory (DAIS) at UIUC.

I owe many thanks to Philip Bohannon, Hank Korth, and P.P.S. Narayan, for the fantastic

internship experiences I had at Bell Labs, for their collaborations, and for supporting me in job search. I thank Min Wang, Yuan-Chi Chang, Lipyeow Lim, and Haixun Wang for my internship at IBM research and for their collaborations that significantly strengthened my dissertation.

I thank the FORWARD group members for giving me an inspiring and pleasant work environment. They are Bin He, Seung-won Hwang, Zhen Zhang, Tao Cheng, Shui-Lung Chuang, Arpit Jain, Govind Kabra, David Killian, Yuping Tseng, Ganesh Agarwal, Amit Behal, Ngoc Bui, Abhilasha Chaudhary, Quoc Le, Tingwei Lin, Kim Cuong Pham, Paul Yuan, and Hengzhi Zhong. I have been fortunate to have an excellent group of fellow students in DAIS and the CS Department, including Deng Cai, Hong Cheng, Shengnan Cong, Xiaohui Gu, Yifei Hong, Changhao Jiang, Jing Jiang, Jingyi Jin, Yoonkyong Lee, Xin Li, Jin Liang, Chao Liu, Zhaoan Liu, Zhaoyu Liu, Robert McCann, Qiaozhu Mei, Apu Praveen, Mayssam Sayyadian, Warren Shen, Bin Tan, Tao Tao, Xuanhui Wang, Dong Xin, Dongyan Xu, Kun Yan, Xifeng Yan, and Xiaoxin Yin. Many thanks go to Rishi Sinha. He generously provided me with his implementation of bitmap index, which helped a lot in my research. It is wonderful to have William Conner as a friend since our internship at Bell Labs. It has always been a pleasure for my wife and myself to explore the food in Chambana with him. I also thank Mohamed Soliman at Waterloo for working together with me on our systems.

My appreciation especially goes to Feng Chen, Hui Fang, Jia Guo, Chao Huang, Haibei Jiang, Lihua Jiang, Vincci Kwong, Xiaoming Li, Jianzhong Liu, Lei Ruan, Xuehua Shen, Mingliang Wei, Bin Yu, Shuo Zhang, and Qingbo Zhu. Most of them are UIUC students and many are in CS. They have given me generous support and countless fun in these years. I thank Long Cao, Huachao Chen, Zhongpin Gu, Mei Han, Feng Li, Jiwen Liu, Yiru Tang, Heng Xu, Jian Xu, Lingzhi Zhang, and Xi Zhu. They have made my initial years at UIUC full of happy memories.

My family has been very supportive of me throughout all these years. I want to thank my parents, my sister, and my grandmothers for their greatest love to me.

I thank my dear wife, Fen Lu, for her endless love and support. I could not possibly have reached this far without her. Nobody else could have made my life equally enjoyable and I am grateful for having my beloved wife with me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The ubiquitous usage of databases for managing structured data, compounded with the expanded reach of the Internet to end users, has brought forward new scenarios of *data retrieval*. Users often want to express non-traditional fuzzy queries with soft criteria (e.g., similarity, relevance, and preference), in contrast to Boolean queries, and to explore what choices are available and how they match the query criteria. We refer to this demand of fuzzy relevance over structured data as *data retrieval*, to parallel the well-established *information retrieval* (IR) over unstructured text. Data retrieval is critical in many emerging applications, including E-commerce, DB/IR integration, decision support, multimedia retrieval, and searching Web databases. While successful in business settings, conventional database management systems (DBMSs) have become increasingly inadequate for such new scenarios.

As a concrete example, imagine a user who is looking for a house in Chicago area. The typical facility available to her through a database is Boolean SQL queries. In terms of *query semantics*, database users are limited by "hard" query conditions or constraints, such as "the price in the range from $200K to $400K", "the number of bedrooms being 4", and so on. Such Boolean queries present a "black and white" world to the user. A house belongs to the query results only if it satisfies the constraints. However, in many situations, the user actually may not have such hard constraints clearly in her mind. Sometimes it is even impossible to come up with such constraints without knowing what the database contains in the first place. In terms of *result organization*, what the user gets is a flat table of query results. On the one hand, the result table may contain too many houses if the query conditions are not restrictive enough. Such an overwhelming table is hard for her to digest. On the other hand, it may contain too few or even no houses if the conditions are

specified in a too restrictive way. Therefore it cannot satisfy her retrieval needs either.

In contrast to pure Boolean queries, it is more desirable if the user is able to specify flexible query conditions such as "A price below $500k is more favorable; But she is willing to pay more for a big house; It is a plus if the house is close to the lake and she wants to avoid airport." We want to support this type of fuzzy retrieval in databases. In terms of *query semantics*, we should allow the users to specify "soft" conditions to capture the notions of preference, similarity, and relevance. In terms of *result organization*, there are many retrieval mechanisms that are more effective than flat tables. For instance, a ranked list of houses can indicate how well they satisfy the above soft conditions, thus the user can just focus on the top several houses, regardless of how many houses there are in the database. As another example, a grouping of the houses by their geographical locations can make it much easier for the user to locate her dream house.

Towards enabling data retrieval, as an important first step, this thesis studies how to fundamentally *integrate ranking into databases*. Ranking has been the central notion in information retrieval, as it naturally provides fuzziness and flexibility in capturing users' query criteria. It is thus immediately an important retrieval facility to be incorporated into database queries. Effective retrieval mechanisms go beyond just ranking. This thesis exploratively studies two such mechanisms, namely *clustering together with ranking*, and *inverse ranking*. Injecting such non-traditional concepts into databases presents significant challenges in both defining query semantics and inventing query processing methods. Below we summarize the technical problems, challenges, and our approaches.

Enabling Data Retrieval by Ranking– RankSQL and Ranking Algebra [67, 68] (Chapter 2): *Ranking* or *top-k* queries aim at finding the best-matching k answers in databases according to user-specified ranking functions, which often are aggregates of multiple criteria. The increasing importance of top-k queries warrants an efficient support of ranking in database management systems and has recently gained continuous attention of the research community [32, 13, 30, 19, 4, 57, 21, 55, 91, 29].

However, a fundamental support of ranking is still missing in DBMSs. Most of the available

solutions to ranking queries are in the middleware scenario [31, 76, 43, 32, 12, 17], or in DBMSs in a "piecemeal" fashion, *i.e.*, focusing on specific types of operators [13, 57, 58] and queries [75, 56], or sitting outside the core of query engines [19, 18, 91, 41, 42, 55, 100]. As the foundation of DBMSs, relational algebra has no notion for ranking. Hence, top-k queries are not treated as first-class query type, losing the advantages of integrating top-k operations with other relational operations. The direct effect is inefficiency in processing such queries. Therefore, the demand of ranking as a first-class citizen in databases poses a significant research challenge.

This thesis presents the RankSQL system, which aims at providing a seamless support and integration of top-k queries with the existing SQL query facility in relational database systems. The general approach of RankSQL is based on extending relational algebra to a ranking algebra. We show that by taking ranking into account as a basic logical property, efficient query processing and optimization techniques can be devised to answer top-k queries. The experimental study on our implementation of RankSQL in POSTGRESQL verifies the effectiveness of the ranking algebra in enabling efficient ranking query plans.

Enabling Data Retrieval by Ranking– Ranking Aggregates [66] (Chapter 3):
Based on the algebraic framework, we further developed techniques for processing ad-hoc ranking (top-k) aggregate queries. Aggregation is a key operation in OLAP (On-Line Analytical Processing), and dominates a variety of decision support applications such as manufacturing, sales, stock analysis, and network monitoring. In aggregate queries, aggregates are computed over the groups of the data. Among potentially large number of groups, often only the ones with certain significance are of interest. Ranking aggregate queries thus look for the top k groups with the highest aggregates. By the exploratory nature of decision support, such queries should be posed in an *ad-hoc* manner with respect to how the data is aggregated and ranked, to enable flexible and expressive data analysis. Moreover, OLAP queries are commonly executed *interactively* so that the users get the results quickly and pose further querying. It is thus crucial to process ad-hoc ranking aggregate queries *efficiently* with large volume of data.

However, such efficient support of ad-hoc ranking is critically lacking in current systems. Most

OLAP query processing techniques focus on *pre-computation* and *full-answer*. Decision support systems maintain pre-computed simple aggregates, thus can only benefit queries over these aggregates. Moreover, the notions of ranking and thus optimization for top-k answers are missing as current systems always provide full answers and do not optimize for the small retrieval size k.

Our idea is to design a new non-blocking grouping and aggregation operator. It incrementally draws tuples from the underlying query tree, generates top groups in the ranking order, and stops processing (prunes early) as soon as the required top k groups are obtained. The key in realizing this goal is to find some good *order* of producing tuples (among many possible orders) that can guide the query engine toward processing the most promising groups first, and exploring each group only when necessary. We develop principles that guide the query processor toward a *provably optimal* tuple schedule. Guided by the principles, we propose a new execution framework, which enables efficient query plans that are both *group-aware* and *rank-aware*.

Supporting Retrieval Mechanisms Beyond Ranking– Integrating Clustering with Ranking [69] (Chapter 4):

Data retrieval essentially mandates *result exploration*, for users to explore what choices are available in the database, and how they match the query criteria. We ask: What exploration functions shall we support? While we want to equip SQL-based querying with such exploration, the answers seem to, interestingly, lie in the design of SQL itself. The **order-by** and **group-by** both stand out as the pillars for result organization.

To begin with, recent works in top-k queries have attempted to generalize **order-by**, from "crispy" result *ordering* by attribute values to "fuzzy" *ranking* of matching qualities. As a parallel step, we propose to generalize **group-by**, from "crispy" grouping by attribute equality to "fuzzy" grouping or *clustering* by object proximity. Furthermore, to form a more complete suite of solutions for data retrieval, we propose to integrate the clustering and ranking of database query results.

The challenges in pushing non-conventional facilities like clustering and ranking together into Boolean queries go far beyond just language extensions. Without novel query processing methods,

a straightforward approach is to do full ranking and clustering after materializing Boolean results, leading to inefficiency. Our solutions are built upon summary-based clustering and ranking using dynamically constructed data summary, incorporating Boolean conditions at query time. We have implemented this framework by utilizing bitmap index to construct such summary on-the-fly and to integrate Boolean filtering, clustering, and ranking. Our results show that this approach significantly outperforms the straightforward approach.

Supporting Retrieval Mechanisms Beyond Ranking– Inverse Ranking [65] (Chapter 5): We identify a novel and interesting type of queries, *inverse ranking queries*, which return the ranks of given tuples among query context. Such queries are useful in many places. For instance, a credit card company may be interested in the standing of a new customer among her peers, in order to determine her credit line. While ranking has gained significant attention from the community, inverse ranking query has not been studied so far, in contrast to its usefulness.

A straightforward *exhaustive* approach of processing inverse ranking queries is to fully materialize the results of a Boolean query, *i.e.*, the context of the ranking, and then count the number of tuples whose ranking scores are higher than the score of the object in question. Such an approach can be inefficient, as the query only asks for the rank of a certain tuple, while the full Boolean results are indeed made.

We propose and define the query model and SQL language extension to express inverse ranking queries. We further introduce a general *partition-and-prune* framework for processing them. The framework embraces implementation methods that exploit common data structures in databases, as well as a novel method that utilizes bitmap index built over ranking functions. The results of experimental study show that our algorithms can be significantly more efficient than the straightforward method.

While inverse ranking query is compelling by itself, we find that its dual form, *quantile query*, is also important. A quantile query returns the results at certain ranking positions according to a ranking function. It locates the quantiles as a fast sketch of the query results, thus helps users in the continuous exploration of the data. Moreover, quantile points have significant statistical meanings.

This thesis is the first that studies such quantile queries in the general context of Boolean queries.

In summary, this thesis makes the following contributions:

- **Systems:** We built RankSQL, a RDBMS that fundamentally supports ranking as a first-class construct, through the foundation of ranking algebra.
- **Concepts:** We proposed the ideas of integrating clustering with ranking and enabling inverse ranking in database queries and designed SQL extensions for defining these queries. To the best of our knowledge, ours is the first in the literature to make such investigations.
- **Frameworks and Algorithms:** We developed novel and efficient query processing frameworks and algorithms for ranking queries and ranking aggregate queries, clustering with ranking, and inverse ranking queries. The frameworks and algorithms also integrate with the conventional Boolean query constructs.
- **Evaluations:** We evaluated our techniques through extensive experiments. The results show that our approaches achieve significant performance improvements over existing approaches, and provide insights into the tradeoffs of our algorithms.

The rest of the thesis is organized as follows. Chapter 2 introduces the ranking algebra and query optimization techniques in RankSQL. Chapter 3 discusses the principles and framework for processing ad-hoc ranking aggregate queries. In Chapter 4, we describe the proposal of integrating clustering and ranking in databases queries. Chapter 5 presents our study of inverse ranking queries. We introduce our initial release of the RankSQL system in Chapter 6 and describe the architecture, challenges, and implementation techniques. We review related work in Chapter 7. Finally, Chapter 8 concludes the thesis and discusses some open issues that warrant further research.

Chapter 2

Data Retrieval by Ranking: Rank-Relational Algebra and Optimization

In this chapter, we present the RankSQL system, which aims at providing a seamless support and integration of ranking with the existing SQL facility in relational database systems. Ranking or top-k queries provide only the top k query results, according to a user-specified ranking function, which in many cases is an aggregate of multiple criteria. The following is an example of top-k query.

Example 1: Consider user Amy, who wants to plan her trip to Chicago. She wants to stay in a hotel, have lunch in an Italian restaurant (condition c_1: $r.cuisine$=Italian), and walk to a museum after lunch; the hotel and the restaurant together should cost less than $\$100$ (c_2: $h.price$+$r.price$<100); the museum and the restaurant should be in the same area (c_3: $r.area$=$m.area$). Furthermore, to rank the qualified results, she specifies several ranking criteria, or "predicates"– for low hotel price, with p_1: $cheap(h.price)$; for close distance between the hotel and the restaurant, with p_2: $close(h.addr, r.addr)$; and for matching her interests with the museum's collections, with p_3: $related(m.collection, \text{“}dinosaur\text{”})$. These ranking predicates return numeric scores and the overall scoring function sums up their values. The query is shown below in POSTGRESQL syntax.

select	$*$
from	Hotel h, Restaurant r, Museum m
where	c_1 **and** c_2 **and** c_3
order by	p_1 + p_2 + p_3
limit	k

■

With current relational query processing capabilities, the only way to execute the previous query is to: (1) consume all the records of the three inputs; (2) join the three inputs and materialize

the whole join results; (3) evaluate the three predicates p_1, p_2, and p_3 for each valid join result; (4) sort the join results on $p_1 + p_2 + p_3$; and (5) report only the top k results to the user. Processing the query in this way suffers from the following problems:

- The three inputs can be arbitrarily large, hence joining these inputs can be very expensive. Moreover, it may be infeasible to assume that we can consume the whole inputs, *e.g.*, if these inputs are from external sources such as Web databases.
- The user is not interested in a total order of all possible combinations (*hotel*, *restaurant*, *museum*). Hence, the aforementioned processing is an overkill with unnecessary overhead.
- The ranking predicates can be very expensive to compute, and hence should be evaluated only when they affect the order (rank) of the results. Current query processing must evaluate all the predicates against every valid join result to be able to sort these results.

Such inefficient processing is mainly due to the lack of fundamental support of ranking queries. As the foundation of RDBMSs, relational algebra has no notion for ranking. Therefore, supporting ranking queries as a first-class query type is a significant research challenge.

Our proposed general approach is based on extending relational algebra to be rank-aware. In the rest of this chapter, we show that by taking ranking into account as a basic logical property, efficient query processing and optimization techniques can be devised to answer top-k queries such as the one in Example 1. We summarize the contributions of RankSQL as follows:

- **Extended algebra:** We propose a "rank-relational" algebra, by extending relational algebra to capture ranking as a first-class construct.
- **Ranking query execution model:** We present a pipelined and incremental execution model, enabled by the rank-relational algebra, to efficiently process ranking queries.
- **Rank-aware query optimization:** We present a rank-aware query optimizer, by addressing the key challenges in plan enumeration and cost estimation, to construct efficient ranking query plans.

We conduct an experimental study on our initial implementation of RankSQL in POSTGRESQL, for verifying the effectiveness of the extended algebra in enabling the generation of efficient ranking plans, and for evaluating the validity of our cardinality estimation method in query optimization.

The rest of the chapter is organized as follows. We start in Section 2.1 by defining and motivating ranking queries as first-class construct. Section 2.2 introduces the rank-relational algebra. Section 2.3 introduces the execution model and physical implementation of ranking query plans. We present our proposed rank-aware query optimization in Section 2.4. We describe the experimental evaluation in Section 2.5.

2.1 Ranking Query Model

This section defines rank-relational queries (Section 2.1.1), and motivates the need for supporting ranking as a first-class construct (Section 2.1.2).

2.1.1 Rank-Relational Queries

A rank-relational query Q, as illustrated by Example 1, is a traditional SPJ (select-project-join) query augmented with ranking predicates. Conceptually, such queries have the "canonical" form of Eq. 2.1 in terms of relational algebra:

$$Q = \pi_* \lambda_k \tau_{\mathcal{F}(p_1,\ldots,p_n)} \sigma_{\mathcal{B}(c_1,\ldots,c_m)} (R_1 \times \cdots \times R_h) \tag{2.1}$$

That is, upon the product of the base relations ($R_1 \times \ldots \times R_h$), the following two types of operations are performed, before the top k tuples (which we denote by λ_k) with projected attributes (as π_* indicates) are returned as the results.

- **Filtering**: a *Boolean function* $\mathcal{B}(c_1, \ldots, c_m)$ filters the results by the *selection* operator $\sigma_{\mathcal{B}}$ (*e.g.*, $\mathcal{B} = c_1 \wedge c_2 \wedge c_3$ for Example 1), and

- **Ranking**: a *monotonic scoring function* $\mathcal{F}(p_1, \ldots, p_n)$ ranks the results by the sorting[1] operator $\tau_{\mathcal{F}}$ (*e.g.*, $\mathcal{F} = p_1 + p_2 + p_3$ for Example 1).

Formally, Q returns k top tuples ranked by $\mathcal{F}$, from the qualified tuples $R_{\mathcal{B}} = \sigma_{\mathcal{B}(c_1,\ldots,c_m)}(R_1 \times \cdots \times R_h)$. Each tuple u has a *predicate score* $p_i[u]$ for every p_i and an overall *query score* $\mathcal{F}(p_1, \ldots, p_n)[u] = \mathcal{F}(p_1[u], \ldots, p_n[u])$. As a result, Q returns a sorted list $\mathcal{K}$ of k top tuples[2], ranked by $\mathcal{F}$ scores, such that $\mathcal{F}[u] \geq \mathcal{F}[v]$, $\forall u \in \mathcal{K}$ and $\forall v \notin \mathcal{K}$. As a standard assumption, $\mathcal{F}$ is monotonic, *i.e.*, $\mathcal{F}(x_1, \ldots, x_n) \geq \mathcal{F}(y_1, \cdots, y_n)$ when $\forall i : x_i \geq y_i$. Note that we use summation as the scoring function throughout Chapter 2, although $\mathcal{F}$ can be other monotonic functions such as multiplication, weighted average, and so on.

Observe that, as Example 1 shows, a rank-relational query has four types of predicates: For filtering, as traditionally supported, the query has *Boolean-selection* predicates (*e.g.*, c_1) and *Boolean-join* predicates (*e.g.*, c_2, c_3). For ranking, according to our proposal, it has *rank-selection* predicates (*e.g.*, p_1, p_3) and *rank-join* predicates (*e.g.*, p_2).

We note that, the new ranking predicates, much like their Boolean counterparts, can be of various costs to evaluate: Some predicates may be relatively cheap, *e.g.*, p_1 may simply be attribute or expression such as $(200 - h.price) \times 0.2$. However, in general, predicates can be expensive as they can be user-defined or built-in functions. For instance, p_1 may as well require accessing online sources (*e.g.*, a Web hotel database) for the *current* price; p_2 may involve comparing $h.addr$ with $r.addr$ according to geographical data; and p_3 may perform an information retrieval style operation to evaluate the relevance.

Our goal is to support such rank-relational queries efficiently. As our discussion above reveals, such queries add a *ranking* dimension to query processing and optimization, which in many ways parallels the traditional dimension of *filtering*: While filtering restricts tuple "*membership*" by applying a function $\mathcal{B}$ of Boolean selection or join predicates, ranking restricts "*order*" by applying a function $\mathcal{F}$ of corresponding ranking predicates. While Boolean predicates can be of various

[1]Note that sorting is defined in the *extended* relational algebra to model the **order-by** of SQL.

[2]More rigorously, it returns $min(k, |R_{\mathcal{B}}|)$ tuples.

costs, ranking predicates share the same concern. We thus ask, while conceptually parallel, are they both well supported in RDBMSs?

2.1.2 Ranking as First-Class Construct

Unlike Boolean "filtering" constructs, which are essentially supported in RDBMSs, the same support for "ranking" is clearly lacking. To motivate, observe that as Eq. 2.1 shows, relational algebra provides the *selection* operator $\sigma_{\mathcal{B}}$ for filtering, and the *sorting* operator $\tau_{\mathcal{F}}$ for ranking. However, as we will see, there is a significant gap between their support in current systems.

Relational algebra models Boolean filtering, *i.e.*, $\sigma_{\mathcal{B}(c_1,\ldots,c_m)}$, as a *first-class* construct in query processing. (Such filtering includes both selections on a single table as well as joins.) With algebraic support for optimization, Boolean filtering is virtually never processed in the canonical form (of Eq. 2.1)– Consider, for instance, $\mathcal{B} = c_1 \wedge c_2$, for c_1 as a selection over R and c_2 a join condition over $R \times S$. The algebra framework supports *splitting* of selections (*e.g.*, $\sigma_{c_1 \wedge c_2}(R \times S) \equiv \sigma_{c_1}\sigma_{c_2}(R \times S) \equiv \sigma_{c_1}(R \bowtie_{c_2} S)$) and *interleaving* them with other operators (*e.g.*, $\sigma_{c_1}(R \bowtie_{c_2} S) \equiv \sigma_{c_1}(R) \bowtie_{c_2} S$). These algebraic equivalences thus enable query optimization to transform the canonical form into efficient query plans by splitting and interleaving.

However, in a clear contrast, such algebraic support for optimization is completely lacking for ranking, *i.e.*, $\tau_{\mathcal{F}(p_1,\ldots,p_n)}$. The sorting operator τ is "monolithic": The scoring function $\mathcal{F}(p_1, \ldots, p_n)$, unlike its Boolean counterpart $\mathcal{B}(c_1, \ldots, c_m)$, is evaluated at its *entirety*, after the rest of the query is materialized– essentially as "naïve" as in the canonical form.

Such naïve *materialize-then-sort* scheme should not be the only choice– in fact, in many cases, it can be prohibitively expensive. If we want only the top k results, full materialization may not be necessary. As we shall see in Section 2.3, ranking predicates can significantly cut the cardinality of intermediate results. Moreover, all the ranking predicates have to be evaluated against every results of the full materialization under this naïve scheme. With the various costs, it may be beneficial in many cases to evaluate ranking predicates one by one, and interleave them with Boolean filtering. Thus, in a clear departure from the monolithic sorting τ, we believe rank-relational queries call for

essentially supporting ranking as a first-class construct– in parallel with filtering. Such essential support, as we have observed, consists of two *requirements*:

1. **Splitting:** Ranking should be evaluated in stages, predicate by predicate– instead of monolithic.

2. **Interleaving:** Ranking should be interleaved with other operators– instead of always after filtering.

There are two major challenges in supporting ranking as a first-class operation. *First*, as foundation, we must extend relational algebra to handle ranking and define algebraic laws for equivalence transformations (Section 2.2). Meanwhile, to realize the algebra, we must define the corresponding query execution model and physical operators in which "rank-relations" are processed incrementally (Section 2.3). *Second*, we need to generalize query optimization techniques for integrating the parallel dimensions of Boolean filtering (*e.g.*, join order selection) and ranking (Section 2.4).

2.2 Rank-Relational Algebra

To enable rank-aware query processing and optimization, we extend relational algebra to be rank-relational algebra, where the relations, operators, and algebraic laws "respect" and take advantage of the essential concept of "rank". In this section, we define rank-relational model (Section 2.2.1) and extend relational algebra (Section 2.2.2). The new rank-relational algebra enables and determines our query execution model and operator implementations. We also discuss several laws (Section 2.2.3) of the new algebra to lay the foundation of query optimization.

2.2.1 Rank-Relations: Ranking Principle

To fundamentally support ranking, the notion of *rank* must be captured in the relational data model. Thus, to start with, we must extend the semantics of relations to be rank-aware. In this extended

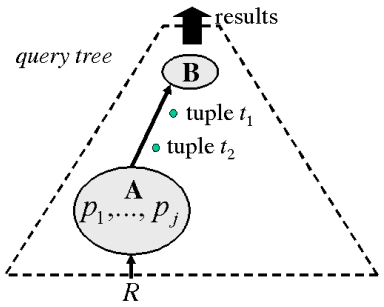


Figure 2.1: Ranking of intermediate relations.

model, we define *rank-relation* as a relation with its tuples *scored* (by some ranking function) and *ordered* accordingly.

In this model, how should we rank a relation?– Note that our algebra extension is to support rank-relational queries: Given a scoring function $\mathcal{F}(p_1, \ldots, p_n)$ for such a query (as in Eq. 2.1), what are the rankings of tuples as they progress in processing? Consider a base relation R. Figure 2.1 conceptually illustrates the query tree. To begin with, when no ranking predicate p_i is evaluated, R as tuples "on the disk" has an arbitrary order. As the splitting requirement (Section 2.1.2) motivates, these ranking predicates will generally be processed in stages. We thus ask, when some predicates, say $\mathcal{P} = \{p_1, \ldots, p_j\}$ (for $j < n$) are evaluated (Figure 2.1, cloud "A"), what should be the ranking? Note that although the final results are to be ranked by $\mathcal{F}(p_1, \ldots, p_n)$, at this stage we do not have the complete scores of all the predicates. Therefore, we want to define a *partial* ranking of tuples by their current incomplete scores, so that the resulted order is consistent with their "desired" order of further processing. As queries are evaluated incrementally by "iterators" (Section 2.3), this ranking will order the output tuples to subsequent operations (Figure 2.1, cloud "B"). Thus, refer to Figure 2.1, when should a tuple t_1 be ranked before t_2? It turns out that we have the following *ranking principle*.

Property 1 (Ranking Principle): With respect to a scoring function $\mathcal{F}(p_1, \ldots, p_n)$, and a set of evaluated predicates $\mathcal{P} = \{p_1, \ldots, p_j\}$, we define the *maximal-possible* score (or upper-bound) of a tuple t, denoted $\overline{\mathcal{F}}_{\mathcal{P}}[t]$, as

$$\overline{\mathcal{F}}_{\mathcal{P}}(p_1, \ldots, p_n)[t] = \mathcal{F}\left(\begin{array}{ll} p_i = p_i[t] & \text{if } p_i \in \mathcal{P} \\ p_i = 1 & \text{otherwise}^3. \end{array} \quad \forall i\right)$$

Given two tuples t_1 and t_2, if $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] > \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$, then t_1 must be further processed if we necessarily further process t_2 for query answering. ■

The proof is straightforward. Intuitively, the maximal-possible score of a tuple t defines what t *can* achieve, with $\mathcal{P}$ already evaluated, by assuming unknown predicates are of perfect scores. (Since $\mathcal{F}$ is monotonic, this substitution will result in its upper bound.) Therefore when $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] > \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$, whatever score t_2 can achieve, t_1 can possibly do even better. Refer to Figure 2.1, the subsequent operation "B" cannot process only t_2 but not t_1. Therefore it is desirable that "B" draws outputs from "A" in this order, *i.e.*, t_1 should precede t_2. By this ranking principle, Definition 1 formalizes rank-relations.

Definition 1 (Rank-Relation): A *rank-relation* $R_{\mathcal{P}}$[4], with respect to relation R and monotonic scoring function $\mathcal{F}(p_1, \ldots, p_n)$, for $\mathcal{P} \subseteq \{p_1, \ldots, p_n\}$, is the relation R augmented with the following ranking induced by $\mathcal{P}$.

- (**Scores**) The *score* for a tuple t is the maximal-possible score of t under $\mathcal{F}$, when the predicates in $\mathcal{P}$ have been evaluated, *i.e.*, $\overline{\mathcal{F}}_{\mathcal{P}}[t]$. It is an implicit attribute of the rank-relation.

- (**Order**) An order relationship $<_{R_{\mathcal{P}}}$ is defined over the tuples in $R_{\mathcal{P}}$ by ranking their scores, *i.e.*, $\forall t_1, t_2 \in R_{\mathcal{P}}$: $t_1 <_{R_{\mathcal{P}}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$. ■

Note that, when there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by unique tuple IDs.

[3]More rigorously, it is the application-specific maximal-possible value of p_i. We assume 1 without losing generality.

[4]To be more rigorous, it should be notated as $R^{\mathcal{F}}_{\mathcal{P}}$. We omit $\mathcal{F}$ for simplicity.

TID	a	b	p_1	p_2
r_1	1	2	0.9	0.65
r_2	2	3	0.8	0.5
r_3	3	4	0.7	0.7

(a) R

TID	a	b	p_1	p_2
r'_1	1	2	0.9	0.65
r'_2	3	4	0.7	0.7
r'_3	5	1	0.75	0.6

(b) R'

TID	a	c	p_3	p_4	p_5
s_1	4	3	0.7	0.8	0.9
s_2	1	1	0.9	0.85	0.8
s_3	1	2	0.5	0.45	0.75
s_4	4	2	0.4	0.7	0.95
s_5	5	1	0.3	0.9	0.6
s_6	2	3	0.25	0.45	0.9

(c) S

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1\}}$
r_1	1	2	1.9
r_2	2	3	1.8
r_3	3	4	1.7

(d) $R_{\{p_1\}}$

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_2\}}$
r'_2	3	4	1.7
r'_1	1	2	1.65
r'_3	5	1	1.6

(e) $R'_{\{p_2\}}$

TID	a	c	$\overline{\mathcal{F}_2}_{\{p_3\}}$
s_2	1	1	2.9
s_1	4	3	2.7
s_3	1	2	2.5
s_4	4	2	2.4
s_5	5	1	2.3
s_6	2	3	2.25

(f) $S_{\{p_3\}}$

Figure 2.2: Examples of rank-relations.

The extended *rank-relational algebra* generally operates on rank-relations. Thus, base relations, intermediate relations, and the results are all rank-relations. That is, rank-relations are *closed* under the algebra operators, which Section 2.2.2 will define, since all operators will account for the new ranking property (in addition to "membership"). Note that a base or intermediate relation, when no predicates are evaluated ($\mathcal{P} = \phi$), is consistently denoted R_ϕ or simply R. On the other hand, when $\mathcal{P} = \{p_1, \ldots, p_n\}$, the partial score is effectively complete, resulting in the final ranking with respect to $\mathcal{F}$.

Example 2: As our running example, Figure 2.2(a)-(c) show three base relations, R, R', and S (*i.e.*, R_ϕ, ${R'}_\phi$, S_ϕ), with their schemas, tuple IDs, and ranking predicate scores. Note that tuple IDs and predicate values are shown for pedagogical purpose only. (These predicate values are unknown until evaluated.) For our discussion, as we will illustrate various operators, we assume R and R' have the same schema (*e.g.*, to be unioned later) and predicates. S is used later to show join operator. Suppose the scoring function for R and R' is $\mathcal{F}_1 = \sum(p_1, p_2)$, and for S is $\mathcal{F}_2 = \sum(p_3, p_4, p_5)$. Figure 2.2(d)-(f) show three rank-relations, $R_{\{p_1\}}, {R'}_{\{p_2\}}, S_{\{p_3\}}$, with tuples ranked by maximal-possible scores. ■

Rank: μ, with a ranking predicate p • $t \in \mu_p(R_\mathcal{P})$ iff $t \in R_\mathcal{P}$ • $t_1 <_{\mu_p(R_\mathcal{P})} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t_2]$
Selection: σ, with a boolean condition c • $t \in \sigma_c(R_\mathcal{P})$ iff $t \in R_\mathcal{P}$ and t satisfies c • $t_1 <_{\sigma_c(R_\mathcal{P})} t_2$ iff $t_1 <_{R_\mathcal{P}} t_2, i.e., \overline{\mathcal{F}}_\mathcal{P}[t_1] < \overline{\mathcal{F}}_\mathcal{P}[t_2]$
Union: $\cup$ • $t \in R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ or $t \in S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}\cup S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$
Intersection: $\cap$ • $t \in R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ and $t \in S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}\cap S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$
Difference: $-$ • $t \in R_{\mathcal{P}_1} - S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1}$ and $t \notin S_{\mathcal{P}_2}$ • $t_1 <_{R_{\mathcal{P}_1}-S_{\mathcal{P}_2}} t_2$ iff $t_1 <_{R_{\mathcal{P}_1}} t_2, i.e., \overline{\mathcal{F}}_{\mathcal{P}_1}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1}[t_2]$
Join: $\bowtie$, with a join condition c • $t \in R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2}$ iff $t \in R_{\mathcal{P}_1} \times S_{\mathcal{P}_2}$ and satisfies c • $t_1 <_{R_{\mathcal{P}_1}\bowtie_c S_{\mathcal{P}_2}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}_1\cup\mathcal{P}_2}[t_2]$

Figure 2.3: Operators defined in the algebra.

2.2.2 Operators

We next extend the relational-algebra operators for manipulating rank-relations. Recall that, by Definition 1, a rank-relation $R_\mathcal{P}$ essentially possesses two logical properties– 1) *membership* as defined by the relation R, and 2) *order* induced by predicates $\mathcal{P}$ (with respect to some scoring function $\mathcal{F}$). For manipulating these two properties, we extend relational algebra by adding a new rank operator μ and generalizing the existing operators to be "rank-aware". Figure 2.3 summarizes the definitions of these operators, and Figure 2.4 illustrates them with examples (as continued from Example 2), which we explain below in more details.

New Operator μ: For supporting ranking as a first-class construct, we propose to add a new operator, *rank* or μ. As Section 2.1.2 motivated, our goal is to satisfy the two requirements: splitting and interleaving. Essentially, we must be able to evaluate ranking predicates (p_i's in $\mathcal{F}$) one at a time– thus ranking is effectively split and can be interleaved with other operations.

The new rank operator (μ) is thus a critical basis of our algebra. As Figure 2.3 defines, $\mu_p(R_\mathcal{P})$ evaluates an additional predicate p upon rank-relation $R_\mathcal{P}$, ordered by evaluated predicate set $\mathcal{P}$ as Definition 1 states, and produces a new order by $\mathcal{P} \cup \{p\}$– That is, by definition, $\mu_p(R_\mathcal{P}) = R_{\mathcal{P} \cup \{p\}}$. For instance, when μ_{p_2} operates on $R_{\{p_1\}}$ in Figure 2.2(d), the result rank-relation is shown in Figure 2.4(a), which equals $R_{\{p_1,p_2\}}$. Note that $R_{\{p_1,p_2\}}$ is already the final result for ranking R with $\mathcal{F}_1$ because $\mathcal{F}_1 = \sum(p_1, p_2)$.

Extended Operators $\pi, \sigma, \cup, \cap, -, \bowtie$**:** We extend the original semantics of existing operators with rank-awareness, and thus enable the interaction between the new μ and traditional Boolean operations. As we will see, in the extended algebra, the operations will now be aware of and compute on dual logical properties– both membership (by Boolean predicate) and order (by ranking predicate). (Note that we omit projection π in Figure 2.3, since it is obvious. We also omit the discussion on Cartesian-product since it is similar to join.)

To begin with, unary operators such as *selection* (and π not shown in Figure 2.3) process the tuples in the input rank-relation as in their original semantics, and simply maintains the same order as the input. Thus, in our notation, $\sigma_c(R_\mathcal{P}) \equiv (\sigma_c R)_\mathcal{P}$. That is, the selection with c on $R_\mathcal{P}$ manipulates only the *membership* of R, by applying c, and maintains the same *order* as induced by $\mathcal{P}$. An example is shown in Figure 2.4(b).

Further, most binary operators, such as *union* ($\cup$), *intersection* ($\cap$), and *join* ($\bowtie$), perform their normal Boolean operations, and at the same time output tuples in the "aggregate" order of the operands– Such aggregate order is induced by *all* the evaluated predicates from both operands. Thus, for instance, $R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2} \equiv (R \cap S)_{\{\mathcal{P}_1 \cup \mathcal{P}_2\}}$, which similarly holds for $\cup$ and $\bowtie$. Examples are shown in Figure 2.4(c), (d), and (f).

Finally, *difference* ($-$) outputs tuples in the order of the outer input operand– since the other is effectively discarded. Thus, $R_{\mathcal{P}_1} - S_{\mathcal{P}_2} \equiv (R - S)_{\mathcal{P}_1}$. An example is shown in Figure 2.4(e).

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1,p_2\}}$
r_1	1	2	1.55
r_3	3	4	1.4
r_2	2	3	1.3

(a) $\mu_{p_2}(R_{\{p_1\}})$

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1\}}$
r_2	2	3	1.8
r_3	3	4	1.7

(b) $\sigma_{a>1}(R_{\{p_1\}})$

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1,p_2\}}$
r_1/r'_1	1	2	1.55
r_3/r'_2	3	4	1.4

(c) $R_{\{p_1\}} \cap R'_{\{p_2\}}$

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1,p_2\}}$
r_1/r'_1	1	2	1.55
r_3/r'_2	3	4	1.4
r'_3	5	1	1.35
r_2	2	3	1.3

(d) $R_{\{p_1\}} \cup R'_{\{p_2\}}$

TID	a	b	$\overline{\mathcal{F}_1}_{\{p_1\}}$
r_2	2	3	1.8

(e) $R_{\{p_1\}} - R'_{\{p_2\}}$

TID_R	TID_S	a	b	c	$\overline{\mathcal{F}_3}_{\{p_1,p_3\}}$
r_1	s_2	1	2	1	4.8
r_1	s_3	1	2	2	4.4

(f) $R_{\{p_1\}} \bowtie_\theta S_{\{p_3\}}$, where θ is $R_{\{p_1\}}.a = S_{\{p_3\}}.a$, $\mathcal{F}_3 = \sum(p_1, p_2, p_3, p_4, p_5)$.

Figure 2.4: Results of operators.

2.2.3 Algebraic Laws

Query optimizers essentially rely on algebraic equivalences to enumerate or transform query plans in search of efficient ones. In the extended rank-relational model and algebra, as the dual logical properties dictate, algebraic equivalences should result in not only the same membership but also the same order. By definition of our algebra, as just discussed, we can assert many algebraic equivalence laws. As we extended the algebra specifically to support ranking, Figure 2.5 gives several such equivalences relevant to ranking. Essentially, these laws concretely state the new freedom of *splitting* and *interleaving*, thus achieving our motivating requirements (Section 2.1.2)– That is, the rank-relational algebra indeed supports ranking as first-class, in parallel with Boolean filtering. These laws are directly from the definition of the algebra, therefore we omit the proof and only briefly discuss their usage in query optimization. In particular, we explain the laws specifically centering around our two requirements:

First, *rank splitting*: Proposition 1 allows us to split a scoring function with several predicates (p_1, ..., p_n) into a series of rank operations (μ_1, ..., μ_n). This splitting is useful for processing the

Proposition 1: Splitting law for μ • $R_{\{p_1,p_2,\ldots,p_n\}} \equiv \mu_{p_1}(\mu_{p_2}(\ldots(\mu_{p_n}(R))\ldots))$
Proposition 2: Commutative law for binary operator • $R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2} \equiv S_{\mathcal{P}_2} \Theta R_{\mathcal{P}_1}, \forall \Theta \in \{\cap, \cup, \bowtie_c\}$
Proposition 3: Associative law • $(R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2}) \Theta T_{\mathcal{P}_3} \equiv R_{\mathcal{P}_1} \Theta (S_{\mathcal{P}_2} \Theta T_{\mathcal{P}_3}), \forall \Theta \in \{\cap, \cup, \bowtie_c{}^a\}$
Proposition 4: Commutative laws for μ • $\mu_{p_1}(\mu_{p_2}(R_{\mathcal{P}})) \equiv \mu_{p_2}(\mu_{p_1}(R_{\mathcal{P}}))$ • $\sigma_c(\mu_p(R_{\mathcal{P}})) \equiv \mu_p(\sigma_c(R_{\mathcal{P}}))$
Proposition 5: Pushing μ over binary operators • $\mu_p(R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2})$ $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c S_{\mathcal{P}_2}$, *if only* R *has attributes in* p $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c \mu_p(S_{\mathcal{P}_2})$, *if the attributes in* p *is a subset of join attributes in* c • $\mu_p(R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) \cup \mu_p(S_{\mathcal{P}_2})$ • $\mu_p(R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) \cap \mu_p(S_{\mathcal{P}_2})$ • $\mu_p(R_{\mathcal{P}_1} - S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) - S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) - \mu_p(S_{\mathcal{P}_2})$
Proposition 6: Multiple-scan of μ • $\mu_{p_1}(\mu_{p_2}(R_\phi)) \equiv \mu_{p_1}(R_\phi) \cap_r \mu_{p_2}(R_\phi)$

[a] When join columns are available.

Figure 2.5: Some algebraic equivalence laws.

predicates individually– Our splitting requirement is thus satisfied.

Second, *interleaving*: Propositions 4 and 5 together assert that rank operations can swap with other operators, thus achieving the interleaving requirement. In particular, Proposition 4 deals with swapping μ with other unary operators (μ or σ)– thus, we can schedule μ freely with σ. Further, Proposition 5 handles swapping with binary operators– we can thus push down μ across $\bowtie$, $\cap$, and others.

The new algebraic laws lay the foundation for query optimization of ranking queries as algebraic equivalences define equivalent plans in the search space of query optimizers. As we will see in Section 2.4, these algebraic laws guide the designing of transformation rules in rule-based optimizers, as well as the plan enumeration and heuristics in bottom-up optimizers.

2.3 Ranking Query Plans: Execution Model and Physical Operators

In common database query engines, a query execution plan is a tree of physical operators as *iterators*, which have three interface methods that allow the consumer operator of a physical operator to fetch one result tuple at a time. The three basic interface methods are: (1) *Open* method that initializes the operator and prepares its internal state; (2) *GetNext* method that reports the next result upon each request; (3) *Close* method that terminates the operator and performs the necessary cleanup. During the execution, query results are drawn from the root operator, which draws tuples from underlying operators recursively, till the scan operators. This provides an efficient *pipelining* evaluation strategy, unless the flow of tuples is stopped by a blocking operator such as sort or a blocking join implementation, in which case, intermediate results have to be materialized.

The nature of ranking query lends itself to pipelined and incremental plan execution. We desire that the small number k not only reduces the size of results presented to users, but also allows less work to be done, *i.e.*, we want the execution cost to be proportional to k. In interactive applications, k may be only an estimate of the desired result size or not even specified beforehand. Hence, it is essentially desirable to support *incremental* processing– for returning top results progressively upon user requests.

Unfortunately traditional implementation of ranking by sticking a sorting operation on top of the execution plan is an overkill solution to the problem and can be prohibitively expensive. Such materialize-then-sort scheme is undesirably *blocking*, as the first result is reported after all results (much more than k in general) are produced and sorted. The cost is independent from k and the startup cost is almost equal to the total cost.

Fortunately rank-relational algebra both advocates and enables non-blocking plans. In this section, we show how ranking query plans, consisting of the new and extended operators, execute according to the ranking principle (Property 1) in Section 2.3.1 and present their physical implementations in Section 2.3.2.

2.3.1 Incremental Execution Model

To realize the rank-relational algebra, we extend the common execution model to handle ranking query plans, with two differences from traditional plans. *First*, operators incrementally output rank-relations $R_{\mathcal{P}}$ (Definition 1), *i.e.*, tuple streams pass through operators in the order of maximal-possible scores (upper-bounds) $\overline{\mathcal{F}}_{\mathcal{P}}[t]$ with respect to the associated ranking predicate set $\mathcal{P}$. As the ranking principle indicates, it is desirable that t_1 precedes t_2 in further processing if $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] > \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$. *Second*, the query has an explicitly requested result size, k. The execution stops when k results are reported or no more results are available.

For an operator to output its intermediate result as a rank-relation, as Definition 1 requires, the output must be in the order by the associated predicate set. That is, a tuple can be output to the upper operator if its upper-bound score is guaranteed to be higher than the upper-bound of any future output tuple. Therefore the key capability of a rank-aware operator is to decide if enough information has been obtained from its input tuples in order to *incrementally* produce the next ranked output tuple.

To illustrate, consider a μ_p operator upon the input $\mathcal{R}_{\mathcal{P}}$ as the result of its preceding operator x. In order to produce outputs in the correct order by $\mathcal{P}\cup\{p\}$, μ_p cannot immediately output a tuple t once t is obtained from x, because there *may* exist some t' such that $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t] < \overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t']$, although $\overline{\mathcal{F}}_{\mathcal{P}}[t] \geq \overline{\mathcal{F}}_{\mathcal{P}}[t']$ (therefore t' has not been "drawn" from x yet). Instead, μ_p has to evaluate $p[t]$ to get $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t]$ and to buffer t in a *ranking queue* (implemented as priority queue) that maintains tuples in the order by $\mathcal{P}\cup\{p\}$. At any time, the top tuple t in the queue can be output when a t' is drawn from x such that $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t] \geq \overline{\mathcal{F}}_{\mathcal{P}}[t']$, thus $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t] \geq \overline{\mathcal{F}}_{\mathcal{P}}[t'] \geq \overline{\mathcal{F}}_{\mathcal{P}}[t'']$ for any future tuple t'' from x. Note that $\overline{\mathcal{F}}_{\mathcal{P}}[t''] \geq \overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t'']$ according to Definition 1. Therefore μ_p can conclude that $\overline{\mathcal{F}}_{\mathcal{P}\cup\{p\}}[t] \geq \overline{\mathcal{F}}_{P\cup\{p\}}[t'']$, thus it can output t.

Example 3: We continue the running example in Example 2, to show how ranking query plans execute differently from traditional plans. Consider a very simple top-k query over base table S (Figure 2.2(c)) and the ranking function $\mathcal{F}_2$ in Example 2,

select $*$ **from** S **order by** $p_3 + p_4 + p_5$ **limit** 1.

Figure 2.6 illustrates three equivalent plans. Plan (a) is a traditional plan consisting of a sorting and a sequential scan operator. It scans tuples from S, evaluates all predicates (p_3,p_4,p_5) for each tuple, buffers and sorts them based on their scores till all tuples are scanned. Plan (b) is a new plan enabled by the rank-relational algebra, with an index scan followed by two μ operators. The index scan accesses tuples in the order of p_3 values, where p_3 can be as simple as attribute or as complex as external or built-in function. (Such index is supported in DBMSs such as POSTGRESQL.)

In these plans, the rank-relation R above each operator op contains the tuples that have ever been processed by op. The portion of R in gray color is the incremental output rank-relation from op to its upper operator op', thus is the incremental input rank-relation to op'. Therefore the rank-relation R' above op' contains the same tuples as the gray portion of R, although may in different order, since op' can apply one more predicate and thus result in a new order. For example, consider μ_{p_4} in plan (b). It processed 3 tuples (s_2,s_1,s_3) during execution. Among them, s_2 and s_1 were drawn to μ_{p_5}, which processed 2 tuples (s_2 and s_1) and output s_2 as the top-1 answer since μ_{p_5} is the top operator in the plan tree.

Note that the order of tuples in the rank-relations are decided by semantics, according to the definition of rank-relation (Definition 1) and operators (Section 2.2.2). For example, μ_{p_4} must output tuples in the order by $\overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}$ since p_3 is accessed by the underlying operator $idxScan_{p_3}(S)$ and p_4 is evaluated by μ_{p_4}. Therefore s_2 must precede s_1 when output from μ_{p_4} since $\overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}[s_2] = 2.75 > \overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}[s_1] = 2.5$.

We further illustrate how tuples flow, still using plan (b) as an example. The operator μ_{p_5} first draws s_2 from μ_{p_4}, then evaluates $p_5[s_2]$ and gets $\overline{\mathcal{F}}_{2_{\{p_3,p_4,p_5\}}}[s_2] = 2.55$. At this point μ_{p_5} cannot output s_2 yet (refer to our explanation in the paragraph right above Example 3). Therefore μ_{p_5} buffers s_2 in its ranking queue and draws the next tuple, s_1, from μ_{p_4}. It is sure at this point that μ_{p_5} can output s_2 as the top answer (again, refer to the paragraph above Example 3). After evaluating $p_5[s_1]$ and getting $\overline{\mathcal{F}}_{2_{\{p_3,p_4,p_5\}}}[s_1] = 2.4$, s_1 is buffered. The execution goes on in this way to get more query results. Other operators in plan (b) work in the same way and the whole plan

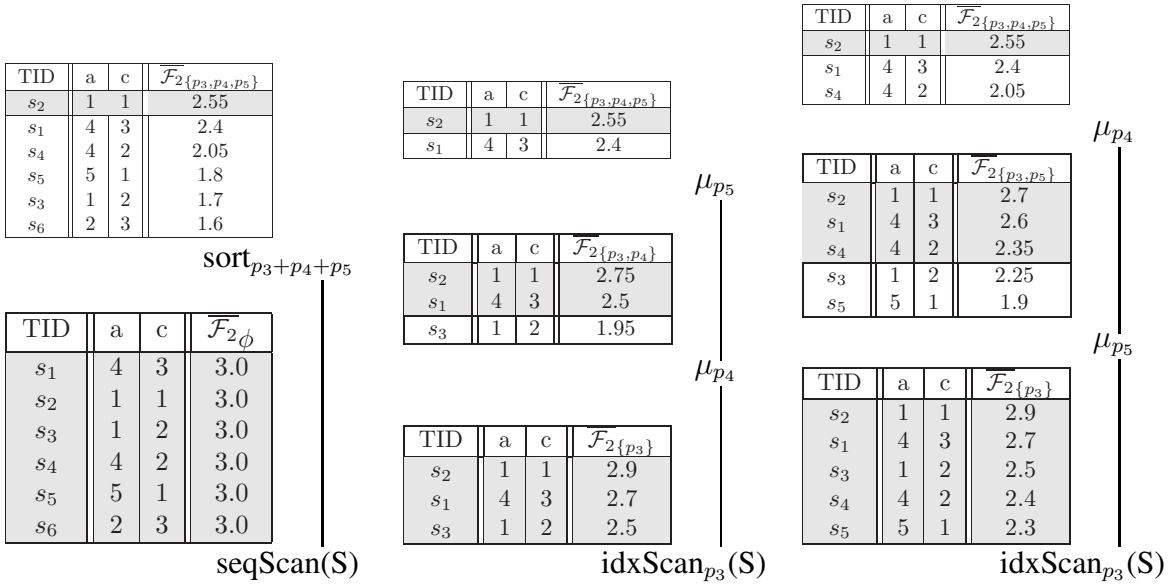


Figure 2.6: Ranking query plans vs. traditional plan.

tree is executed in pipeline by recursively drawing tuples, resulting in the diagram in Figure 2.6(b).

■

Binary operators such as join work in the same principle as μ, except that they obtain inputs from two streams, combine the scores from the two inputs to get updated upper-bound scores for seen and unseen output tuples.

Illustrated by the previous example, the execution model indicates that rank-aware operators are *selective*, *i.e.*, they reduce the cardinality of intermediate results as they do not output all of their processed tuples. For instance, the selectivity of μ_{p_4} in Figure 2.6(b) is $2/3$ as the rank-relation above it clearly shows.

In contrast to traditional operators, the selectivity of rank-aware operators is *context-sensitive*. The reason is, selectivities of rank-aware operators are dependent on k, and furthermore, cannot be assumed to be independent from their locations in a whole plan, as assumed for selection and join selectivities traditionally. For instance, plan (c) in Figure 2.6 is similar to plan (b) except that the order of μ_{p_4} and μ_{p_5} is reversed. The selectivities of μ_{p_4}, μ_{p_5}, and $idxScan_{p3}(S)$ in this plan are $1/3$, $3/5$, and $5/6$ respectively, while they are $2/3$, $1/2$, and $3/6$ in plan (b) (remember there are 6 tuples in S).

Being selective enables operators to both reduce the evaluation of predicates that have various costs and reduce the cost of join, therefore ranking query plans do not need to materialize a query, in contrast to the traditional materialize-then-sort scheme of processing ranking queries. This makes ranking query plans much more efficient than traditional ones, which can be prohibitively expensive. Moreover, different scheduling and interleaving of rank-aware operators will result in different number of tuples being processed, therefore query optimizers have to non-trivially explore the new type of ranking plans (Section 2.4). Furthermore, the context-sensitiveness of selectivities indicate that cardinality estimation of these ranking plans will be challenging (Section 2.4.2).

Example 4: Continuing Example 3, this example shows that ranking query plans (Figure 2.6(b)(c)) outperform traditional plans (Figure 2.6(a)) and different ranking plans have different costs, thus it calls for query optimization.

Assume the costs of predicates p_3, p_4, and p_5 are C_3, C_4, and C_5, then the predicate evaluation cost of plan (a) is $6(C_3 + C_4 + C_5)$ since it has to evaluate all predicates for every tuples. It also needs to scan 6 tuples. (If there are more tuples in S, it has to scan all of them.) In plan (b), μ_{p_5} evaluates p_5 over two tuples (s_2, s_1) and μ_{p_4} evaluates p_4 over three tuples (s_2, s_1, s_3). Therefore the predicate evaluation cost of plan (b) is $3C_4 + 2C_5$. It only needs to scan 3 tuples. The predicate evaluation cost of plan (c) is $3C_4 + 5C_5$ and it needs to scan 5 tuples, according to similar analysis. ■

2.3.2 Implementing Physical Operators

We must implement new physical operators in order to realize the execution model. Fortunately previous works on top-k queries in middleware and relational settings provide a good basis to leverage. Below we briefly discuss the implementation of operators.

The implementation of μ is straightforward from Example 3 and it is a special case (because it schedules one predicate) of the algorithms (*MPro* [17], *Upper* [12]) for scheduling random object

accesses in middleware top-k query evaluation. The implementation of $\bowtie_C$ adopts the *HRJN* (hash rank-join) and *NRJN*(nested-loop rank-join) algorithms in [57] [58], which are built upon symmetrical hash join [54, 95] or hash ripple join [47].

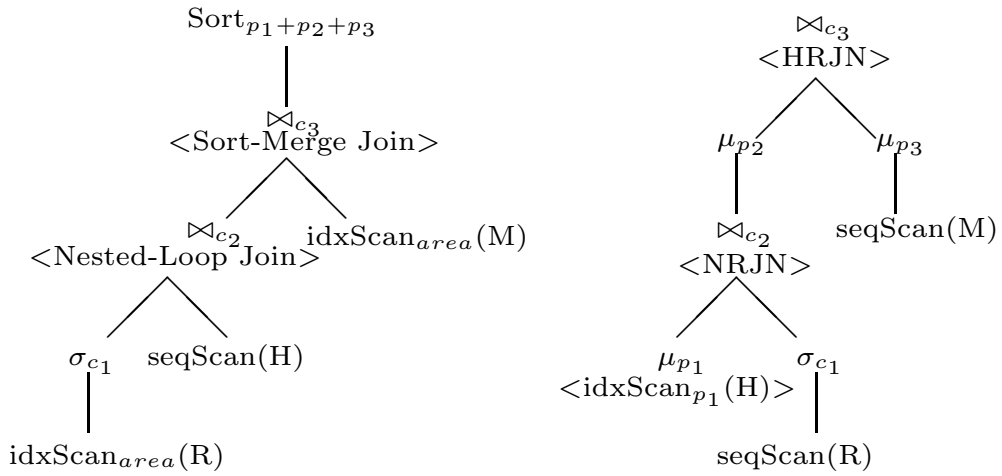
New algorithms for other operators are similarly implemented. Use $\cap$ under set semantics as an example. Traditionally it has to exhaust both input streams to ensure that no duplicate tuple is output. However, with the input streams being ranked, it can judge if duplicates of a tuple may have appeared or may be seen in the future according to the predicate values of that tuple. Therefore it can output ranked results incrementally.

As another example, *scan* must be provided as a physical operator although it is not in relational algebra. Index-scan can be used to access tuples of a table in the order of some predicate p when there exists an index such as B+tree on p. (Thus we name it *rank-scan*.) Such index can be available when p is some attribute, expression, or function, as all are supported in practical DBMSs such as POSTGRESQL. Moreover, scan-based selection can be used to combine a scan operator on p with a selection operator on selection condition c when a multi-key index on p and c is available.

2.4 A Generalized Rank-Aware Optimizer

The task of cost-based query optimization is to transform a parsed input query into an efficient execution plan, which is passed to the query execution engine for evaluation. The transformation task is usually accomplished by examining a large *search space* of plans. The optimizer utilizes a *plan enumeration algorithm* that can efficiently search the plan space and prune plans according to their estimated execution costs. To estimate the cost of a plan, the optimizer adopts a *cost model*.

Extending relational algebra to support ranking as introduced in Section 2.2 and Section 2.3 has direct impact on query optimization. In this section, we motivate the need for extending the query optimizer to support ranking and study the significant challenges associated with the extension. Then we show how to incorporate ranking into practical query optimizers used by real-world database systems.



(a) A traditional plan. (b) A ranking plan.

Figure 2.7: Two alternative plans for Example 1.

The rank-relational algebra enables an extended plan space with plans that cannot be expressed traditionally. For instance, for the query in Example 1, traditional optimizers only allow materialize-then-sort plans such as the one in Figure 2.7(a). In contrast, the rank-relational algebra enables equivalent plans such as the one in Figure 2.7(b). The equivalence is guaranteed by the algebraic laws in Figure 2.5. First, the ranking function in the $sort$ operator is split into μ_{p_1}, μ_{p_2}, μ_{p_3} by Definition 1 and Proposition 1 of Figure 2.5. The μ operators are pushed down across join operators by Proposition 4 and 5. Note that μ_{p_1} is combined with scan operation to form an $idxScan$. Such splitting and interleaving may achieve significant improvements in performance as discussed in Section 2.3.1.

In order to fully incorporate the rank-relational algebra into a cost-based query optimizer, we must address the impact of the extended search space on plan enumeration and costing. In *plan enumeration*, the desirability of splitting and interleaving ranking predicates requires the optimizer to fully explore the extended plan space for generating efficient query plans. In *plan costing*, cardinality estimation must be performed for the rank-aware operators for costing and pruning plans.

There are two categories of cost-based query optimizers used by real-world database systems,

namely the top-down rule-based optimizers exemplified by Volcano [39] and Cascade [38], and the System R-style bottom-up dynamic programming optimization framework [86].

In Volcano and Cascade, *transformation* and *implementation* rules are the two key constructs used for searching the plan space. The transformation rules transform between equivalent algebraic expressions, and the implementation rules map logical operators into physical implementations to realize a plan tree. For extending rule-based optimizers with the rank-relational algebra, the algebraic laws presented in Section 2.2.3 naturally enable the introduction of new transformation rules to enumerate ranking plans. Implementation rules can be devised to trigger the mapping of physical algorithms presented in Section 2.3.2. Cost estimation in top-down optimizers can apply similar techniques for extending bottom-up optimizers since it only costs complete plans instead of subplans.

Extending bottom-up optimizers to incorporate ranking, however, is more challenging as plans are constructed and pruned in bottom-up fashion without global information of a complete plan. Therefore focusing on bottom-up optimizers, we show how to extend the System-R style bottom-up dynamic programming (DP) approach for plan enumeration (Section 2.4.1) and how to cost and prune plans during enumeration (Section 2.4.2).

2.4.1 Two-Dimensional Plan Enumeration

We take a principled way to extend DP plan enumeration by treating ranking predicates as another dimension of enumeration in addition to Boolean predicates, based on the insight that the ranking (order) relationship is another logical property of data, parallel to membership (Section 2.1.1). Recall that, by Definition 1, a rank-relation $R_{\mathcal{P}}$ essentially possesses two logical properties: Boolean membership (R) and ranking order ($\mathcal{P}$). In a ranking query plan, new ranking predicates are only introduced in μ operators. Therefore the predicate set $\mathcal{P}$ of a subplan, *i.e.*, the μ operators in a subplan, determines the order, just like how join conditions (together with other operations) determine the membership. Moreover, for the same logical algebra expression, the optimizer must be able to produce various plans that schedule and interleave μ operators, and to select the most

efficient plan, just like it must be able to select the best join order. This *dimensional enumeration* approach not only reflects the fact that order and membership are dual logical properties in the rank-relational model, but also takes advantages of the dynamic programming paradigm in reducing searching costs. Furthermore, the *dimensional enumeration* subsumes the conventional plan enumeration for join order selection and does not affect the optimization of non-ranking plans.

The concept of dimensional enumeration is general and extensible for naturally including more dimensions, *e.g.*, ordering other operators such as selection, union, intersection, *etc.* For example, scheduling selection predicates is traditionally considered less important than join order selection and is rather handled by heuristics such as selection pushdown. Under the situation that it is necessary to handle such task, as motivated in [24, 53, 23], dimensional enumeration can incorporate the scheduling of both selection and ranking predicates by treating Boolean predicates as another dimension. We focus on how to integrate the scheduling of ranking predicates and join order selection and omit the consideration of other operators.

The DP 2-dimensional enumeration algorithm is shown in Figure 2.8. (The format is similar to [23].) For each subplan, we define its *signature* (S_R, $S_\mathcal{P}$) as the pair of two logical properties, the set of relations S_R and the set of ranking predicates $S_\mathcal{P}$ in the subplan. Subplans with the same signature result in the same rank-relation. The algorithm first enumerates the number of joined relations, $||S_R||$, then the number of ranking predicates, $||S_\mathcal{P}||$. Plans with the signature $(S_R, S_\mathcal{P})$ are generated by joining two plans with the signature $(S_{R1}, S_{\mathcal{P}1})$ and $(S_{R2}, S_{\mathcal{P}2})$ (joinPlan), adding a μ_p upon a plan with the signature $(S_R, S_\mathcal{P} - \{p\})$ (rankPlan), or using a scan operator (scanPlan). Based on the principle of optimality, no sub-optimal subplan can be part of the optimal execution strategy, hence for all the plans with the same signature, only the best plan is kept.

Example 5: We illustrate how the algorithm optimizes a simple query over the tables in Figure 2.2,

select $*$ **from** R, S **where** $R.a = S.a$ **order by** $p_1 + p_3 + p_4$ **limit** k.

In Figure 2.9, each row contains the best plans for signatures of the same size, with one best plan

Procedure 2_Dimensional_Enumeration

1: //The 1st dimension: join size
2: **for** $i \leftarrow 1$ **to** h **do**
3: **for** each $S_R \subseteq \{R_1, ..., R_h\}$ s.t. $||S_R|| = i$ **do**
4: **for** each pair S_{R1}, S_{R2} s.t. $S_R = S_{R1} \cup S_{R2}$, $S_{R1} \neq \phi$, $S_{R1} \cap S_{R2} = \phi$ **do**
5: //The 2nd dimension: ranking predicates
6: $P \leftarrow$ all predicates that are evaluable on S_R
7: **for** $j \leftarrow 0$ **to** $||P||$ **do**
8: **for** each $S_P \subseteq P$ s.t. $||S_P|| = j$ **do**
9: $bestPlan \leftarrow$ a pseudo plan with cost +∞
10: **for** each pair S_{P1}, S_{P2} s.t. $S_P = S_{P1} \cup S_{P2}$, $S_{P1} \cap S_{P2} = \phi$ **do**
11: $plan \leftarrow$ a pseudo plan with cost +∞
12: **if** $S_{R2} \neq \phi$ **then**
13: $plan \leftarrow$ joinPlan (bestPlan(S_{R1},S_{P1}), bestPlan(S_{R2},S_{P2}))
14: **if** $S_{R2} = \phi$ and $S_{P2} = \{p\}$ **then**
15: $plan \leftarrow$ rankPlan(bestPlan(S_{R1} ,S_{P1}), μ_p)
16: **if** $i = 1$ and $||S_{P1}|| \leq 1$ and $||S_{P2}|| = \phi$ **then**
17: $plan \leftarrow$ scanPlan(S_{R1},S_{P1})
18: **if** cost($plan$)$\leq$cost($bestPlan$) **then**
19: $bestPlan \leftarrow plan$
20: bestPlan(S_R, S_P) $\leftarrow bestPlan$
21: **return** bestPlan($\{R_1, ..., R_h\}, \{p_1, ..., p_n\}$)

Figure 2.8: The 2-dimensional enumeration algorithm.

per signature. For instance, row $(2, 1)$ show the best plans for $(\{R, S\}, \{p_1\})$, $(\{R, S\}, \{p_3\})$, and $(\{R, S\}, \{p_4\})$ respectively. We also show the pruned plans (as crossed out) on single table, but omit that for joined relations.

The enumeration starts with signature size $(1, 0)$ to find scan plans for signatures $(\{R\}, \phi)$ and $(\{S\}, \phi)$. Assume that $seqScan$ is kept for both signatures; and $idxScan_a(R)$ and $idxScan_c(S)$ are pruned. The enumeration continues with size $(1, 1)$ to look for plans for $(\{R\}, \{p_1\})$, $(\{S\}, \{p_3\})$, and $(\{S\}, \{p_4\})$. For example, plans for $(\{S\}, \{p_3\})$ can be built by adding μ_{p_3} on top of $seqScan(S)$ or by using $idxScan_{p_3}(S)$. By comparing their estimated costs, the former is pruned. The enumeration proceeds in this way until the final plan is generated. ■

One important detail of System-R algorithm is that multiple plans with the same logical properties may be kept if they have different physical properties. Example physical properties are *inter-*

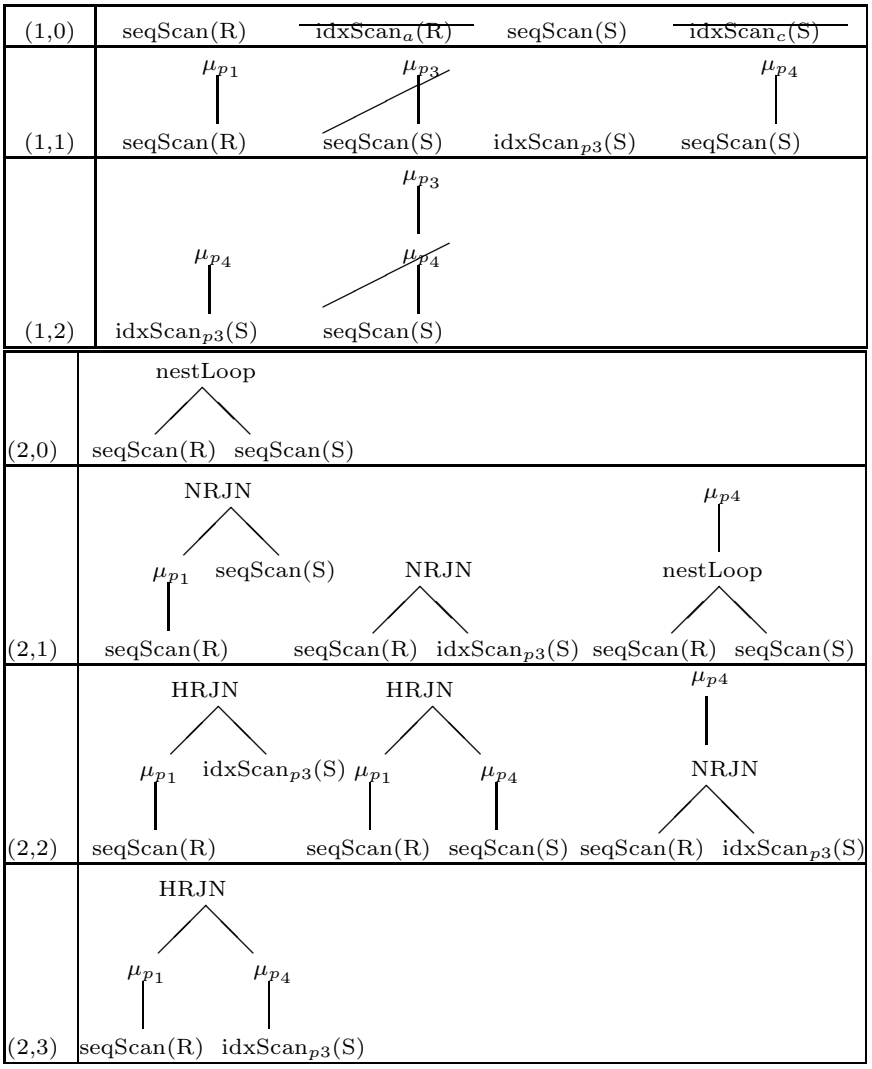


Figure 2.9: Plan enumeration.

esting orders [86] that are potentially beneficial to subsequent operations. For instance, $idxScan_a$ can be kept since its sorted access on $R.a$ can be useful for sort-merge join when R is joined with S. In the dimensional enumeration algorithm, the support of physical properties is not affected. It can keep multiple plans that have different physical properties for the same signature. Note that interesting order will only be possessed by plans with empty predicate set (*i.e.*, $S_{\mathcal{P}} = \phi$), since by definition rank-relations must be output in the order with respect to $\mathcal{P}$, which is not the kind of order that is useful to operators such as sort-merge join.

The 2-dimensional enumeration algorithm is exponential in the number of the ranking pred-

icates as well as the number of relations, as the System-R style algorithm is exponential in the number of relations. As a common practice, query optimizers apply heuristics to reduce the search space. For example, a query optimizer can choose to consider only left-deep join trees and to avoid Cartesian products. Such heuristics are often found effective in improving efficiency and being able to find comparably good query plan.

Therefore, we propose a heuristic to reduce the space on the dimension of ranking predicates, as shown in Figure 2.10. The algorithm in Figure 2.10 modifies that in Figure 2.8 by incorporating the left-deep join heuristic (Line 2) and our new heuristic on the ranking predicate dimension (Line 4). The ranking predicate scheduling heuristic greedily appends μ operators in a sequence instead of considering all valid permutations of μ operators. Given a subplan $plan$, suppose $plan'$ is to be built by adding one μ upon $plan$. The optimizer does not use μ_{p_u} to build $plan'$ if there exists another applicable μ_{p_v} such that appending μ_{p_v} is (likely) better than appending μ_{p_u}. The goodness of appending μ_{p_u} upon $plan$, is based on its selectivity and cost, defined as $rank(\mu_{p_u}) = \frac{1-card(plan')/card(plan)}{cost(\mu_{p_u})}$, where $cost(\mu_{p_u})$ is the evaluation cost of p_u, and $card(plan')$ and $card(plan)$ are the output cardinalities of $plan'$ and $plan$. (This $rank$ should not be confused with the concept of rank in our algebra.) Therefore μ_{p_u} is appended upon $plan$ only if there exists no other applicable μ_{p_v} that has a higher rank. Intuitively the rank of a μ is higher if its cost is lower and its selectivity is smaller, *i.e.*, its power of reducing cardinality is higher. In the formula, $cost(\mu_{p_u})$ is one component of the cost model of μ_{p_u} that should be defined together with its implementation. Techniques for estimating the cardinality of a subplan is presented in Section 2.4.2.

The above greedy scheduling heuristic for ranking predicates is inspired by the $rank$ metric in [53] for scheduling independent selection predicates and the adaptive approach in [7] for ordering correlated filters in streaming data processing. The $rank$ metric in [53] guarantees an optimal fixed order of independent selection predicates, that is, a selection predicate should always be applied before another one if it has higher rank. However, the same property cannot be guaranteed for scheduling μ operators simply because of their context-sensitive selectivities (Section 2.3.1). We adopt $rank$ metric as a heuristic, just like applying left-deep join heuristic, which sacrifices

Procedure 2_Dimensional_Enumeration_with_Heuristics
1: replace line 4 of Figure 2.8 with the following
2: **for** each pair S_{R1}, S_{R2} s.t. $S_R = S_{R1} \cup S_{R2}$, $||S_{R2}|| \leq 1$, $S_{R1} \cap S_{R2} = \phi$ **do**
3:
4: insert the following into Figure 2.8, between line 10 and 11
5: **if** $||S_{R2}|| = 0$, $S_{P2} = \{p_u\}$ and $\exists p_v$ s.t. $p_v \in P - S_P$ and $rank(\mu_{p_v}) > rank(\mu_{p_u})$ **then**
6: continue to line 10

Figure 2.10: Heuristics for improving efficiency.

optimality for efficiency as a common practice of query optimizers.

2.4.2 Costing Ranking Query Plans

The optimizer prunes plans according to their estimated execution costs based on a cost model. The cost model for various operators in real-world query optimizers is quite complex and depends on many parameters, including cardinality of inputs, available buffers, type of access paths and many other system parameters. Although cost model can be very complex, a key ingredient of its accuracy is cardinality estimation of intermediate results.

Cardinality estimation for ranking query plans is much more difficult than that for traditional ones because cardinality information cannot be propagated in a bottom-up way. In conventional query plans, the input cardinality of an operator is independent from the operator itself and depends only on the input subplans. The output cardinality depends only on the size of inputs and the selectivity of the logical operation. In ranking query plans, however, an operator consumes only partial input, therefore the actual input size depends on the operator itself and how the operator decides that it has obtained "enough" information from the inputs to generate "enough" outputs. Hence, the input cardinality depends on the number of results requested from that operator, which is unknown for a subplan during plan enumeration. Note that the number of final results, k, is known only for a complete plan. This imposes a big challenge to System-R style optimizers that build subplans in bottom-up fashion, because the propagation of k value to a specific subplan depends on the location of that subplan in the complete plan.

To address this challenge, we propose a sampling-based cardinality estimation method for rank-aware operators. Let x be the score of the k-th query result tuple. Our technique is based on the intuition that tuples whose upper-bound scores are lower than x do not need to be output from an operator. Although x is unknown during plan enumeration, the sampling method can be used to estimate x, and to further estimate the output cardinality of a subplan.

The optimizer randomly samples a small number of tuples from each table and evaluates all the predicates over each tuple. Note that this step is not necessarily performed every time since it is possible to re-use the predicate values for succeeding queries. To estimate x, before plan enumeration, the optimizer evaluates the original query on the sample using any conventional execution plan, to retrieve k' top results proportional to the sample size. Suppose the sampling ratio is $s\%$, *i.e.*, each tables t_i with original size N_i has a sample size $n_i = N_i \times s\%$, then $k' = \lceil k \times s\% \rceil$. That is, it transforms a top-k query on the database into a top-k' query on the samples. The score of the k'th topmost answer, x', is used as an estimation of x, based on the intuition that k' is proportional to the sample size with respect to k over the database size.

With x', during plan enumeration, the optimizer estimates the output cardinality of a subplan P, $card(P)$, by executing P on the small samples. The results are kept together with P so that there is no need to execute P again when estimating the output cardinality of a future plan that is built based on P. Suppose P outputs u answers that have upper-bound scores above x'. Then $card(P)$ is estimated in the following way:

- $card(P) = u/(s\%)$: if P has only one operator, *i.e.*, a scan operator on a base table.
- $card(P) = u \times card(P')/card_s(P')$: if the top operator of P is a unary operator, on top of a subplan P', which has output cardinality $card_s(P')$ during the execution of P on the sample and an estimated output cardinality $card(P')$ during previous steps of plan enumeration.
- $card(P) = u \times \frac{\frac{card(P_1)}{card_s(P_1)} + \frac{card(P_2)}{card_s(P_2)}}{2}$: if the top operator of P is a binary operator, taking inputs from two subplans P_1 and P_2. P_1 and P_2 have output cardinality $card_s(P_1)$ and $card_s(P_2)$, respectively, during the execution of P on the sample; and estimated output cardinalities

$card(P_1)$ and $card(P_2)$, respectively, during previous steps of plan enumeration.

Our experimental study (Section 2.5) shows that the simple sampling method with a small sample ratio (*e.g.*, 0.1%) gives accurate cardinality estimates. With small sample size, sampling method does not introduce much overhead to query optimization.

Accurate random sampling over joins is known to be difficult [20]. We plan to investigate the possibilities of using techniques such as [20] in future work to improve our sampling method.

2.5 Experiments

We build a prototype of the RankSQL system in POSTGRESQL 7.4.3. We extend the internal representation of tuples to include the implicit ranking score attribute in rank-relational model and implement the rank operator, the rank-aware join, and the rank-scan operators. In this section, we present two sets of experiments that we conducted on the system. The first set compares different execution plans to demonstrate the performance diversity of the plan space, thus motivates the need of query optimization. It also illustrates that under general circumstances, the performance of plans that are only possible in the extended plan space of the new algebra is superior to traditional plan for evaluating top-k queries. The second set of experiments verifies the accuracy of the sampling-based method for estimating the cardinalities of rank-aware operators.

The experiments are conducted on a PC having a 1.7GHz Pentium-4 CPU with 256KB cache, 768MB RAM, and 30GB disk, running Linux 2.4.20 operating system. The `shared_buffers` (shared memory buffer size) and `sort_mem` (internal memory for sorting and hashing) settings in POSTGRESQL are configured as 24MB and 20MB, respectively. We use a synthetic data set of three database tables (A, B, C) having the same size and schema. Table A and B each have one Boolean attribute with 0.4 as their selectivities. The three tables have $2, 2$, and 1 ranking predicates, respectively. The ranking predicates have the same cost. They are implemented as user-defined functions, taking attributes of the tables as parameters. Scores of different ranking predicates are within the range between 0 and 1 and are independently generated by different distributions,

including uniform, normal (with mean 0.5 and variance 0.16), and cosine distributions. Each table has two attributes jc1 and jc2 as join columns.

We use a simple top-k query $\mathcal{Q}$ as shown below in POSTGRESQL syntax. Summation is used as the scoring function $\mathcal{F}$.

$\mathcal{Q}$:
select $*$
from A, B, C
where $A.jc1 = B.jc1$ **and** $B.jc2 = C.jc2$ **and** $A.b$ **and** $B.b$
order by $f1(A.p1) + f2(A.p2) + f3(B.p1) + f4(B.p2) + f5(C.p1)$
limit k

Figure 2.11 illustrates four execution plans for the above query. $Plan1$ is a conventional materialize-then-sort plan, in which *filter* is the physical selection operator and sort-merge join is used as the physical join operator. $Plan2 - 4$ are new ranking query plans. The implementations of μ operator (*rank*), rank-aware join operator (*HRJN*), and rank-scan operator ($idxScan$) were described in Section 2.3.2. In $plan2$, rank-scans are used for accessing base tables and μ is scheduled before join. $Plan3$ uses sequential scan instead of rank-scan. $Plan4$ applies μ operators above normal sort-merge join to replace one of the *HRJN* operators.

2.5.1 Cost of Ranking Execution Plans

In this suite of experiments, we show that the costs of execution plans for top-k queries vary with respect to (among other factors) the number of final results (k, from 1 to $1,000$), the number of tuples in each table (s, from $10,000$ to $1,000,000$), the join selectivity (j, from 0.001 to 0.00001, *i.e.*, the number of distinct values of each join attribute ranges from $1,000$ to $100,000$), and the cost of each ranking predicate (c, from 0 to $1,000$ unit costs).

We performed 4 groups of experiments. The default values of the parameters are $k = 10$, $s = 100,000$, $j = 0.0001$, and $c = 1$. In each group, we vary the value of one parameter and fix the values of the other three parameters. We then execute each plan under these parameter settings

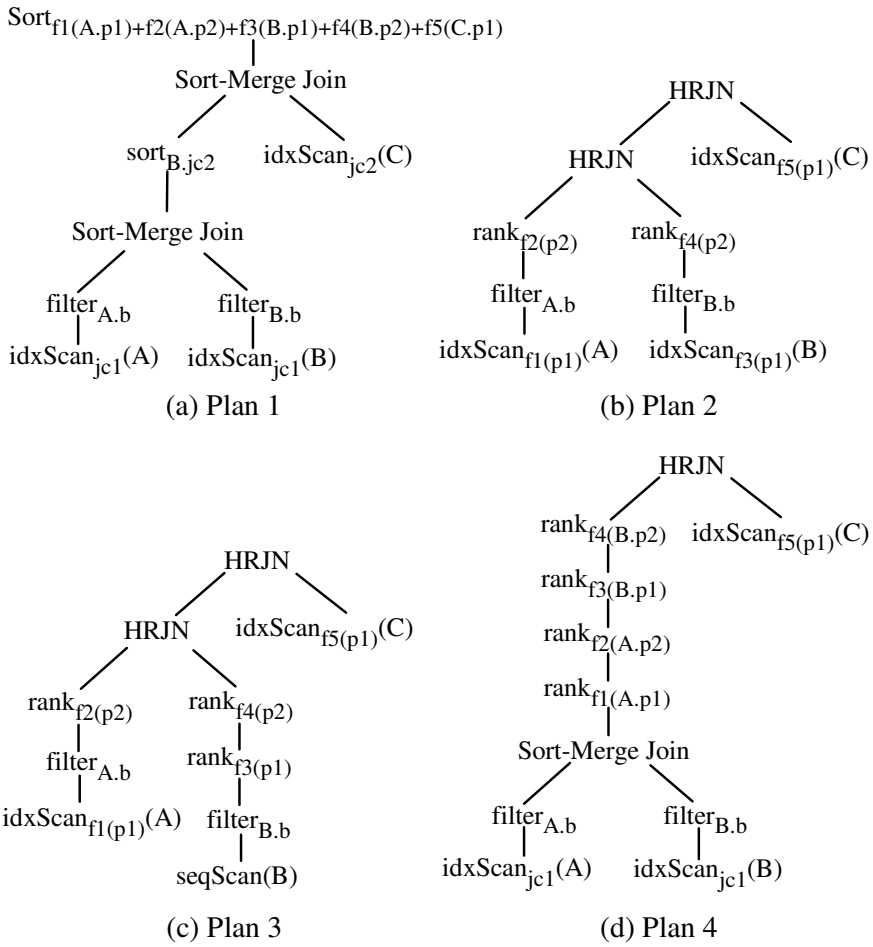
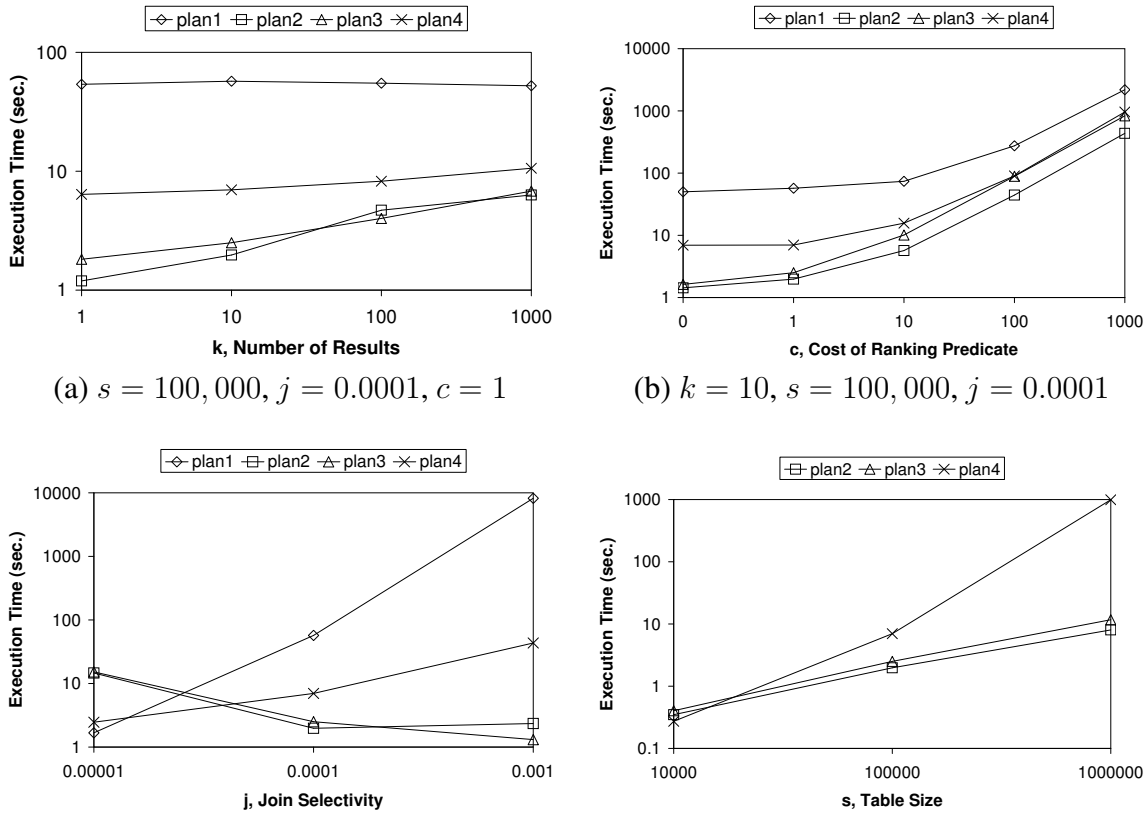


Figure 2.11: Execution plans for query $\mathcal{Q}$.

and measure their execution time. The results are shown in Figure 2.12. (Note that both horizontal and vertical axes are in logarithmic scale.)

The figures illustrate that none of the plans is always the best under all situations. Moreover, different plans can have orders of magnitude differences in their costs. The diversity of plan costs verifies the need of query optimization in choosing efficient plans. Apparently, the traditional plan ($plan1$ in Figure 2.11) is far outperformed by rank-aware plans ($plan2 - 4$ in Figure 2.11). Its performance is only comparable to other plans when the size of tables and requested results are small, when joins are very selective, and when predicates are cheap. In many situations, the traditional plan becomes prohibitively expensive.



(a) $s = 100,000$, $j = 0.0001$, $c = 1$

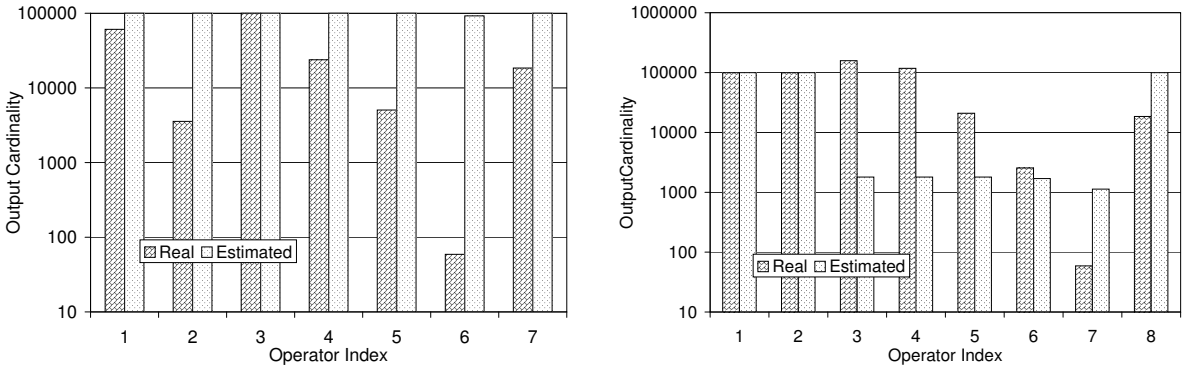
(b) $k = 10$, $s = 100,000$, $j = 0.0001$

(c) $k = 10$, $s = 100,000$, $c = 1$

(d) $k = 10$, $j = 0.0001$, $c = 1$

Figure 2.12: Performances of different execution plans.

Specifically, Figure 2.12(a) shows that the traditional plan for ranking queries is blocking, while the new rank-aware plans are incremental. Figure 2.12(b) illustrates that the cost difference between plans increases (shown as parallel lines in logarithmic scale) together with the cost of predicates. This is because the predicate cost will dominate the plan execution cost while getting larger and the number of predicate evaluations does not change for a given plan when only predicate cost is changing. Figure 2.12(c) shows that the traditional plan is efficient when joins are very selective (thus performing join first will result in very small intermediate results, upon which ranking predicates are evaluated). Finally, Figure 2.12(d) shows that some ranking query plans (*e.g.*, $plan2$) are very efficient even with very large tables, while some others are not. For instance, $plan4$ was relatively acceptable in other situations, but became much less efficient than $plan2$ and $plan3$ when each table has 1 million tuples. Note that we remove $plan1$ from Figure 2.11(d) since it takes days to finish and is well off the scale.



(a) Operators in $plan3$ (b) Operators in $plan4$

Figure 2.13: Estimated and real output cardinalities of operators.

2.5.2 Cardinality Estimation

To evaluate the accuracy of the sampling-based cardinality estimation method, we compare the original and estimated output cardinalities of each operator in a given execution plan except the top operator and selection operators, which do not need estimation. The output cardinality of the top operator, k, is given by the query. The output cardinality of selection operator can be estimated by the estimated output cardinality of its input operator and its selectivity, that is often obtained from database statistics. For example, $plan3$ has 10 operators in total, among them the output cardinalities of 7 operators are estimated, since we do not estimate for the 2 selection operators and the root operator. Similarly $plan2$ and $plan4$ have the estimated cardinalities for 6 and 8 operators, respectively.

The experiment is based on a sample database with 0.1% sample ratio. Each of the original tables contains $100,000$ tuples and the join selectivity for the original tables is 0.0001. The number k is set to 10 (thus k' is 1). Figure 2.13 illustrates the estimation results of $plan3$ and $plan4$. The result of $plan2$ is very similar to that of $plan3$ therefore we do not include it. As we can see from the figure, although we used a very small sample, the real and estimated output cardinalities of the majority of the operators are in the same magnitude, validating the estimation method.

Chapter 3

Data Retrieval by Ranking: Ad-Hoc Ranking Aggregate Queries

In this chapter, we propose a principled framework for efficient processing of ad-hoc top-k aggregate queries. We define a cost metric on the number of "consumed" tuples, capturing our goal of producing only *necessary* tuples for generating top k groups. We identify the best-possible goal in terms of this metric that can be achieved by any algorithm, as well as the must-have information for achieving the goal. The key in realizing this goal is to find some good *order* of producing tuples (among many possible orders) that can guide the query engine toward processing the most promising groups first, and exploring a group only when necessary. We further discover that a *provably optimal* total schedule of tuples can be fully determined by two orders– the order of retrieving groups (*group ordering*) and the order of retrieving tuples (*tuple ordering*) within each group. Based on this view, we develop three fundamental principles and a new execution framework for processing top-k aggregate queries. We summarize our contributions as follows:

- **Principle for optimal aggregate processing:** We develop three properties, the *Upper-Bound*, *Group-Ranking* and *Tuple-Ranking Principles* that lead to the *exact-bounding*, *group-ordering* and *tuple-ordering requirements*, respectively. We formally show that the optimal aggregate query processing, with respect to our cost metric, can be derived by following these requirements.
- **Execution framework and implementations:** Guided by the principles, we propose a new execution framework, which enables query plans that are both *group-aware* and *rank-aware*. We further address the challenges of applying the principles and implementing the new query operators in this framework.

- **Experimental Study:** We implement the proposed techniques in POSTGRESQL. The experiments verify that our techniques can achieve orders of magnitude improvement in performance.

The rest of the chapter is organized as follows. We start in Section 3.1 by defining and motivating ranking aggregate queries. Section 3.2 analyzes the limitations of current techniques in processing such queries. We presents the fundamental principles in Section 3.3. In Section 3.4 we further address the challenges of applying the principles and implementing the new query operators in the execution framework. We experimentally evaluate the proposed framework in Section 3.5.

3.1 Query Model and Motivating Examples

Below is a SQL-like template for expressing top-k aggregate queries and an example query Q. While we use **limit**, various DBMSs use different SQL syntax to specify k.

select	ga_1, ..., ga_m, F
from	R_1, ..., R_h
where	c_1 **and** ... **and** c_l
group by	ga_1, ..., ga_m
order by	F
limit	k

$\mathcal{Q}$:

select	$A.g$, $B.g$, $C.g$, **sum**($A.v$ + $B.v$ + $C.v$) **as** $score$
from	A, B, C
where	$A.jc$ = $B.jc$ **and** $B.jc$ = $C.jc$
group by	$A.g$, $B.g$, $C.g$
order by	$score$
limit	k

That is, the groups are ordered by a *ranking aggregate* F=$G(T)$, where G is an *aggregate function* (*e.g.*, sum) over an *expression* T on the table columns (*e.g.*, $A.v$+$B.v$+$C.v$). The top k groups with the highest F values are returned as the query result. Formally, each group g=$\{t_1, \ldots, t_n\}$ has a *ranking score* $F[g]$, defined as

$$
\begin{aligned}
F[g] &= G(T)[g] = G(T[g]) = G(T[\{t_1, \ldots, t_n\}]) \\
&= G(\{T[t_1], \ldots, T[t_n]\}). \qquad (3.1)
\end{aligned}
$$

As the result, Q returns a sorted list $\mathcal{K}$ of k groups, ranked by their scores according to F, such that $F[g] \geq F[g']$, $\forall g \in \mathcal{K}$ and $\forall g' \notin \mathcal{K}$. When there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by the grouping attribute values of each group.

A distinguishing goal of our work is to support *ad-hoc* ranking aggregate criteria. With respect to G, we aim to support not only standard (*e.g.*, sum, avg), but also user-defined aggregate functions. With respect to the aggregated expression T, we allow T to be any expression, from simply a table column to very complex formulas. Below we show some motivating examples.

Example 6 (Motivating Queries):

$\mathcal{Q}1$:

select $zipcode$, **avg**($income$ * $w1$ + age * $w2$ + $credit$ * $w3$) **as** $score$
from Customer
where $occupation$ = 'student'
group by $zipcode$
order by $score$
limit 5

$\mathcal{Q}2$:

select $P.category$, $S.zipcode$, **mid_sum**($S.price$ - $P.manufacturer\_price$) **as** $score$
from Parts **as** P, Sales **as** S
where $P.part\_key$ = $S.part\_key$
group by $P.category$, $S.zipcode$
order by $score$
limit 5

The above query, $Q1$, returns the best 5 areas to advertise a student insurance product, according to the average customer score of each area. The score indicates how likely a customer will buy the insurance. A manager can explore various ways in computing the score, according to her analysis. For example, a weighted average of customer's income, age, and credit is used in $Q1$. Query $Q2$ finds the 5 best matches of part category and sales area that generate the highest profits. A pair

of category and area is evaluated by aggregating the profits of all sales records in that category and area. A user-defined aggregate function mid_sum is used to accommodate flexible metrics in such evaluation. For example, it can remove the top and bottom 5% (with respect to profit) sales records within each group and sum up the rest, to reduce the impact of outliers. ■

We emphasize that such ad-hoc aggregate queries often run in sessions, where users execute related queries with similar Boolean conditions but different ranking criteria, for exploratory and interactive data analysis. For example, in the above $Q1$, the manager can try various aggregate function and/or many combinations of the values of w_1, w_2, and w_3 until an appropriate ranking criterion is determined. Moreover, such related queries also exist across different sessions of decision support tasks over the same data.

We concentrate on a special but large class of aggregate queries $F = G(T)$, where the aggregate function G satisfies what we refer to as the *max-bounded* property: *An upper-bound of the aggregate F over a group g, denoted by $\overline{F}_g$, can be obtained by applying G to the maximum values of the member tuples in g.* The class of max-bounded functions include many typical aggregate functions such as sum, $weighted\ average$, *etc.*, as well as user-defined aggregate functions such as the mid_sum in query $Q2$ above. In fact, we believe that most ranking aggregate queries will use functions that satisfy this property.

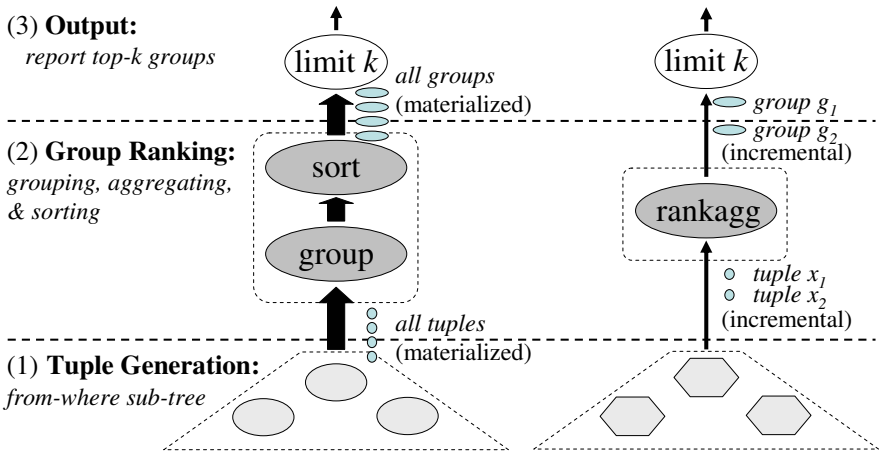
3.2 Limitations of Current Techniques

A popular conceptual data model for OLAP is *data cube* [40]. A data cube is derived from a *fact table* consisting of a *measure attribute* and a set of *dimensional attributes* that connect to the *dimension tables*. A cube consists of a lattice of cuboids, where each cuboid corresponds to the aggregate of the measure attribute according to a Group-By over a subset of the dimensional attributes. With respect to various measures and dimensions, multiple cubes may be necessary. As a conceptual model, data cube is seldom fully materialized given its huge size. Instead, in ROLAP servers, many materialized views (or *summary tables*) are selected to be built to cover certain tables

and attributes for answering aggregate queries [50]. Pervasive summary and index structures are further built upon the base tables and materialized views.

Many works studied the problem of answering aggregate queries using views [44, 90, 26, 1, 93]. They provide significant performance improvement when appropriate materialized views for the given query are available. However, they cannot answer ad-hoc ranking aggregate queries. Materialized views only maintain information of the *pre-determined* attribute or expression using the *prescribed* aggregate function. In contrast, ad-hoc ranking conditions are determined or defined *on-the-fly* during decision making. Therefore in order to answer a ranking aggregate F=$G(T)$, G must be the aggregate function used when the cubes (views) are materialized or can be derived from the materialized aggregate function (*e.g.*, avg derived from sum and $count$), and T must happen to be simply some measure attribute or expression that can be derived from the summary tables, instead of arbitrarily complex expression. Given the virtually infinite choices of G and T in ad-hoc data analysis, the pre-computed information easily become irrelevant when the query is different from what the summary tables are built for.

When pre-computed cubes cannot answer the query, the processing has to fall back to base tables, where the query is evaluated by the relational query engine of the ROLAP server as follows: (1) fully consume all the input tables; (2) fully materialize the selection and join results; (3) group the results by grouping attributes and compute the aggregates for every group; (4) fully sort the groups by their ranking aggregates; and (5) report only the top k groups. The user is only interested in the k top groups instead of a total order on all groups. The traditional processing strategy is thus an overkill, with unnecessary overhead from full scanning, joining, grouping, and sorting. Given the large amount of data in a warehousing environment, such a naïve *materialize-group-sort* scheme can be unacceptably inefficient. Moreover, the users may have to go through it many times in their exploratory and interactive analysis tasks. Such inefficiency thus significantly impacts the usefulness of decision support applications, resulting in low productivity.



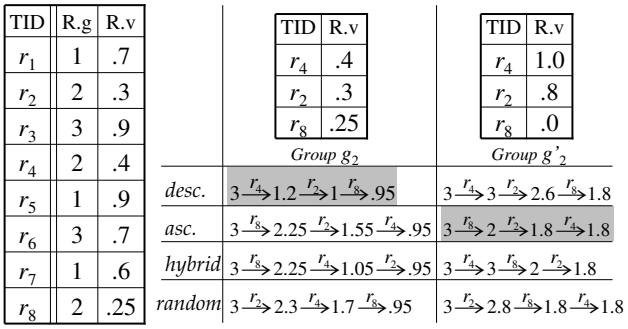
(a) Traditional query plan. (b) New query plan.

Figure 3.1: Top-k aggregates query processing.

3.3 Principles: Optimal Aggregate Processing

Efficient support of ranking aggregate queries is critically lacking in current systems. To motivate, Figure 3.1(a) illustrates the traditional materialize-group-sort query plan, consisting of three components: 1) *tuple generation*: the from-where subtree for producing the member tuples of every group; 2) *group ranking*: the *group* and *sort* operators for generating the groups and ranking them; and 3) *output*: the *limit* operator returning the top-k groups. As Section 3.2 discussed, this approach fully materializes and aggregates all tuples, and then fully materializes and sorts all groups. Since only top-k groups are requested, much of the effort is simply wasted.

Our goal is thus to design a new execution model, as Figure 3.1(b) contrasts. We need a new non-blocking *rankagg* operator, which *incrementally* draws tuples, as its input from the underlying subtree, and generates top groups in the ranking order, as its output. For efficiency, *rankagg* must minimize its consumption of input tuples: Although in practice the cost formula can be quite complex with many parameters, this input cardinality (*i.e.*, number of tuples consumed) is always an essential factor. As our metric, for a group g with n tuples $\{t_1, \ldots, t_n\}$, how "deep" into the group shall we process, for determining the top-k groups? We refer to this number of tuples



TID	R.g	R.v
r_1	1	.7
r_2	2	.3
r_3	3	.9
r_4	2	.4
r_5	1	.9
r_6	3	.7
r_7	1	.6
r_8	2	.25

(a) Relation R.

	Group g_2	Group g'_2
	TID R.v: r_4 .4, r_2 .3, r_8 .25	TID R.v: r_4 1.0, r_2 .8, r_8 .0
desc.	$3 \xrightarrow{r_4} 1.2 \xrightarrow{r_2} 1 \xrightarrow{r_8} .95$	$3 \xrightarrow{r_4} 3 \xrightarrow{r_2} 2.6 \xrightarrow{r_8} 1.8$
asc.	$3 \xrightarrow{r_8} 2.25 \xrightarrow{r_2} 1.55 \xrightarrow{r_4} .95$	$3 \xrightarrow{r_8} 2 \xrightarrow{r_2} 1.8 \xrightarrow{r_4} 1.8$
hybrid	$3 \xrightarrow{r_8} 2.25 \xrightarrow{r_4} 1.05 \xrightarrow{r_2} .95$	$3 \xrightarrow{r_4} 3 \xrightarrow{r_8} 2 \xrightarrow{r_2} 1.8$
random	$3 \xrightarrow{r_2} 2.3 \xrightarrow{r_4} 1.7 \xrightarrow{r_8} .95$	$3 \xrightarrow{r_2} 2.8 \xrightarrow{r_8} 1.8 \xrightarrow{r_4} 1.8$

(b) Example tuple orders for *Group* g_2.

Figure 3.2: Relation R and some tuple orders.

consumed for g as its *tuple depth*, denoted H_g. Our goal is thus to minimize the total cost of all groups, *i.e.*, $\Sigma_g H_g$.

As the foundation of our work, while the new *rankagg* can be implemented in different ways, what are the requirements and guidelines for *any* such algorithm? To minimize tuple consumption (*i.e.*, to stop processing and to prune the groups early), what information must we have and what is the criterion in such pruning? As tuples flow from the underlying subtree, in what *order* shall *rankagg* request and process tuples? Is there an optimal *tuple schedule* that minimizes the total tuple depths? We develop three fundamental principles for determining *provably optimal* tuple schedule (Theorem 1) that achieves the theoretical minimal tuple consumption: the Upper-Bound Principle for early pruning, the Group-Ranking Principle for asserting "inter-group" ordering, and the Tuple-Ranking Principle for further deciding "intra-group" ordering. These principles guide our implementation of *rankagg* and determine its impact to the underlying query tree. (The subtrees for tuple generation in Figure 3.1(a) and (b) use different shapes to emphasize that *rankagg* requires modifying the underlying operators such as scan and join.)

The following query is our running example. The input relation R, with two attributes $R.g$ and $R.v$, is shown in Figure 3.2(a). The query groups by $R.g$, and our following discussion refers to those $R.g$=1, 2, and 3 as group g_1, g_2, and g_3, respectively. The query specifies a ranking aggregate function F= $G(T)$=$sum(R.v)$ and k=1. Throughout this work, we assume T is in the range of $[0, 1]$.

select $R.g$, **sum**($R.v$) **from** R

group by $R.g$ **order by sum**($R.v$) **limit** 1

3.3.1 Upper-Bound Principle

Our first principle deals with the requirements of early pruning: what information *must* we have in order to prune? During processing, before a group g is fully evaluated, the obtained tuples of g can effectively bound its ultimate aggregate score. For a ranking aggregate F= $G(T)$, we define $\overline{F}_{\mathcal{I}_g}[g]$, the *maximal possible score* of g, with respect to a set $\mathcal{I}_g$ of obtained tuples ($\mathcal{I}_g \subseteq g$), as the upper-bound score that g may eventually achieve, *i.e.*, $F[g] \leq \overline{F}_{\mathcal{I}_g}[g]$.

The upper-bound score of a group thus indicates the best the group can achieve. For our discussion, call the lowest top-k score of the query as the *threshold*, denoted θ. (For instance, θ=$F[g_1]$=2.2 in our running example.) Note that θ would not be known before the processing ends. Given a group g, if its upper-bound score is higher than or equal to θ, it has a chance to make into the top k groups. To conclude that g does not belong to the top k and thus prune it from further processing, the upper-bound score of g must be below θ, otherwise we may incorrectly prune g that indeed belongs to the top k. Therefore the upper-bound score decides the minimal number of tuples that any algorithm (that processes by obtaining tuples) must obtain from g before it can prune g. As stated in the following Property 2, this minimal tuple depth is the *best-possible goal* of any algorithm, due to that pruning a group with less obtained tuples can result in wrong query answers.

For the properties and theorems in this chapter, the proofs are provided in the appendix of the thesis.

Property 2 (Best-Possible Goal): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^{min} be its minimal tuple depth, *i.e.*, the number of tuples to retrieve from g before it can be pruned from further processing, or otherwise determined

to be in the top-k groups. The H_g^{min} is the smallest number of tuples from g that makes the maximal possible score of g to be below θ, *i.e.*,

$$H_g^{min} = \min\{|\mathcal{I}_g| \overline{F}_{\mathcal{I}_g}[g] < \theta, \mathcal{I}_g \subseteq g\}, \tag{3.2}$$

or otherwise H_g^{min}=$|g|$ if such a depth does not exist. ■

We emphasize that, as Property 2 implies, an algorithm must have certain information about upper-bound score for pruning. Various algorithms may exploit various ways in computing $\overline{F}_{\mathcal{I}_g}[g]$, resulting in different pruning power, *i.e.*, different H_g^{min}. For an algorithm that has no knowledge to derive a non-trivial upper-bound, $\overline{F}_{\mathcal{I}_g}[g]$ would be in its most trivial form, that is, infinity. Such a trivial upper-bound cannot realize any pruning at all, since the upper-bound score would never go below θ.

Property 3 (Must-Have Information): For any group g, with a trivial upper-bound $\overline{F}_{\mathcal{I}_g}[g]$= $+\infty$ under every $\mathcal{I}_g$, H_g^{min}=$|g|$. ■

Therefore we must look for a non-trivial definition of $\overline{F}_{\mathcal{I}_g}[g]$ in order to prune. Since we focus on aggregate functions G that are max-bounded (which describes a wide class of functions, as Section 3.1 defined), the maximal-possible score can be obtained by Eq. 3.3 below, which simply substitutes unknown tuple scores with their maximal value of T, denoted by $\overline{T}_{\mathcal{I}_g}$.

$$\overline{F}_{\mathcal{I}_g}[g] = G\left(\{T_i| \begin{array}{ll} T_i = T[t_i] & \text{if } t_i \in \mathcal{I}_g \text{ (\textit{seen tuples})}; \\ T_i = \overline{T}_{\mathcal{I}_g} & \text{otherwise (\textit{unseen tuples}).} \end{array} \forall t_i \in g\}\right). \tag{3.3}$$

Example 7 (Maximal-Possible Scores): Consider our running example F=$G(T)$=$sum(R.v)$. Suppose the *rankagg* operator has processed tuples r_1 and r_2; *i.e.*, $\mathcal{I}_{g_1}$=$\{r_1\}$, $\mathcal{I}_{g_2}$=$\{r_2\}$, and $\mathcal{I}_{g_3}$=ϕ. Suppose the exact size of every group is known a priori. As g_1 has seen only r_1 with $T[r_1]$=.7 and the two unseen tuples (its count is 3) can score as high as 1.0, $F[g_1]$ can aggregate to $F[g_1]$≤sum(.7+

$1.0\times2)$=2.7, or $\overline{F}_{\mathcal{I}_{g_1}}[g_1]$=2.7. Similarly, $\overline{F}_{\mathcal{I}_{g_2}}[g_2]$=$sum$(.3+1.0 $\times2)$=2.3; $\overline{F}_{\mathcal{I}_{g_3}}[g_3]$=$sum(1.0\times2)$=2.0. ■

Note that Eq. 3.3 requires to know $\overline{T}_{\mathcal{I}_g}$ and the count (size) of a group, or at least an upper-bound of this count, to constraint the number of unknown tuples. (For example, if 4 is used as the upper-bound of g_1's size in Example 7, $F[g_1]\leq sum(.7+1.0\times3)=3.7$.) We refer to these values as the "*grouping bounds*", consisting of *tuple count* ($\overline{|g|}$, the upper-bound of g's size) and *tuple max* ($\overline{T}_{\mathcal{I}_g}$).

Eq. 3.3 captures a class of definitions of maximal-possible score, as different ways can be explored in getting the grouping bounds, resulting in different $\overline{F}_{\mathcal{I}_g}[g]$ and thus different H_g^{min}. For instance, using infinity to bound the tuple count or the tuple max results in $\overline{F}_{\mathcal{I}_g}[g]$ as infinity, with no pruning power. Given Eq. 3.3, for any group g, the smaller $\overline{|g|}$ and $\overline{T}_{\mathcal{I}_g}$, the smaller $\overline{F}_{\mathcal{I}_g}[g]$. Therefore the most pruning power, *i.e.*, the smallest H_g^{min}, is realized by the exact group size and the exact highest T, as stated below.

Requirement 1 (Exact Bounding): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. Given the definition of $\overline{F}_{\mathcal{I}_g}[g]$ in Eq. 3.3, to obtain the smallest H_g^{min}, we must use $\overline{|g|}$=$|g|$ and $\overline{T}_{\mathcal{I}_g}$=max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$. ■

Based on Requirement 1, our implementation choice of grouping bounds is the exact count of a group as $\overline{|g|}$ and a value very close to max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$ as $\overline{T}_{\mathcal{I}_g}$. We justify this choice and show how to obtain such grouping bounds in Section 3.4.1. Note that our discussion of the following principles is orthogonal to the choices of grouping bounds, which only result in different best-possible tuple depth H_g^{min} that our algorithm sets to achieve.

3.3.2 Group-Ranking Principle

Property 2 gives the minimal tuple depth H_g^{min} for each g, thus the minimal total cost $\Sigma_g H_g^{min}$. The essence of Eq. 3.2 lies in that we should stop processing a group as soon as it can be excluded

from top-k answers. That is, we should only further process a group if it is *proven* to be absolutely necessary, *i.e.*, its upper-bound score is above the threshold θ. While Eq. 3.2 hints on such "necessity", it does not suggest how to determine the necessity, because θ can only be known at the conclusion of a query. Therefore we wonder, as an algorithm retrieves tuples one by one, is there an optimal tuple schedule that achieves the minimum depth?

A schedule is determined by *inter-group* and *intra-group* ordering. Our Group-Ranking principle asserts inter-group ordering: When selecting the next tuple t to process, how to order *between* groups? Which group should t be selected from? (While this work defines such insight of "branch-and-bounding" for aggregate queries for the first time, similar intuition has also been applied to ordering *individual* tuples [17, 12, 67] in top-k queries.) Thus the *Group-Ranking Principle* builds upon the basis that groups with higher bounds must be processed further before others.

Such bounds guide our selection of the next tuple. Let's illustrate with Example 7: The next tuple *should* be selected from g_1. Consider g_1 vs. g_2 (and similarly g_3). If g_1 will be the top-1, we must complete its score. Otherwise, since $\overline{F}_{\mathcal{I}_{g_1}}[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, whatever score g_2 can achieve, g_1 can possibly do better. Thus, first, although g_2 is incomplete, it may not be *necessary* for further processing, since g_1 may turn out to be the answer (*i.e.*, g_1 should be processed before g_2). Second, even if g_2 were complete, it is not *sufficient* to declare g_2 as the top-1, since g_1 may be a better answer. In all cases, we must process the next tuple from g_1.

The above explanation intuitively motivates the priority between g_1 and g_2, for the special case when k=1. The Group-Ranking Principle formally states this property, for general top-k ($k \geq 1$) situations, which mandates the priority of current top k groups (*i.e.*, g_1) over others (*i.e.*, g_2).

Property 4 (Group-Ranking Principle): Let g_1 be any group in the current top-k ranked by maximal-possible scores $\overline{F}$ and g_2 be any group not in the current top-k. We have 1) g_1 must be further processed if g_1 is not fully evaluated, 2) it may not be necessary to further process g_2 even if g_2 is not fully evaluated, and 3) the current top-k are the answers if they are all fully evaluated. ■

The Group-Ranking Principle guides our inter-group ordering for query processing, by prioritizing on $\overline{F}_{:}$ Essentially, the principle states that, to avoid unnecessary tuple evaluations, our algorithms must prioritize any incomplete g_1 within the current top-k over those g_2 outside. Thus, *first*, as the *progressive* condition, to reach the final top-k, any such g_1 must be further processed (or else there are no enough k complete groups to conclude as better than g_1). *Second*, as the *stop* condition, when and only when no such g_1 exists, *i.e.*, all top-k groups are completed, we can conclude these groups as the final answers. Below we summarize this requirement.

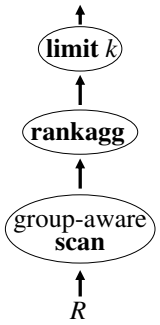
Requirement 2 (Group Ordering): To avoid the unnecessary tuple consumption, query processing should prioritize groups by their maximal-possible score $\overline{F}_{:}$

- (**Progressive Condition**) If there are some incomplete groups g_1 in the top-k, then the next tuple should be selected from such g_1;
- (**Stop Condition**) Otherwise, we can stop and conclude the current top-k groups as the final answers. ■

Example 8 (Sample Execution 1): For our example F= $G(T)$ = $sum(R.v)$, to find the top-1 group, Figure 3.3(b) conceptually executes Requirement 2. (We discuss the corresponding Figure 3.3(a) in Section 3.4.) We prioritize groups by $\overline{F}$ scores, initially (3.0, 3.0, 2.0), when no tuples in any group g are seen ($\mathcal{I}_g$=ϕ) and thus $\overline{T}_{\mathcal{I}_g}$=1.0 in Equation 3.3. As the *Progressive Condition* dictates, we always choose the top-1 group (marked *) for the next tuple, thus accessing r_1 from g_1, r_2 from g_2, ..., and finally r_4 from g_2. Now, since the top-1 group g_1 is completed (with final score $F[g_1]$=$\overline{F}[g_1]$=2.2), the *Stop Condition* asserts no more processing necessary, and thus we return g_1 as the top-1. ■

3.3.3 Tuple-Ranking Principle

Our last principle addresses the *intra-group* order: When we must necessarily process group g (as the Group-Ranking Principle dictates), which tuple in g should we select? This tuple ordering,



(a) Query plan.

step	*action*	$\overline{F}[g_1]$	$\overline{F}[g_2]$	$\overline{F}[g_3]$
0.	initial	3.0*	3.0	2.0
1.	(r_1, 1, .7)	2.7	3.0*	2.0
2.	(r_2, 2, .3)	2.7*	2.3	2.0
3.	(r_5, 1, .9)	2.6*	2.3	2.0
4.	(r_7, 1, .6)	**2.2**	2.3*	2.0
5.	(r_4, 2, .4)	**2.2***	1.7	2.0

S*: *Top-1 score* **S**: *Final score*

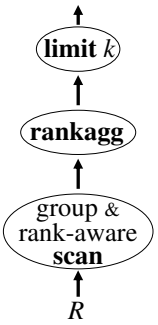
(b) Tuple flow to **rankagg**.

Figure 3.3: Query execution 1: GroupOnly.

together with the group ordering just discussed, will determine a total schedule of tuple access for the *rankagg* operator (Figure 3.1(b)).

To start with, we note that different tuple orders result in different cost efficiency, in terms of tuple depth of each group. Given a tuple order α for group g, what would be the resulting tuple depth H_g^α that must be accessed? Recall in Example 8 we order tuples arbitrarily by tuple IDs (see relation R in Figure 3.2(a)), *i.e.*, group g_1 as x_1:$r_1 \rightarrow r_5 \rightarrow r_7$, g_2 as x_2:$r_2 \rightarrow r_4 \rightarrow r_8$, and g_3 as x_3:$r_3 \rightarrow r_6$. These orders result in depths $H_{g_1}^{x_1}$=3 (*i.e.*, all of r_1, r_5, r_7 accessed), $H_{g_2}^{x_2}$=2, $H_{g_3}^{x_3}$=0, as Figure 3.3(b) shows. To contrast, Example 9 below shows how different tuple orders result in different depths.

Example 9 (Sample Execution 2): Rerun Example 8 but with tuple orders as sorted by tuple scores T=$R.v$ in each group, thus ordering g_1 as d_1:$r_5 \rightarrow r_1 \rightarrow r_7$, g_2 as d_2:$r_4 \rightarrow r_2 \rightarrow r_8$, and g_3 as d_3:$r_3 \rightarrow r_6$. These descending orders, together with Requirement 2, result in the execution of Figure 3.4(b). (Again, Figure 3.4(a) is discussed in Section 3.4.) Note that, for each group, the *descending* order sorted by T effectively bounds the T-score of unseen tuples by the *last-seen* T-score. Thus, for group g_1, after r_5 at step 1 with $T[r_5]$=$r_5.v$=.9, $\overline{F}[g_1]$=0.9+0.9×2 (for 2 unseen tuples)=2.7. Then, after r_1 in step 3, $\overline{F}[g_1]$=0.9+0.7+0.7×1 (for 1 unseen tuple) = 2.3. In this execution, each group is accessed to depth $H_{g_1}^{d_1}$=3, $H_{g_2}^{d_2}$=1, and $H_{g_3}^{d_3}$=0. In particular, group g_2 has a depth $H_{g_2}^{x_2}$ = 2 and $H_{g_2}^{d_2}$ = 1 (out of 3), and thus d_2 is a better order than x_2. ■



(a) Query plan.

step	*action*	$\overline{F}[g_1]$	$\overline{F}[g_2]$	$\overline{F}[g_3]$
0.	initial	3.0*	3.0	2.0
1.	(r_5, 1, .9)	2.7	3.0*	2.0
2.	(r_4, 2, .4)	2.7*	1.2	2.0
3.	(r_1, 1, .7)	2.3*	1.2	2.0
4.	(r_7, 1, .6)	**2.2***	1.2	2.0

S*: *Top-1 score* **S**: *Final score*

(b) Tuple flow to **rankagg**.

Figure 3.4: Query execution 2: GroupRank.

To minimize the total costs (as the sum of tuple depths), how do we find the optimal order α for each group g such that H_g^α=H_g^{min}? Apparently, this "space" of orders seems prohibitively large: If there are n tuples in each of the m groups, as each group has $n!$ permutations, there will be $(n!)^m$ different orders. Thus, our Property 5, or the *Tuple-Ranking Principle*, addresses this tuple ordering issue. It has two main results:

First, *order independence*: To find the optimal orders, shall we consider the *combinations* among the orders of *different* groups? It turns out that, *if* we follow Requirement 2 for group ordering, the optimal tuple order of each group is independent of all others. That is, the tuple depth of a group g depends on only its *own* order α.

To see why, let's consider g_2 in Figure 3.3 and 3.4. As Requirement 2 dictates, by the Progressive Condition, we only necessarily access a next tuple from the group, *when* and *only when* $\overline{F}[g_2]$ remains in the top-k (k=1 in this case). The execution halts, as the Stop Condition asserts, when the top-k groups are completed and "surfaced" to the top, at which point $\overline{F}[g_2]<\theta$ and thus no longer needs further processing. Thus, in Figure 3.3, with tuple order x_2:$r_2 \rightarrow r_4 \rightarrow r_8$, $\overline{F}[g_2]$ progressively lowers its upper bounds as $3.0 \xrightarrow{r_2} 2.3 \xrightarrow{r_4} 1.7$, at which point it stops, because $1.7<2.2$, or $1.7<\theta$. To contrast, in Figure 3.4, the tuple order d_2 results in $3.0 \xrightarrow{r_4} 1.2$, where it stops as $1.2<\theta$. While the different orders result in different depths $H_{g_2}^{x_2}$=2 and $H_{g_2}^{d_2}$=1, both are the "smallest" depths (under the respective orders) that make $\overline{F}[g_2]$ go below θ– which are dependent on only the tuple order of g_2 and independent of others.

Second, T*-based Ranking*: While groups are independent, for each group, what orders α as $t_1 \rightarrow \cdots \rightarrow t_n$, out of the $n!$ permutations (for a group of n tuples), should we consider? As just explained above, a better order (*e.g.*, d_2 vs. x_2) of g will decrease the upper bounds $\overline{F}[g]$ more rapidly to go below θ with less tuple accesses. What orders can achieve such rapid decreasing?

As the upper bounds $\overline{F}$ are defined by Eq. 3.3, the answer naturally lies there. There are two components in the equation: 1) the scores $T[t_i]$ of the *seen* tuples t_i in $\mathcal{I}_g$, and 2) the upper bound $\overline{T}_{\mathcal{I}_g}$ of the *unseen* tuples. Intuitively, *first*, a good order can lower the seen scores, by accessing tuples with the smallest T. *Second*, it can also lower the upper bounds of those unseen, by retrieving tuples from high T to low, where the unseen are bounded by the last-seen tuple (as in Example 9). Following this intuition, we only need to consider T*-desc/asc*, a class of orders that always choose either the highest or the lowest from the unseen tuples as the next. That is, any other order must be inferior to some order in this class.

Property 5 formalizes the two results.

Property 5 (Tuple-Ranking Principle): With respect to a ranking aggregate F=$G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^α be the tuple depth with respect to tuple order α:$t_1 \rightarrow \cdots \rightarrow t_n$, when the inter-group ordering follows Requirement 2.

- (**Order Independence**) The depth H_g^α depends on only α (the order of this group) and θ (the global threshold), and not on the order of other groups. Specifically, H_g^α is the smallest depth l of sequence α that makes the maximal possible score of g to be below θ, *i.e.*,

$$H_g^\alpha = \min_{l \in [1:n]} \{l | \overline{F}_{\{t_1,\ldots,t_l\}}[g] < \theta\}, \tag{3.4}$$

  or otherwise H_g^α=n if such a depth does not exist.
- (T**-based Ranking**) To find the optimal order α that results in the minimum H_g^α, *i.e.*, H_g^α=H_g^{min},

we only need to consider the class of orders T-*desc/asc* =

$$\{\alpha : t_1 \rightarrow \cdots \rightarrow t_n | \begin{array}{ll} \text{either} & T[t_i] \geq T[t_j] \forall j > i \text{ (from top);} \\ \text{or} & T[t_i] \leq T[t_j] \forall j > i \text{ (from bottom).} \end{array} \forall t_i\}. \quad (3.5)$$

■

To conclude, we summarize the implementation implications of the Tuple-Ranking Principle as Requirement 3, which guides our design of a processing model for the optimal tuple ordering.

Requirement 3 (Tuple Ordering): If Requirement 2 is followed, to minimize the total tuple depths across all groups: 1) the order of each group can be optimized independently; and 2) the optimal order is one from T-*desc/asc*, that results in the minimum H_g^α as governed by Eq. 3.4. ■

3.3.4 Putting Together: Overall Optimality

Together, the Upper-Bound Principle dictates the best-possible goal and the must-have information (the maximal-possible score) in early pruning for any algorithm; based on the maximal-possible score, the Group-Ranking and Tuple-Ranking Principles guide the tuple scheduling for our *rank-agg* operator to selectively draw from the underlying query tree (Figure 3.1(b)). We stress that, as the following Theorem 1 states, the Group-Ranking and Tuple-Ranking Principles enable the finding of an *optimal* tuple schedule which processes every group minimally, thus achieving an overall minimum tuple depth $\Sigma_g H_g^{min}$ (*i.e.*, the best-possible goal) with respect to some upper-bound mechanism $\overline{F}_{\mathcal{I}_g}[g]$. We note that Requirement 2 determines an inter-group order that only accesses a group when *necessary*, and Requirement 3 further leads to a "cost-based" optimal intra-group order for each group, with a significantly reduced space of only T-desc/asc orders.

Theorem 1 (Optimal Aggregate Processing): If query processing follows Requirements 2 and 3, the number of tuples processed across all groups, *i.e.*, $\Sigma_g H_g$, is the minimum possible for query answering, *i.e.*, $\Sigma_g H_g^{min}$. ■

3.4 Execution Framework and Implementations

The principles developed in Section 3.3 provide a guideline in realizing the new model of execution plans in Figure 3.1(b). In this section, we propose an execution framework for applying the principles (Section 3.4.1). We address the challenges in implementing the new *rankagg* operator (Section 3.4.2) and discuss its impacts to the existing operators (Section 3.4.3).

3.4.1 The Execution Framework

We design a framework to apply the principles. The framework consists of two orthogonal components. The first component provides the grouping bounds, which define the maximal possible score $\overline{F}_{\mathcal{I}_g}[g]$, the must-have information according to the Upper-Bound Principle. The second component schedules tuple processing based on the grouping bounds by exploiting the Group-Ranking and Tuple-Ranking Principles. The two components are orthogonal because the Grouping-Ranking and Tuple-Ranking principles are applicable to any grouping bounds, from which the only impact is that different bounds result in different best-possible tuple depth H_g^{min} that can be achieved by tuple scheduling. In this section we first give a detailed discussion on how to obtain the grouping bounds. We then present how to implement the Tuple-Ranking Principle, how to implement the Group-Ranking Principle, and how to enable new group-aware and rank-aware query plans that apply the principles. Finally, we discuss variations of the querying plans that are applicable under various situations.

Obtaining Grouping Bounds: Exploiting Upper-Bound Principle

Based on Requirement 1, the smallest H_g^{min} with respect to Eq. 3.3 is obtained by the tightest grouping bounds, *i.e.*, the exact tuple count and the highest unseen tuple value. In our framework we aim at using this tightest bounds for maximal pruning.

With respect to the tuple max, the tightest bound $\overline{T}_{\mathcal{I}_g}$=max$\{T[t_i] \mid t_i \in g - \mathcal{I}_g\}$ is impossible to obtain though. The reason is simply that, without actually seeing the unseen tuples, we cannot

know the exact highest value among them. However, we can obtain a value that is very close to it. For instance, if the tuples in g are retrieved in T-*desc/asc* order, the T value of the last seen tuple from the top end bounds the value of the unseen tuples. Before any member tuple of g is retrieved, $\overline{T}_{\mathcal{I}_g}$ has an initial value $\overline{T}_g$, which is the maximum-possible value of T among all the tuples in g. A tight $\overline{T}_g$ can be obtained either by application semantic (*e.g.*, according to the definition of T), or by the indices that are pervasively built upon base tables and materialized views in OLAP environment. It can be either global (*e.g.*, using the overall highest T value according to the index), or group-specific (*e.g.*, using multi-key index over the grouping attribute and the attributes involved in T.)

With respect to the tuple count, the tightest bound (*i.e.*, the exact size of a group), $\overline{|g|}$=$|g|$, provides the most pruning power. Looser bounds can be also obtained. For example, we may use the size of a base table to bound the size of any base table group, and the product of base table group sizes to bound the joined group size (by assuming full join, *i.e.*, Cartesian product). However, such upper-bounds are very loose and are unlikely to realize early pruning. We note that any efficient method to compute a tight upper-bound of the count can be plugged into our framework as another choice of the tuple count. Below we discuss how to obtain the exact tuple count $\overline{|g|}$=$|g|$. There are three situations:

- **Counts ready:** In decision support, although the ranking aggregate function $G(T)$ can be very ad-hoc, the join and grouping conditions are largely shared across many related queries, as motivated in Section 3.1. In such an environment, materialized views are built based on the query workload to cover frequently asked query conditions. As a very basic aggregate function in OLAP, the count of each group is thus often ready through the materialized views, *e.g.*, in data cube.

- **Counts computed from materialized information:** In certain cases, the counts are not directly ready, but can be efficiently obtained by querying the materialized views [44, 90, 26, 1, 93]. For example, for a top-k aggregate query with selection conditions involving some dimensional attributes (*e.g.*, May$\leq month\leq$June), a group (*e.g.*, $city$='Chicago') corresponds to the aggregate of multiple underlying groups (*e.g.*, ($city$='Chicago', $month$=May) and ($city$='Chicago',

$month$=June)). Its size can thus be obtained by aggregating upon the materialized views (*e.g.*, the view containing the count of each ($city$, $month$) group).

- **Counts computed from scratch:** When counts cannot be directly or indirectly obtained, we have to compute it from scratch. That is, we replace the ranking function F= $G(T)$ by $count(*)$ and remove the **order by** and **limit** clauses. The resulting query (let's call it *count query*) is executed by any traditional approach to obtain the counts. For instance, the count query corresponding to our running example is

  select $R.g$, **count**($*$) **from** R **group by** $R.g$

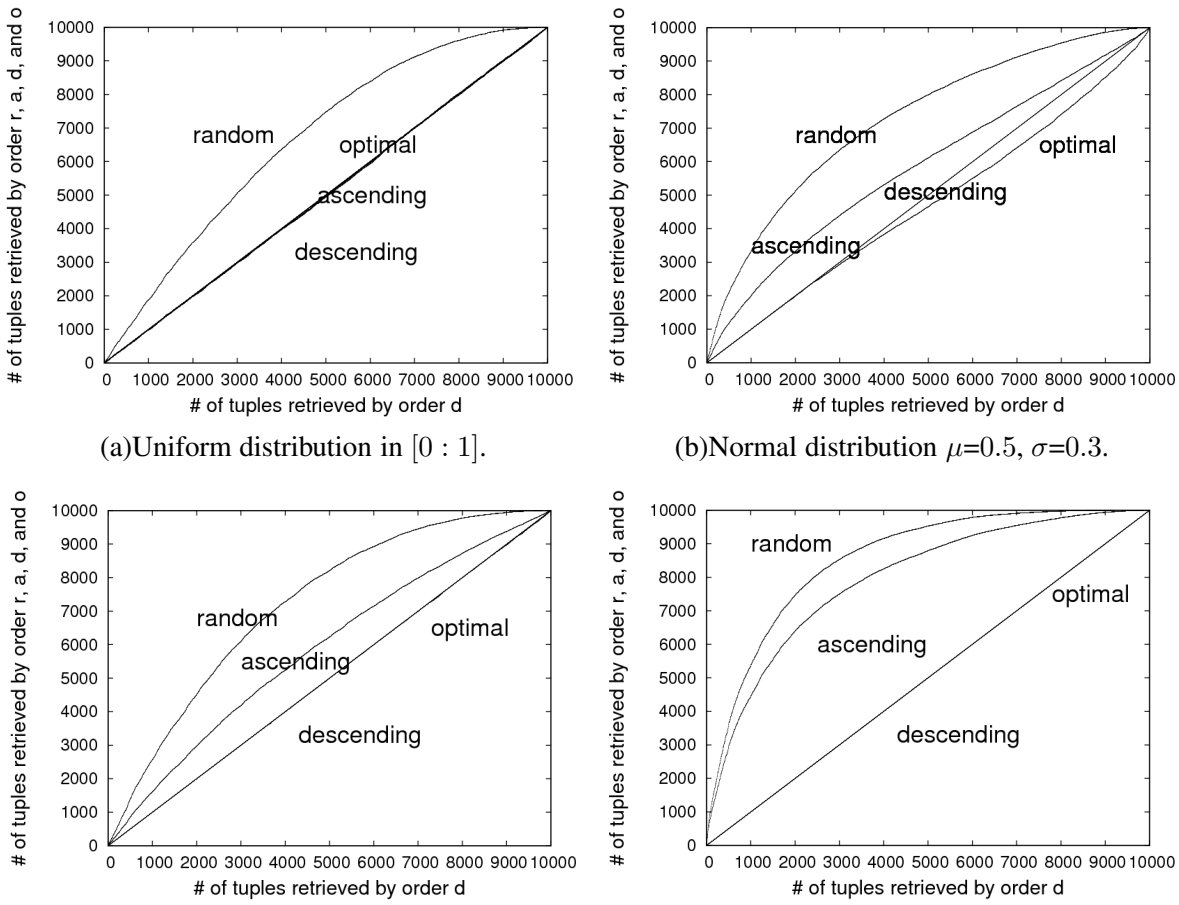
In Section 3.5, our experimental results show that our approach is orders of magnitude more efficient than the materialize-group-sort approach when counts are available. When we have to compute the counts from scratch (or similarly from materialized views), the cost of the first single query is comparable to that of materialize-group-sort. More importantly, the resulting counts can be materialized and maintained to benefit many subsequent related ad-hoc queries, thus the cost of computing the counts is amortized.

T-descending Heuristic: Implementing Tuple-Ranking Principle

As Requirement 3 states, we only need to consider the class of T-*desc/asc* orders for finding an optimal tuple order of a group. Such orders retrieve the tuples from only the top and bottom ends with respect to the value of T, thus exploit T-based ranking to either reduce the seen scores or the upper bound of those unseen. Based on the intuition of retrieving tuples from two ends, we thus consider two simple heuristics of choosing intra-group order, each of which produces a representative case of T-*desc/asc*, for α: $t_1 \rightarrow \cdots \rightarrow t_n$.

1. T-*descending*: Always choose the tuple with the *highest* T-score as the next, *i.e.*, $T[t_1] \geq \cdots \geq T[t_n]$.

2. T-*ascending*: Always choose the tuple with the *lowest* T-score as the next, *i.e.*, $T[t_1] \leq \cdots \leq T[t_n]$.



(a)Uniform distribution in $[0:1]$.

(b)Normal distribution μ=0.5, σ=0.3.

(c)Exponential distribution λ=1.

(d)Gamma distribution α=0.5, β=0.5.

Figure 3.5: Number of tuples retrieved by random(r), ascending (a), descending (d), and optimal (o) order to get the same or lower upper-bound as descending order.

Example 10 (Tuple Orders): Consider group g_2 in our example. Figure 3.2(b) shows four example orders: *descending*, *ascending*, *hybrid*, and *random*. The *descending* and *ascending* are special cases of T*-desc/asc*, *hybrid* is another instance in T*-desc/asc*, and *random* is an order that does not belong to T*-desc/asc*. For each order, the figure shows how $\overline{F}[g_2]$ changes in sequence, *e.g.*, *descending* decreases $\overline{F}$ as 3, 1.2, 1, .95 (the final score). By comparison, *descending* is the best order, which lowers $\overline{F}$ most rapidly. ■

We choose T*-descending* as our implementation heuristic. We show that T*-descending* in practice is often the best choice for typical score distributions (*e.g.*, *uniform* and *normal*) and aggregate functions (*e.g.*, *sum* and *avg*). In Figure 3.5 we empirically compare T*-ascending* (a),

T-*descending* (d), the random order (r), and the optimal order (o) which results in H_g^{min}. The tuple scores $T[t_i]$ within a group g of n=$10,000$ tuples are generated by various distributions, in the range of $[0, 1]$. The aggregate function is *sum*. (The results for *avg* are similar.) Suppose the maximal-possible score is f after $|\mathcal{I}_g^d|$ tuples are retrieved by d. Ranging $|\mathcal{I}_g^d|$ from 1 to n (x-axis), we compare $|\mathcal{I}_g^a|$, $|\mathcal{I}_g^d|$, $|\mathcal{I}_g^r|$, and $|\mathcal{I}_g^o|$(y-axis), the number of retrieved tuples by a, d, r, and o, respectively, to get their maximal-possible scores lower than or equal to f. The curve for T-*descending* is the diagonal since it is the reference order. The figure shows that (1) T-*descending* in most cases overlaps with the optimal order, justifying our implementation heuristic; and (2) the random order is always worse than others, verifying that simply choosing any order is not appropriate.

There are data distributions where T-*descending* is worse than other orders. For instance, we change g_2 to g_2' in Figure 3.2(b), which shows four example orders for both g_2 and g_2'. Now, T-*ascending* is the best order, by getting low scores from the bottom(*i.e.*, $r_8.v$=0). In general, in a dataset, if many tuples are in the high score end, T-*descending* at the beginning cannot effectively lower the upper-bound of unseen tuples, resulting in low efficiency.

Note that more sophisticated heuristic may be applied in determining intra-group tuple order. For instance, a heuristic can indeed retrieve both the high and low score ends, by determining to retrieve from top or bottom based on the distribution of seen data. Such heuristic would require complex implementation and bring more overheads, for ranking on both ends and the analysis of data distribution. More seriously, such greedy algorithm based on the seen data may led to local optimum. For instance, if there are several tuples with the same score clustered at the high score end, such heuristic may determine that retrieving tuples from the top end cannot reduce the upper-bound of the unseen tuples, thus will only retrieve from the bottom end. However, it may turn out that tuple scores decrease rapidly after those with the same score, thus retrieving from the top end can be much better in the long run. Compared with such heuristic, T-*descending* is much simpler and empirically almost as good as the optimal order, as discussed above.

Group-Aware and Rank-Aware Plans: Exploiting Group- and Tuple-Ranking Principles

To exploit the Group-Ranking Principle, our proposed new *rankagg* operator (Figure 3.1(b)) explicitly controls the inter-group ordering. Instead of passively waiting for the underlying subtree to fully materialize all the groups, the *rankagg* operator *actively* determines the most promising group g according to the maximal-possible scores of all valid groups, and draws the next tuple in g from the underlying subtree. (By Requirement 2, any current top-k incomplete group can be such g to request.) When the most promising group is complete, its aggregate is returned as a query result. Therefore, the groups are always output from the *rankagg* operator in the ranking order of their aggregates, eliminating the need for the blocking sorting operator in Figure 3.1(a).

This "active grouping" is a clear departure from the materialize-group-sort scheme and it requires changing the interface of operators. Specifically, we change the *GetNext* method of the iterator to take g as a parameter. Our operators are thus *group-aware* so that grouping is seamlessly integrated with other operations. Recursively starting from the *rankagg* operator, an upper operator invokes the *GetNext*(g) methods of its lower operators, providing the most promising group g as the parameter. For a unary operator, the same g is passed as the parameter to its lower operator. For a binary operator such as join, g is decomposed into two components g' and g'' and are passed as the parameters to the left and the right child operators, respectively. In response, each operator sends the next output tuple from the designated group g to its upper operator.

To enforce the aforementioned T*-descending* heuristic, the query tree underlying *rankagg* must be *rank-aware* as well. For this purpose, we leverage our framework of ranking query processing in Chapter 2. However, we must address the challenges in satisfying group-awareness and rank-awareness together.

Example 11: Consider again Example 8. Figure 3.3 illustrates (a) a group-aware plan which we call GroupOnly and (b) its execution. The group-aware scan operator can produce tuples from the group designated by the *rankagg* operator above it. The tuples within each group are produced in their on-disk order. To contrast, Figure 3.4 illustrates (a) a plan that is both group-aware and rank-

aware which we call GroupRank and (b) its execution. The group- and rank-aware scan operator in this plan produces tuples in the descending order of $R.v$ within each group. The executions of these two plans are already explained in Examples 8 and 9. Note that GroupOnly does not exhaust the tuples in g_2 and does not touch the tuples in g_3 at all. GroupRank takes even fewer steps than the GroupOnly plan, by exploiting order within groups. ■

Variations of Query Plans: Trading off Group- and Rank-Awareness

The group- and rank-aware query plans can be much more efficient than traditional plans. We call them GroupRank plans (*e.g.*, Figure 3.4(a)). However, there can be situations under which these plans are inapplicable or inefficient, therefore we propose variations of plans. These variations cover many different applicable situations, thus serve as a robust solution that provides better strategies than the traditional approach. Moreover, both group-awareness and rank-awareness can bring overhead, respectively. For example, to enable rank-aware join, we adapt techniques in recent work [57], where the join operator buffers the joined tuples in ranking queues. Maintaining the ranking queues can bring significant overheads or even offset the advantages of processing joins incrementally. Therefore the variations provide ways to trade off their overheads. We study the performances of these plans in Section 3.5. However, we leave the problem of optimizing among multiple applicable plans as our future topic.

First, GroupOnly plans (*e.g.*, Figure 3.3(a)), where the operators are group-aware but not rank-aware. The *rankagg* operator still gets the next tuple from the most promising group, but in arbitrary order within each group. Such plans are applicable when ranking on T cannot be efficiently supported. For example, ranking processing techniques require monotonic T or splitting and interleaving T, which may not be applicable in certain situations.

Second, RankOnly plans where the operators are rank-aware only. Instead of telling the underlying operator the designated group, the *rankagg* operator gets interleaved tuples from all groups and orders the groups by their aggregate scores.

Finally, GroupRank-ϵ ($0 \leq \epsilon \leq 1$) plans which are the same as GroupRank except that the join

operators output tuples out-of-order, while at the same time not in arbitrary order. Since full ranking can be expensive, we experiment with approximations, which trade ranking overhead with precision of tuple ranking. In a GroupRank-ϵ plan, upon the request of sending the next tuple from a given group, a join operator outputs the top tuple t in its ranking queue for that group if $ub_t \geq ub \times \epsilon$, where ub_t is the upper-bound of t and ub is the upper-bound of the unseen tuples. The greater value between ub_t and ub is reported to the upper operator as the upper-bound of any future tuples to be reported. Note that the scan operators in GroupRank-ϵ are still rank-aware and group-aware. It is clear GroupRank is actually an extreme case, GroupRank-1. As another extreme case, in GroupRank-0, a join operator outputs the top tuple in the ranking queue of a group whenever the queue is not empty. Note that GroupRank-0 is not a GroupOnly plan as all seen tuples in the ranking queue are still ordered.

3.4.2 Implementing the New `rankagg` Operator

The iterator interface for *rankagg* is shown in Figure 3.6. The *rankagg* operator maintains a priority queue storing the upper-bounds of groups that are not output yet. Note that the priority queue in *rankagg* and the ranking queues in the group- and rank-aware join operators serve different purposes. While the ranking queues in joins are used to buffer tuples for providing ranking access to the tuples, the priority queue is used for efficiently maintaining the current top group dynamically. The *rankagg* always gets the next tuple from the top group in the priority queue and updates its upper-bound. When the top group is complete, it is guaranteed to be the best among those in the queue, thus can be reported. (In RankOnly plans, a hash table instead of priority queue is used to give fast access to the upper-bounds. An iteration through the hash table is performed periodically and the top group is output when it is complete.) Below we discuss how to maintain the routines for upper-bound computation and how to manage the priority queue.

Upper-Bound Computation: For a ranking aggregate F=$G(T)$, the maximal-possible score of a group g with obtained tuples $\mathcal{I}_g$, $\overline{F}_{\mathcal{I}_g}[g]$, can be computed by Eq. 3.3. Starting from the initial

```
1: //input: the underlying operator.
2: //k: the requested number of groups.
3: //q: the priority queue of groups.
4: //g.obtained: the number of obtained tuples in g, i.e., |I_g|.
5: //g.count: the size of g.
Procedure Open()
1: input.Open(); q.clear()
2: for each group g do
3:    init_ub(g); q.insert(g)
4: return
Procedure GetNext()
1: while true do
2:    if k==0 ∨ q.isEmpty() then
3:       Close()
4:       return
5:    g ← q.top()
6:    if g.count==g.obtained then
7:       finalize_ub(g); k ← k − 1
8:       return g
9:    t ← input.GetNext(g); update_ub(g,t); q.insert(g)
Procedure Close()
1: input.Close(); q.clear()
2: return
```

Figure 3.6: The interface methods of *rankagg*.

upper-bound, we must keep updating $\overline{F}_{\mathcal{I}_g}[g]$ when tuples are incrementally obtained. When the last tuple from g is obtained, $\overline{F}_{\mathcal{I}_g}[g]$ becomes the aggregate value $F[g]$. This description clearly indicates that the upper-bound itself can be maintained by an external aggregate function. (Let's call it *upper-bound routine*.) For example, in POSTGRESQL, a user-defined aggregate function is defined by an initial state, a state transition function, and a final calculation function. Therefore for the G in a ranking aggregate query F=$G(T)$, the corresponding upper-bound routines consist of *init_ub*, *update_ub*, and *finalize_ub*. They are invoked in the interface methods of *rankagg* (Figure 3.6). Such routines can be pre-defined if G is a built-in function. For example, Figure 3.7 illustrates the upper-bound routines for G=sum. As an alternative, in GroupOnly plans, the upper-bound in the *update_db* procedure should become $g.ub \leftarrow g.sum + (g.size - g.count) \times \overline{T}_g$. When G

1: //$g.ub$: the maximal-possible score of a group g, *i.e.*, $\overline{F}_{\mathcal{I}_g}[g]$.
2: //$g.sum$: the sum of T for obtained tuples in g.
3: //$\overline{T}_{\mathcal{I}_g}$: the maximal-possible value of T among g's unseen tuples, retrieved in T-*descending* order.
4: //$\overline{T}_g$: the initial $\overline{T}_{\mathcal{I}_g}$ when no tuple is obtaind.

Procedure init_ub(g)
1: $g.sum \leftarrow 0$; $g.obtained \leftarrow 0$
2: $\overline{T}_{\mathcal{I}_g}$=$\overline{T}_g$
3: $g.ub = g.count \times \overline{T}_{\mathcal{I}_g}$
4: **return**

Procedure update_ub(g,t)
1: $g.sum \leftarrow g.sum + T[t]$
2: $g.obtained \leftarrow g.obtained + 1$
3: $\overline{T}_{\mathcal{I}_g}$=$T[t]$
4: $g.ub \leftarrow g.sum + (g.count - g.obtained) \times \overline{T}_{\mathcal{I}_g}$
5: **return**

Procedure finalize_ub(g)
1: //nothing needs to be done
2: **return**

Figure 3.7: The upper-bound routines for G=sum.

is a user-defined aggregate function itself, the upper-bound routine is defined by straightforward adaptation of the utilities (initialization, state transition, final calculation) of G, mainly to substitute the value of unknown tuples with $\overline{T}_{\mathcal{I}_g}$. We omit further discussion of these details.

Efficient Ranking Priority Queue Implementation: For a ranking aggregate query, the total number of groups can be huge although only the top k groups are requested. For example, joining three tables with $1,000$ groups on each table can potentially lead to 1 billion joined groups. Managing the upper-bounds of the huge number of groups by a simple priority queue implementation can thus bring significant overhead.

We address this challenge from two aspects, illustrated by our new priority queue in Figure 3.8. *First*, we populate the priority queue incrementally. It is necessary to insert a group into the priority queue only when its maximal-possible score is among the current top-k, by Requirement 2. By using a global tuple max (the overall highest T value across all groups), the tuple count effec-

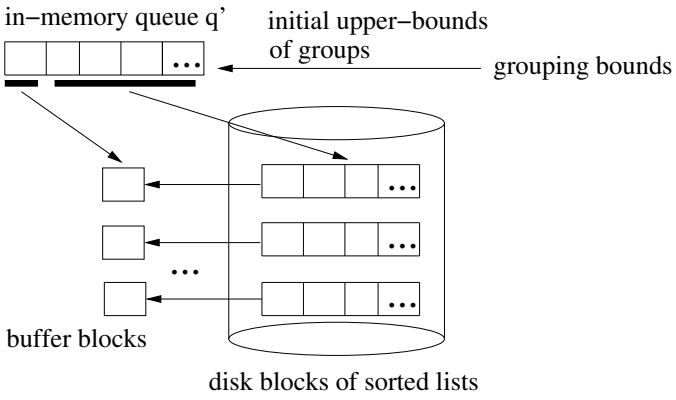


Figure 3.8: Priority queue.

tively determines the initial maximal-possible score of every group, based on Eq. 3.3. Therefore the groups can be incrementally inserted from higher to lower counts, utilizing the index on the tuple count. Such index over summary tables is extensively built and utilized in decision support. Moreover, there are techniques (*e.g.*, [70]) for getting the groups with the largest sizes (incrementally). *Second*, when the (incrementally expanding) priority queue does become too big to fit in the memory, we use a 2-level virtual priority queue q consisting of (1) an in-memory priority queue q' (implemented by the heap algorithm), and (2) a set of in-memory buffer blocks of sorted lists and a set of on-disk sorted lists.

Initially, only the first batch of groups ($1,000$ in our experiments) with the largest counts are inserted into q'. Whenever q' is full, it is emptied and its elements are converted into a sorted list (ordered by upper-bounds), of which the first top block is kept in buffer and the rest is sent to the disk. When a request is issued to get the top element (group) from q, the top elements from q' and from every buffer block are compared and the overall top group is returned. When a buffer block is exhausted, the next block from the corresponding sorted list is read from the disk into the buffer. If the top group is complete, it is returned as a query result, otherwise the next tuple from the group is obtained to update its upper-bound and the group is inserted back to q. It is possible the upper-bound of the top group becomes smaller than that of the group with the largest size among those that are not inserted. Under such situation, the next batch of groups are inserted into q'.

With the new priority queue, only the top groups (which are more likely to remain at the top) are kept in memory, in analogy to various cache replacement policies. Moreover, many groups may have initial upper-bounds smaller than the top-k threshold θ, thus may even never be necessarily touched when the top k answers are obtained. Therefore our concern with the potentially huge number of groups is addressed, as verified by the experiments in Section 3.5.

3.4.3 Impacts to Existing Operators

In this section we discuss the impacts of *rankagg* to other query operators, scan and join in particular.

Scan: To be group-aware, the new scan operator must access the next tuple in the group g requested by its upper operator. In [51], a round-robin index striding method was introduced to compute on-line aggregates with probabilistic guarantees. Our scan operator adopts the index striding technique. Multiple *cursors*, one per group, are maintained on the index to enable such striding. A cursor is advanced whenever a tuple is obtained from the cursor. However, there are two important differences: (1) in our case, index retrieval is governed by the dynamically designated group instead of fixed weights; and (2) to access tuples within each group in the descending order of T, *i.e.*, to be rank-aware, we build multi-key index, by using the grouping attribute as the first key and the attribute in T as the second key. For example, for the following query:

select $R.g$, $S.g$, **sum**($R.v$+$S.v$) **from** R, S

group by $R.g$, $S.g$ **order by sum**($R.v$+$S.v$) **limit** 1,

a multi-key index on ($R.g$, $R.v$) can be used for accessing R and another index on ($S.g$, $S.v$) for S. (Similarly when there are multiple grouping attributes on a table.) Note that we do not discuss how to select which indices to build, as such index selection problem has been studied before (*e.g.*, [45]) and is complementary to our techniques. When index on a table is unavailable, we have to scan the whole table and build a temporary index or search structure.

Join: For group-awareness, when a join operator is required to produce a tuple of group g, it

outputs such a tuple from its buffer when available, otherwise it recursively invokes the *GetNext*(g') and *GetNext*(g'') methods of its left and right input operators, respectively. For instance, for the above query, suppose a join operator that joins R and S is requested by *rankagg* to output the next tuple from a group ($R.g$=1, $S.g$=2). The join operator directly returns a joined tuple from its buffer when available. Otherwise, it requests the next tuple with $R.g$=1 from R or the next tuple with $S.g$=2 from S.

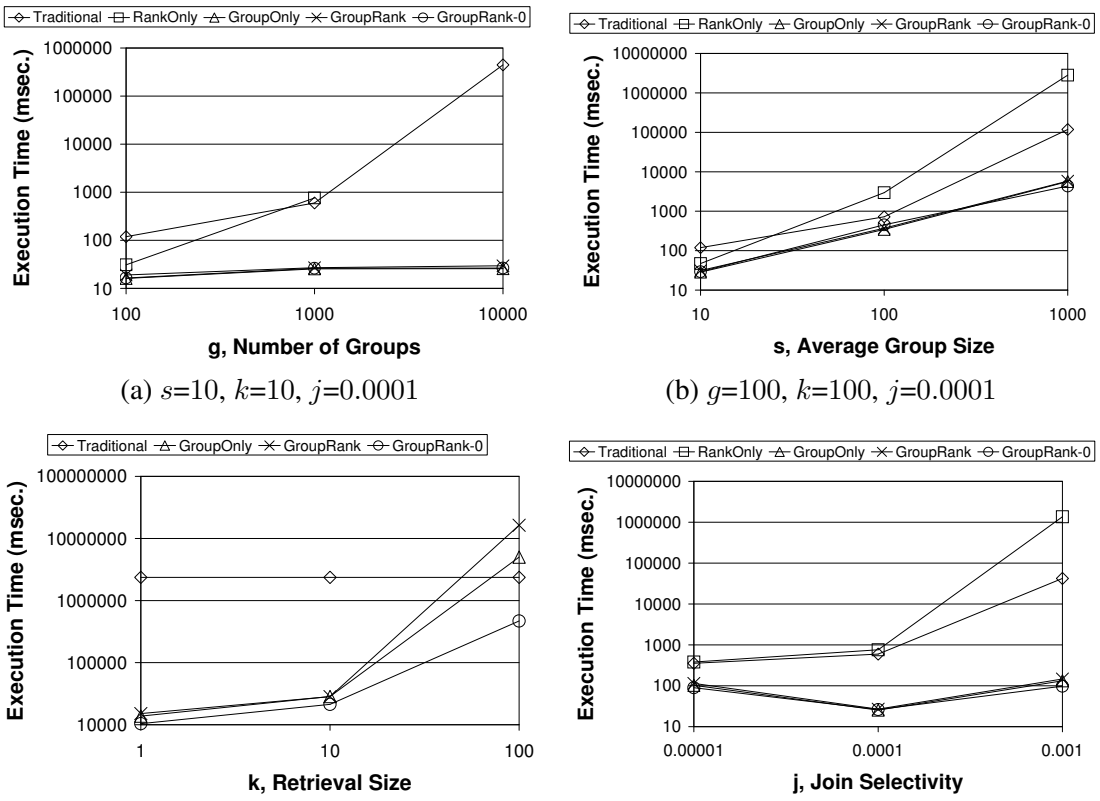
To be rank-aware, the join operator must output joined tuples in the order with respect to T, *e.g.*, $R.v$+$S.v$. We adopt the HRJN algorithm [57]. The algorithm maintains a ranking priority queue (not to be confused with the priority queue in Section 3.4.2) for buffering joined tuples, ordered on their upper-bound scores. The top tuple from the queue is output if its upper-bound score is greater than a threshold, which gives an upper-bound score of all unseen join combinations. Otherwise, the algorithm continues by reading tuples from the inputs and performs a symmetric hash join to generate new join results. The threshold is continuously updated as new tuples arrive. In the new implementation, we manage multiple ranking queues, one for each joined group and use a hash table to maintain the pointers to each ranking queue. In GroupOnly plans, the join operator uses a FIFO queue instead of priority queue to buffer join results (thus HRJN becomes the hash ripple join [47]).

3.5 Experiments

3.5.1 Settings

The proposed techniques are implemented in POSTGRESQL. The experiments are conducted on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 512KB cache), 2GB RAM, and 260GB RAID5 array of 3 SCSI disks, running Linux 2.6.9.

We use a synthetic data set of three tables (A, B, C) with the same schema and similar size. Each table has one join attribute jc, one grouping attribute g and one attribute v that is aggregated.



(a) s=10, k=10, j=0.0001

(b) g=100, k=100, j=0.0001

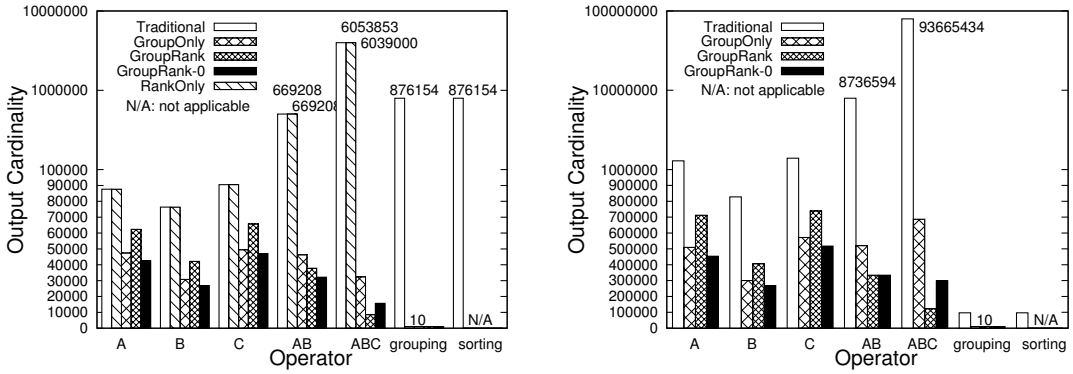
(c) g=100, s=10,000, j=0.00001

(d) g=1,000, s=10, k=10

Figure 3.9: Performance of different execution plans.

For each tuple, the three attribute values are independently generated random numbers. In each base table, the values of v follow the uniform distribution in the range of $[0, 1]$. The number of distinct values of j is $\frac{1}{j}$, where j is a configurable parameter capturing join selectivity. The values of j follow the uniform distribution in the range of $[1, \frac{1}{j}]$. The number of distinct values of g is g, *i.e.*, g captures the number of groups on each table. For example, when g=10, the maximal number of joined groups over ABC is g^3=1,000. The number of tuples corresponding to each distinct value of g follows normal distribution, with average s, *i.e.*, s is the average size of base table groups.

We use the star-join query $\mathcal{Q}$ in Section 3.1. We compare five execution plans, Traditional, RankOnly, GroupOnly, GroupRank (*i.e.*, GroupRank-1), and GroupRank-0. They have the same plan structure that joins A with B and then with C. Traditional is an instance of the materialize-group-sort plan in Figure 3.1(a). It uses sort-merge join as the join algorithm and scans the base ta-



(a)g=100,s=1,000,k=10,j=0.0001 (b)g=100,s=10,000,k=10,j=0.00001

Figure 3.10: Output cardinalities.

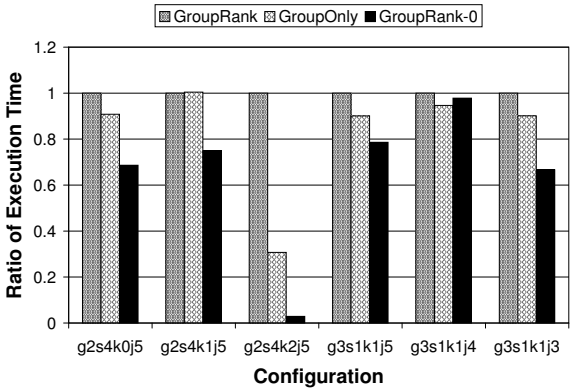


Figure 3.11: Comparisons of the new plans.

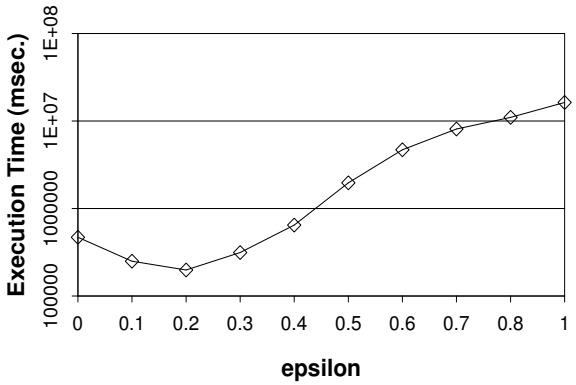


Figure 3.12: Performance of GroupRank-ϵ.

bles by the indices on the join attributes. The RankOnly, GroupOnly, GroupRank, and GroupRank-0 use the new *rankagg* operator. Moreover the join and scan operators in these plans are group-aware and/or rank-aware, as described in Section 3.4.1. We executed these plans under various configurations of four parameters, which are the number of requested groups (k), the number of groups on each table (g), the average size of base table group (s), and the join selectivity (j). We use $gWsXkYjZ$ to annotate the configuration g=10^W, s=10^X, k=10^Y, and j=10^{-Z}.

3.5.2 Results

We first performed 4 sets of experiments. In each set, we varied the value of one parameter and fixed the values of other three parameters, among k, g, s, and j. The plan execution time

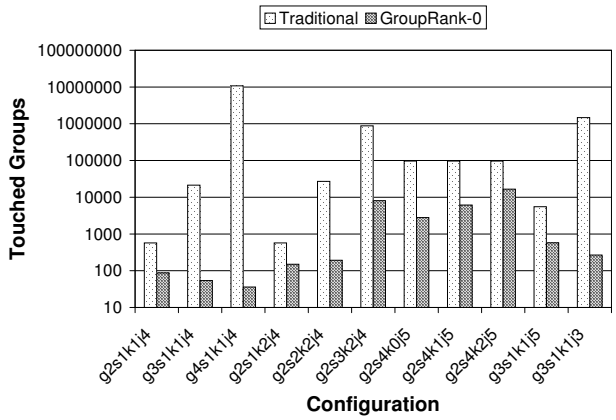


Figure 3.13: Touched groups.

under these settings is shown in Figure 3.9. (Both x and y axes are in logarithmic scale.) The figure clearly shows that our new plans outperformed the traditional plan by orders of magnitude. Traditional is only comparable to the new plans when there are not many groups, the group size is small, many results are requested, and joins are very selective. RankOnly is as inefficient as Traditional. It did not finish after running for fairly long under some configuration (g=$10,000$ in Figure 3.9(a)) and is excluded from Figure 3.9(c) for the same reason. As an intuitive explanation, if the top-1 group has a member tuple that is ranked at the last place, all the groups must be materialized in order to obtain the top-1 group. This indicates that being rank-aware itself [57, 67] does not help to deal with top-k aggregate queries.

The differences among the new plans are not obvious in Figure 3.9(a)(b)(d) because Traditional and RankOnly are too far off the scale. However, Figure 3.9(c) clearly illustrates their differences. In Figure 3.11, we further compare GroupOnly, GroupRank and GroupRank-0 under the 8 configurations in Figure 3.9(c)(d). For each plan, we show the ratio of its execution time to the execution time of GroupRank. The results show that GroupOnly in many cases is better than GroupRank, verifying that the ranking overhead can offset the advantages of group-awareness in certain cases. On the other hand, the performance is much improved when we reduce the ranking overhead, as GroupRank-0 almost always outperformed GroupOnly and GroupRank.

We further analyze these plans by comparing the output cardinalities of their operators. Fig-

ure 3.10 reports the comparisons under two configurations. The results for other configurations are similar. As it shows, Traditional enforces full materialization. RankOnly was not able to reduce the cardinalities and further incurred ranking overhead, which explains why it is even worse than Traditional in many cases. GroupOnly reduced the cardinalities significantly by partial consumption of base tables and partial materialization of join results. GroupRank produced less join results than GroupOnly because of rank-awareness. However, it also consumed more base table inputs because join operators must buffer more inputs to produce ranked outputs (the ranking overhead). Finally, GroupRank-0 balanced the benefits and overhead of rank-awareness, as explained in Section 3.4.1. Therefore it consumed less number of base table inputs, although produced some more join results.

To further study the tradeoff in being rank-aware, we show the performance of GroupRank-ϵ in Figure 3.12 by ranging ϵ from 0 to 1. Note that GroupRank and GroupRank-0 are extreme cases for ϵ=1 and 0, respectively. Interestingly none of them is the best, which indicates the choice of ϵ should be captured by query optimizer.

We verify that managing the priority queue of *rankagg* (Section 3.4.2) does not require significant overhead, although the total number of groups can be potentially huge. In Figure 3.13, we compare the number of joined groups touched by GroupRank-0 and Traditional under the 11 distinct configurations from Figure 3.9. (We count a group as "touched" if at least 1 tuple from the group is produced during the plan execution. Therefore the touched groups are maintained by the priority queue and the top k groups come from the touched groups.) The results show that most of the groups never need to be touched by the new plans, therefore it is not expensive to maintain the priority queue. Figure 3.13 also clearly illustrates why the new plans outperform Traditional, together with Figure 3.10. While Traditional processes every group and every tuple in each group due to its nature of full materialization, our new plans save significantly by the early pruning resulting from the group-ranking and tuple-ranking principles.

Our framework requires tuple count, which can be obtained as discussed in Section 3.4.1.

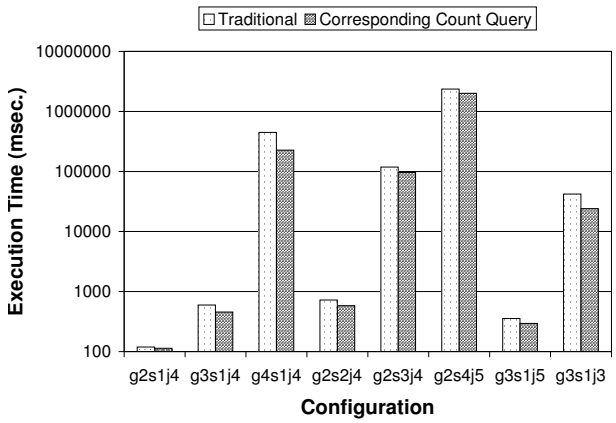


Figure 3.14: The cost of computing counts from scratch.

Specifically, when the tuple count must be computed from scratch by a count query, the cost of answering one ranking aggregate query consists of the cost of the corresponding count query and executing the new query plan based on the obtained counts. In Figure 3.14, we compare the costs of Traditional and the count query under 8 configurations from Figure 3.9. Note that k is irrelevant in this experiment since Traditional generates the total order of all groups and the count query generates the count of every group. (There are overlapping configurations in Figure 3.9(a)-(d) when k is ignored, resulting in totally 8 distinct configurations.) The results verify that computing the count query is slightly cheaper than the original ranking aggregate query. Since our new query plans are orders of magnitude more efficient than the traditional plan, the total cost of a count query and a new plan is comparable to, or even cheaper than, that of the traditional plan. More importantly, the materialized tuple counts are then used by the future related ranking aggregate queries that share the same Boolean conditions with the original query (scenario 1 in Section 3.4.1), or of which the tuple count can be computed from the materialized counts (scenario 2). Nevertheless, it brings us the advantages of "paying one, getting the following (almost) free".

Discussions: We should emphasize that although the new query plans are not always equally efficient, they provide better strategies than the traditional approach in processing top-k aggregate queries, under various applicable conditions, as discussed in Section 3.4.1. Moreover, the experimental results indicate that none of the plans is always the best and their costs can be orders of

magnitude different. Their diverse applicability and performance thus call for new query optimization techniques. Especially, the performance of our methods depends on multiple parameters, including the number of groups, the sizes of groups, the distribution of tuple scores, the memory buffer size, and so on. Thus a cost model incorporating these parameters to estimate the costs of plans is the key to the new optimizer. The estimates can enable us to choose among the new plans and even the traditional plans. Developing such a cost model and optimizer thus is an important topic of future research.

Chapter 4

Beyond Ranking: Supporting Ranking and Clustering as Generalized Order-By and Group-By

Toward a systematic support, while *ranking* conceptually generalizes *Order-By* into fuzzy ordering, this chapter aims at generalizing *Group-By* into "fuzzy" grouping– or clustering– to form a more complete suite of solutions for data retrieval. We propose to integrate the two generalizations, allowing clustering and ranking of database query results together. This integration is non-trivial in terms of both semantics and query processing. We investigate a variety of semantics in defining such queries. To process the queries, our solution is to do clustering and ranking using dynamically constructed data summary. We realize this approach by utilizing bitmap index to build summary on-the-fly and to integrate Boolean filtering, clustering, and ranking. Experimental study shows that our approach significantly outperforms the straightforward one and maintains high clustering quality.

Note that our proposal is essentially different from the clustering that has been extensively studied for years in many areas including machine learning, pattern recognition, and data mining [60, 48]. In typical data mining applications, clustering is more or less an infrequent or one-shot operation, over static data, and performed by a small number of analysts. In contrast, the fuzzy clustering engaging us in this study is a day-to-day operation, upon dynamic results of Boolean conditions, over different clustering attributes, and requested by a large number of users. Therefore our focus is to efficiently support such *on-the-fly* clustering and yet maintain high quality of clustering. Moreover, we integrate clustering with Boolean filtering and ranking. The proposal is not to replace the existing clustering algorithms. Instead, we simply adopt existing algorithms and focus on how to cluster over data summary dynamically constructed.

In summary, this chapter makes the following contributions:

- **Concept: Generalizing Group-By for Fuzzy Grouping**. We propose to support clustering with SQL as a generalization for **group-by**, parallel ranking for **order-by**. Moreover, to the best of our knowledge, ours is the first in the literature to propose the integration of fuzzy grouping and ranking in relational databases.
- **Framework: Summary-based Processing**. We develop on-the-fly summary construction and summary-based clustering and ranking, for efficient query support.

The rest of the chapter is organized as follows. We first motivate the problem in Section 4.1. We then define a new type of queries, explore its semantics in supporting fuzzy ranking and grouping, and discuss the challenges in processing such queries. An overview of our solutions is given in Section 4.3. We further present the detailed data structure and algorithms in Section 4.4 and optimization heuristics in Section 4.5. The experimental results are discussed in Section 4.6.

4.1 Motivation

Example 12 (Motivating Scenario: House Search): Consider user Amy looking for a house in Chicago, from a database House(id, price, size, zipcode, longitude, latitude, rating). She will consider houses priced below $300k, and is willing to pay a little more *if* the size is large. She would like to consider different areas– She will accept even more expensive choices *if* they are near the lakeshore; but *if* there is no such choices, she would like to look for better prices in the suburb. Notice the many "if"s in her preference, which depend on what the database can offer. Will a SQL-based database, such as *realtor.com*, support her "exploration" by querying effectively? ■

As Example 12 illustrated, data retrieval essentially mandates *result exploration*, for users to explore what choices are available in the database, and how they match the query criteria. We ask: What functions shall we support for such exploration?

While we want to equip SQL-based querying with such exploration, the answers seem to, interestingly, lie in the design of SQL itself: With the test of time, SQL has proven to be a well-designed, compact suite of constructs, balancing both functionality and simplicity. Can we draw inspiration from SQL constructs to support data retrieval, in particular, to explore query results? For this purpose, **order-by** and **group-by** both stand out as the pillars for organizing results– *e.g.*, for our example `House` relation, we may

$$\textbf{group-by}\ \mathsf{zipcode}\quad \textbf{order-by}\ min(\mathsf{price}). \qquad (S_1)$$

By ordering and grouping on attribute values, RDBMS can organize results for tabular presentation and report generation.

To begin with, many recent works have attempted to generalize **order-by** beyond "crispy" result ordering– For the same syntactic construct, these efforts seek to generalize the semantics from *ordering* of attribute values to *ranking* of matching qualities. Therefore, from ordering to ranking, the generalization boils down to 1) supporting fuzzy scoring functions (instead of only attributes) and 2) targeting at only top-k (partial) results (instead of total ordering). For our `House` example, we may order by some preferred balance of price, size, and realtor rating, and look for only the top 10 results:

$$\textbf{order-by}\ \frac{\mathsf{size}}{\mathsf{price}} \cdot \mathsf{rating}\quad \textbf{limit}\ 10 \qquad (S_2)$$

Drawing these insights from SQL constructs, as **order-by** has been generalized into ranking, as a parallel step toward result exploration, we propose to generalize **group-by**. Just like from ordering to ranking, we believe current grouping has two major limitations: *First*, the prerequisite of data understanding: As a dilemma, while grouping should help users to learn the data distribution of available choices in the database, in its current semantics, users must know this "distribution" in order to specify a good grouping scheme– For instance, is **group-by** zipcode meaningful? (How if there are 1000 different zipcodes, thus 1000 groups?) *Second*, the limitation of equality partitioning: Inherited from SQL's "crispiness," current grouping semantics partitions the space only by

"identical" values. For instance, is **group-by** longitude, latitude meaningful? (As no houses will share the same coordinates, we should instead group by their proximity.)

Our solution, much like from ordering to ranking, is to generalize "crispy" grouping to "fuzzy" grouping– or *clustering* [60, 48]. As a well-established technique for data exploration, in abstraction, with input of attributes c_1, ..., c_m and a result size of t, clustering will output t groups, or *clusters*, that best partition the space according to how objects are *similar* in c_1, ..., c_m (instead of strict equality of values). It thus ameliorates the two limitations simultaneously: For the input specification, users simply specify the desired number of t clusters, much like the desired result size k in top-k ranking, and the system will automatically weigh in the data distribution to generate t clusters. (So even if there are 1000 zipcodes, they will be grouped into a small number of t clusters.) Further, as the grouping criteria, clustering will form partitions by data distribution. Similar objects that do not share strictly identical values in c_1, ..., c_m will still be grouped. (Thus grouping on longitude and latitude will put together houses in similar locations.)

This clustering, or fuzzy grouping, maybe expressed[1] as follows, with an additional "**into** t" to indicate the target number of groups that the fuzziness should achieve. For our example, a user may want to cluster houses into 5 groups by their location proximity:

group-by longitude, latitude **into** 5 (S_3)

We examine this generalization from grouping to clustering, for supporting data retrieval with SQL. What should be the "fuzziness" of grouping– or, what clustering algorithms should we assume? Ideally, in a comprehensive setting, the system shall support a set of clustering schemes as operators to choose, and even allow extensions by, say, external functions. However, as a first step to start with, and to focus on the essence of the problem, we study *K-means* as the clustering scheme, because it is the most well-known and widely applicable partitioning-based clustering method [48] and is by far the most popularly used method in scientific and industrial applications

[1] We focus on the generalization of "functionality" and not syntax. The same can be expressed with OLAP functions. See footnote 2.

[8]. Our framework can apply other distance-based clustering methods, as long as the distance or similarity functions are based on the proximity of attribute values. Section 4.3.2 will discuss such extensions. Thus, in a more general setting, to specify our choice of clustering, we may express it as:

$$\textbf{group-by } \textit{K-means}(\mathsf{longitude}, \mathsf{latitude}) \quad \textbf{into } 5 \qquad (S_3')$$

Putting together, in a SQL-like syntax, we may express the complete suite of clustering and ranking in the following form.

select	$\ldots$		
from	$T_1, \ldots, T_s$		
where	$\mathcal{B}(b_1, \ldots, b_h)$		
group by	$c_1, \ldots, c_m$	**into**	t
order by	$\mathcal{F}(r_1, \ldots, r_n)$	**limit**	k

In this complete form, our generalization boils down to two challenges: *First*, for *the context of* SQL, as SQL has been well developed for managing structured data, our clustering must integrate with the core Boolean constructs. What does such integration mean? As **group-by** is meant to execute after the **where** clause with Boolean selection or join, our clustering should similarly partition with respect to the "dynamic" result $\sigma_{\mathcal{B}}(T_1 \times \ldots \times T_s)$. That is, the dynamic Boolean result, instead of the static tables, is the "population" whose data distribution will define the clusters. How can we execute clustering efficiently after such dynamic filtering?

Second, for *our objective of data retrieval*, as both pillars of result exploration, clustering and ranking must be seamlessly integrated. What does such integration mean? In standard SQL the combination of **group-by** and **order-by** will lead to ordering among groups (thus S_1 will return groups ordered by their minimal prices). While such *order-among-groups* is a useful semantics, we believe it is equally (if not more) important to support *order-within-groups*. In data retrieval scenarios, as evident from similar functions for text retrieval (*e.g.*, Web search), when clustering and ranking are combined (*e.g.*, *vivisimo.com*), clustering will partition the results into alternative groups, and ranking then orders answers within each group. In fact, for crispy **group-by**, this

order-within-groups semantics can be realized in OLAP *functions*[2] [102], which was introduced in SQL-99 and supported by major RDBMSs. Thus, for combining ranking and clustering, we consider different groups as equal alternatives, and only those answers within the same group are compared in ranking. For example, we may cluster houses by areas (as in snippet S_3) and only rank among those in the same area by prices and sizes (as in S_2). Can we support such integration of ranking "within" clustering efficiently?

4.2 The Proposal: Clustering + Ranking in Database Queries

4.2.1 The *ClusterRank* Query

We introduce a new type of *ClusterRank* query. The semantics of such a query is to conceptually perform the following three steps. Note that we ignore less important operations in our context such as attribute projection.

- **Filtering**: Upon a base relation or the Cartesian product of base relations, we apply Boolean function $\mathcal{B}$, resulting in a relation of qualifying tuples, $\sigma_{\mathcal{B}}$;
- **Clustering**: The tuples in $\sigma_{\mathcal{B}}$ are partitioned into t clusters, based on the clustering attributes $c_1, \ldots, c_m$;
- **Ranking**: A scoring function $\mathcal{F}$ defined over a set of ranking attributes R assigns a ranking score $\mathcal{F}(R)[t]$ to each tuple t. Within each cluster, the top k tuples with the highest scores (or all if there are less than k tuples in the cluster) are returned. [3]

In relational database, currently no SQL syntax can support such queries, nor can OLAP functions express our query. Still, since OLAP functions support ranking within a group or a partition, the closest way to express our semantics maybe the following:

[2] In DB2, $rank()$ **over** (**partition by** $attr_1, \ldots, attr_m$ **order by** v) groups tuples by $attr_i$ and orders tuples in each group by v.

[3] When there are ties in scores, an arbitrary deterministic "tie-breaker" can determine an order, *e.g.*, by unique tuple IDs.

select	..., *rank*() **over** (	**partition by**	*K-means*$(t, c_1, \ldots, c_m)$
		order by	$\mathcal{F}(r_1, \ldots, r_n)$
		) **as** $score\_rank$	
from	$T_1, \ldots, T_s$		
where	$\mathcal{B}(b_1, \ldots, b_h)$		
when	$score\_rank <= k$		

Besides the fact that OLAP does not support functions such as *K-means*$(t, c_1, \ldots, c_m)$ in the **partition by** clause, there is a fundamental difference between the task achieved by the above query and the goal we want to achieve. The query treats *K-means*$(t, c_1, \ldots, c_m)$ as a black box, which prevents the system from optimizing the query. Instead, we focus on integrating the ranking and the clustering process in a tight manner, so that we can minimize the cost of the *ClusterRank* query.

In essence, our semantics is based on the concept of fuzzy clustering. We require that partitions have fuzzy boundaries, and we specify the total number of clusters, as in *K-means*. Borrowing the syntax of SQL, we denote fuzzy clustering by "**group by** ... **into** ...", and our goal is to integrate it with the "**order by** ... **limit** ..." clause. The sketch of such a query is shown below [4].

select	...		
from	$T_1, \ldots, T_s$		
where	$\mathcal{B}(b_1, \ldots, b_h)$		
group by	$c_1, \ldots, c_m$	**into**	t
order by	$\mathcal{F}(r_1, \ldots, r_n)$	**limit**	k

More formally, a *ClusterRank* query Q is a SPJ query augmented with clustering and ranking conditions. The query consists of the following tables, attributes, functions, and constants.

- $\mathcal{T}$: a set of tables $\{T_1, \ldots, T_s\}$;
- $\mathcal{B}$: a *Boolean function* $\mathcal{B}$ over a set of attributes $b_1, \ldots, b_h$. The Boolean function $\mathcal{B}$ can be a complex Boolean condition such as conjunctions and disjunctions of sub-conditions;
- $c_1, \ldots, c_m$: a set of *clustering attributes*
- t: the number of clusters;

[4] For simplicity, we assume **order by asc**|**desc** uses descending order as default, although ascending is the default in some systems.

- $\mathcal{F}$: a *ranking function* (a.k.a. *scoring function*) over the *ranking attributes* $r_1, \ldots, r_n$;
- k: the number of top tuples to retrieve within each cluster.

We want to point out that clustering, by nature, is a fuzzy and unstable operation, as different algorithms and configurations on the same data will generate different clusters. Therefore on the one hand, in contrast to the deterministic semantics of conventional database queries, a *ClusterRank* query may generate different answers in each run. On the other hand, such non-determinism is consistent with our goal of enabling fuzzy data retrieval and exploration, and we believe sacrificing the crispness of queries is worthwhile.

Note that the above syntax is for illustrating our concept only, as the use of SQL's **group-by** has many restrictions. The main reason is that when **group-by** is present, the columns in **order-by** must either appear in the columns of **group-by** or be some aggregate functions. The meaning of such a query is to order the groups based on some grouping attributes or aggregate values over the groups. Moreover, we will not be able to specify the number of clusters desired, and **group-by** does not allow function either.

Up till this moment, we have assumed only one semantics for *ClusterRank* queries, that is, returning top k tuples within each cluster. (Call it *global clustering/local ranking*.) However, we may extend our query model to embrace a richer set of semantics, tailored for various application needs. One example is *local clustering/global ranking*, where the clustering is only performed over the global top k tuples instead of $\sigma_{\mathcal{B}}$. Another example is *global clustering/global ranking*, where within each cluster, only those tuples that belong to the global top k (instead of local top k) are returned. Moreover, we may further allow ranking of the clusters by aggregate functions. While it is very interesting to study these alternative semantics and corresponding techniques for processing queries, we focus on *global clustering/local ranking* in this work.

4.2.2 Challenges: The Problems with a Straightforward Approach

Literally following the semantics in Section 4.2.1, we obtain a straightforward approach for evaluating *ClusterRank* queries. That is to, (1) materialize intermediate Boolean results $\sigma_{\mathcal{B}}$; (2) cluster

$\sigma_{\mathcal{B}}$; (3) sort all the tuples within each cluster; and (4) return the top k tuples in each cluster.

However, such *materialize-cluster-rank* approach is clearly an overkill due to the fact that it clusters and ranks all Boolean results although we only need the top k in each cluster. It can thus be very inefficient. Materializing $\sigma_{\mathcal{B}}$ itself can be expensive, especially with joins. The cardinality of $\sigma_{\mathcal{B}}$ can be large when Boolean conditions are not selective. As a costly procedure, clustering such a large $\sigma_{\mathcal{B}}$ is expensive and may take multiple iterations. Sorting the tuples in each cluster further adds to the overhead. Moreover the tuples may be dumped out and read in many times, between materializing $\sigma_{\mathcal{B}}$ and clustering, during the iterations in clustering, and for sorting them. All these result in significant disk I/O cost.

The high overhead of materialize-cluster-rank can seriously impact the usefulness of *ClusterRank* queries. It may be acceptable if the query was only one-shot, where the clustering and ranking results, or at least $\sigma_{\mathcal{B}}$, can be even materialized beforehand. This is clearly not the case in our target applications (*e.g.*, house search in Example 12), where users *on-the-fly* specify all kinds of Boolean conditions, form clusters upon different attributes, and apply different ranking criteria over different ranking attributes.

4.3 Framework: Overview

In this section, we first give a high-level overview of our approach (Section 4.3.1), then specify the data and query model and assumptions (Section 4.3.2), and finally briefly introduce the background on bitmap index (Section 4.3.3).

4.3.1 Our Approach: Summary-Based *ClusterRank*

The materialize-cluster-rank approach in Section 4.2.2 is very costly since it involves a large amount of tuple-based operations. For clustering, the approach goes through every tuple and assigns it to its closest cluster, for iterations until the algorithm converges. For ranking, it computes the score of each tuple and sorts all the tuples in each cluster. Obviously, it obtains the clustering

and ranking results for each tuple. However, the query requests only a small portion of the tuples processed, *i.e.*, the top k within each cluster. A natural question to ask is: To reduce the cost in processing each tuple individually, can we process at a "coarser" level?

Using appropriate data summary instead of all tuples in both clustering and ranking is our answer to this question. Below we outline the summary-based *ClusterRank* approach. For *clustering*, with any distance-based method, if two tuples are close enough to each other, it is natural to assign them to the same cluster. In our approach, we use a grid-based data summary to put similar tuples into the same "bucket" and then cluster at the bucket-level. To be more specific, we perform partitioning (or binning) on each clustering attribute. The intersection of the bins over the clustering attributes gives us a summary grid with buckets. If two tuples fall into the same bucket (*i.e.*, the same bin along each clustering attribute), we can consider them the "same" tuple, *i.e.*, inseparable. Thus a bucket is the smallest unit in our clustering. As long as the bucket size is appropriate, the quality of clustering on the buckets is comparable to that on the original tuples. However, the bucket-level clustering is much more efficient than the tuple-level one, since the number of buckets is much smaller than the number of tuples.

For *ranking*, we can use a summary grid for efficient processing as well. For each cluster, the grid for the tuples in the cluster is constructed over the ranking attributes. For the tuples in each bucket, the upper-bound and lower-bound of their scores can be computed based on the boundaries of the corresponding bins on individual attributes. The bounds enable us to prune those buckets that do not contain any of the top k tuples. The top k in the unpruned candidate buckets are guaranteed to be the top k among all the tuples.

The clustering and ranking operate on two orthogonal summary grids built over clustering and ranking attributes, respectively. Note that the grids are query dependent since different queries may have different clustering and ranking attributes. Thus how to efficiently construct the grids *on-the-fly* at query time is one big challenge. Also, the clustering and ranking are on the results of Boolean conditions, thus we must integrate the Boolean filtering, clustering, and ranking in an efficient way.

We use bitmap indexes to meet the challenge and the integration goal. A bitmap index uses one vector of bits to indicate the membership of tuples for each value or each value range on an attribute. By intersecting the bit vectors for the bins over the individual clustering attributes, we construct the summary grid for clustering. The grid for ranking is constructed similarly. In summary, the bit vectors serve as the basic unit in unifying Boolean filtering, clustering, and ranking through the following steps: (1) Bit vectors are used to process the Boolean conditions, (2) The resulting bit vectors are used in building the summary grid for clustering, (3) Clustering is performed on the grid, (4) The resulting bit vectors corresponding to each cluster are used in constructing the summary grid for ranking, and (5) Ranking is performed within each cluster.

4.3.2 Data and Query Model

We assume the tables have a snowflake-schema, consisting of one fact table and multiple dimension tables. There are multiple dimensions, each of which is described by a hierarchy, with one dimension table for each node on the hierarchy. The fact table is connected to the dimensions by foreign keys. The tables on each dimension are also connected by keys and foreign keys. As a special case of snowflake-schema, star-schema has only one table on every dimension, thus no hierarchy.

With respect to the *ClusterRank* queries in Section 4.2.1, we make the following assumptions on the Boolean, clustering, and ranking conditions.

- $\mathcal{B}(b_1, \ldots, b_h)$: The Boolean condition consists of conjunctive key and foreign-key joins and range selections, including two-side range selection (*e.g.*, $10 \leq a$ **and** $a < 20$, or $10 \leq a < 20$), one-side range selection (*e.g.*, $a \leq 20$), and equality selection (*e.g.*, $a = 10$). Both sides of the two-side selection condition can be either open-end or closed-end. Note that one-side and equality selections are extreme cases of two-side selection.
- $c_1, \ldots, c_m$: The clustering attributes are all numerical attributes. We assume a *K-means* clustering algorithm. Note that the summary-based approach can be applied to other distance-based

clustering algorithms, as long as the distance function is based on the proximity of attribute values (thus the insight of considering the tuples in the same bucket inseparable is applicable). The only difference observed by the clustering algorithms is that the buckets instead of real tuples are clustered. Therefore the number of tuples in the buckets, or their weights, must be taken into consideration. The algorithms can be simply adjusted to consider such weights [103]. In general, applicable algorithms can be supported as clustering operators to choose, or even registered as external functions. The algorithms may require parameters such as stopping criteria and distance functions (*e.g.*, Euclidean or Manhattan distances). These parameters can be specified through configuration settings in database systems.

- $\mathcal{F}(r_1, \ldots, r_n)$: The ranking function is *monotonic* over numerical ranking attributes, as commonly assumed in ranking query processing [32]. Without losing generality, in this chapter we focus on the weighted-sum, a typical monotonic function. Note that our approach in fact is valid for any monotonic ranking function.

Under these assumptions, the sketch of the resulting simplified query is shown below.

select	*	
from	$T_1, \ldots, T_s$	
where	$v_1^1 \leq b_1 \leq v_1^2$ **and** ... **and** $v_p^1 \leq b_p \leq v_p^2$	
	and $b_{p+1} = b_{p+2}$ **and** ... **and** $b_{h-1} = b_h$	
cluster by	$c_1, \ldots, c_m$	**into** t
order by	$w_1 \times r_1 + \ldots + w_n \times r_n$	**limit** k

4.3.3 A Review of Bitmap Index

As an efficient index for dealing with complex decision support queries, bitmap index [79, 80, 81] has gained broad interests and has been adopted in commercial systems. For a bitmap index on an attribute, there exists a bitmap (a vector of bits) for each unique attribute value. The length of the vector equals the number of tuples in the indexed relation. With respect to the vector for value x of attribute a, its i^{th} bit is set to 1, when and only when the value of a on the i^{th} tuple is x, otherwise 0. With bitmap indices, complex selection queries can be efficiently answered by bit-wise logical

operations (**and**, **or**, **xor**, and **not**) over the bit vectors. Moreover, as studied in [81], bitmap indices also enable the efficient computation of some common aggregates, such as **sum** and **count**.

The original form of bitmap index has a problem with high-cardinality attributes. One bit vector must be created for each attribute value in the domain, resulting in big overhead of storage and maintenance if an attribute has many values. In tackling this challenge, researchers have studied a variety of ways in encoding bitmap index [16, 97]. For example, binning can be used to merge the bit vectors for a range of attribute values. Moreover, *bit-sliced index* (BSI) [84] directly captures the binary representations of attribute values.

4.4 Realization: Data Structure and Algorithms

Building on the insights provided in the overview (Section 4.3.1), we present the detailed algorithms in this section. In Section 4.4.1, we first give a formal description of summary grid, and show how to construct the grid using bitmap index. We then introduce the summary-based algorithms for clustering (Section 4.4.2) and ranking (Section 4.4.2). To simplify the discussion, our discussion first focuses on single table queries without Boolean conditions. In Section 4.4.2 we investigate how to extend our framework to incorporate selection and join conditions.

4.4.1 Data Structure: Building Summary Grids

Consider a relation T. A *partitioning attribute* a over T has a set of disjoint ranges that partition the value domain of a. More formally, a has y *partitioning points* $\{a^1, \ldots, a^y\}$ and two special *endpoints* a^0=min_a and a^{y+1}=max_a. The endpoints give the domain of a. That is, $[min_a, max_a)$ subsumes the a values of all the instances in T. The partitioning points and endpoints together define y+1 ranges over a, that are $ranges$=$\{range^0, \ldots, range^y\}$, where $range^i$=$[a^i, a^{i+1})$. Given a set of x partitioning attributes A=$\{a_1, \ldots, a_x\}$, their partitioning ranges determine a *grid* $\mathcal{G}(T,A,\{ranges_1, \ldots, ranges_x\})$. [5] The grid partitions the multi-dimensional space over A into z=$\prod_i(y_i$+1) buck-

[5] We will often use the simplified notation $\mathcal{G}(T, A)$ when there is no need to emphasize the ranges in the context.

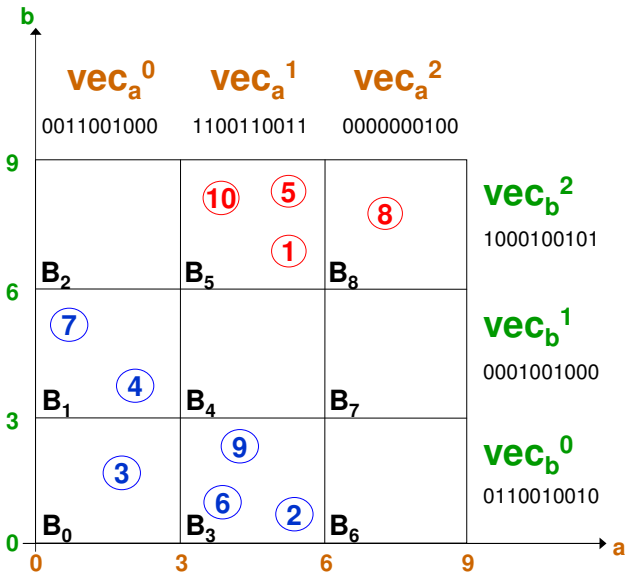


Figure 4.1: Summary grid.

ets $\mathcal{B}$=$\{B_1, \ldots, B_z\}$ [6], where each bucket is given by the ranges over A, one range per attribute. More formally, the subscription of a bucket is determined by the subscriptions of the corresponding ranges. That is, B_{id}=$<range_1^{id_1}, \ldots, range_x^{id_x}>$ = $<[a_1^{id_1}, a_1^{id_1+1}), \ldots, [a_x^{id_x}, a_x^{id_x+1})>$, where $id = \sum_{i=1}^{x-1} (id_i \times \prod_{j=i+1}^{x}(y_j + 1)) + id_x$. A bucket in the grid thus represents the intersections of the corresponding ranges. By partitioning the multi-dimensional space, the grid also partitions the tuples of T into the z buckets. That is, $B_{id} = \{t | t \in T \bigwedge t.a_i \in [a_i^{id_i}, a_i^{id_i+1}), \forall i\}$. A *summary grid* $\mathcal{S}_\mathcal{G}$ has $|B_{id}|$, the cardinality (*i.e.*, the number of tuples) of each B_{id} in $\mathcal{G}$.

Example 13 (Summary Grid): Figure 4.1 shows a grid over 10 tuples $t_1, \ldots, t_{10}$ by partitioning attributes $\{a, b\}$. The ranges on a and b are $ranges_a = \{[0, 3), [3, 6), [6, 9)\}$ and $ranges_b = \{[0, 3), [3, 6), [6, 9)\}$. There are 9 buckets, $B_0, \ldots, B_8$. The ID of each tuple is shown inside its bucket. For instance, B_5=$<range_a^1, range_b^2>$ = $<[3, 6), [6, 9)>$. It has 3 tuples (t_1, t_5, t_{10}). ■

To construct a summary grid over a relation T by a set of partitioning attributes A, we may simply go through all the tuples in T. In other words, we fully scan T if T is a base table or fully materialize T if T is the (join) query result over base table(s). As motivated in Section 4.2.2,

[6]For the ease of presentation, we abuse the notation $\mathcal{B}$ to denote buckets, although $\mathcal{B}$ denotes Boolean conditions in Section 4.2 and 4.3.

our goal is to avoid such materialize-cluster-rank approach. We thus propose a novel method to construct summary grids by intersecting bitmap index. This method not only is efficient in building the summary grids, but also benefits the clustering and ranking operations based on the grids.

Given a set of partitioning attributes A=$a_1, \ldots, a_x$, we require the existence of a bitmap index I_i over each a_i, which contains y_i+1 bit vectors $\{vec_i^0, \ldots, vec_i^{y_i}\}$, corresponding to the y_i+1 ranges over a_i, $ranges_i$. Each vector vec_i^j is a sequence of $|T|$ bits, where the k-th bit is 1 if the value of attribute a_i in the k-th tuple of relation T is within $range_i^j$, the (j+1)-th range of a_i, otherwise 0.

As a bucket in the summary grid represents the intersections of the corresponding ranges, we can obtain the members in a bucket by intersection (*bit-and* operation, *i.e.*, $\&$) of the bit vectors for the ranges. To be more specific, consider a bucket B_{id} = $<range_1^{id_1}$, ..., $range_x^{id_x}>$, we aim to construct a bit vector $vec_{B_{id}}$ that contains $|T|$ bits, where the k-th bit is 1 if the k-th tuple of T belongs to the bucket B_{id}, otherwise 0. It is thus obvious $vec_{B_{id}}$ = $vec_1^{id_1}$ $\&$ ... $\&vec_x^{id_x}$. The set bits (*i.e.*, 1 bits) in $vec_{B_{id}}$ give the IDs of the tuples that fall in the bucket. Moreover, it is easy to obtain the cardinality of the bucket B_{id} by counting the number of set bits (*bit-count* operation, *i.e.*, $\#$) in the resulting vector $vec_{B_{id}}$. That is, $\#vec_{B_{id}}$=$|B_{id}|$.

Example 14 (Constructing Summary Grid): Continuing with Example 13 and the grid in Figure 4.1, we obtain the members in each bucket by intersecting the corresponding bit vectors over attributes a and b. For instance, vec_{B_5} = vec_a^1 $\&$ vec_b^2 = 1100110011 $\&$ 1000100101 = 1000100001. The 1st, 5th and 10th bits in vec_{B_5} are set, indicating that B_5=$\{t_1, t_5, t_{10}\}$. ■

4.4.2 Algorithms

Summary-Based Clustering

With the summary grid, we are able to cluster much more efficiently. The key idea is to cluster the buckets in data summary and assign the tuples in the same bucket to the same cluster.

Given a set of tuples T to be clustered and the clustering attributes C=$\{c_1, \ldots, c_m\}$, we obtain the summary grid $S_{\mathcal{G}(T,C)}$ using C as the partitioning attributes. Associated with each bucket is a

Procedure

1: choose k virtual tuples as the initial cluster centroids;
2: **repeat**
3: assign each virtual tuple to its closest cluster, with weight n, as if n identical copies are assigned into the same cluster;
4: update the centroid of the clusters;
5: **until** the clusters converge;

Figure 4.2: Weighted *K-means* algorithm.

virtual point, located at the center of that bucket. We approximate the tuples in the bucket as a set of identical tuples at the virtual point, with the number of identical tuples equaling the cardinality of the bucket. Such approximation is based on the intuition that the tuples inside the same bucket are close enough to each other if the grid is fine-grained enough, so that their differences can be ignored without introducing significant impacts to the clustering results.

We apply clustering on the virtual points. The algorithms are similar to the conventional clustering algorithms, except that the algorithms must take into consideration the weights of the virtual points, where the weight of a virtual point is the cardinality of the corresponding bucket. For instance, in the weighted *K-means* algorithm (Figure 4.2), when the virtual point of a bucket with n tuples is inserted into a cluster, the centroid of the cluster is updated as if n identical points are inserted. Note that such simple weighted *K-means* extension has been used in various data mining and machine learning applications [61, 74], although the "weight" in their situation has different meaning.

With such adaptation, the algorithm continues for multiple rounds, as centroids are updated and virtual points are reassigned, until the clusters converge. At the end, the virtual points (*i.e.*, the buckets and thus the corresponding original tuples) are grouped into t clusters. The union (*bit-or* operation, *i.e.*, $|$) of the vectors for the buckets in the same cluster gives us the members in that cluster.

Example 15 (Weighted *K-means*): Continue our running example in Figure 4.1. Consider the case when we partition the 10 tuples into 2 clusters, using a and b as the clustering attributes.

Suppose at the beginning we choose $<4.5, 7.5>$ (the virtual point of B_5) and $<1.5, 1.5>$ (the virtual point of B_0) as the initial centroids of $cluster_1$ and $cluster_2$, respectively. Then the virtual points of all the buckets are inserted into their closest clusters. Suppose virtual point $<4.5, 7.5>$ with weight 3 (since there are 3 tuples in B_5) is inserted into $cluster_1$ first. Later $<7.5, 7.5>$ with weight 1 (the virtual point of B_8) is inserted into $cluster_1$. The centroid of $cluster_1$ is changed to $(5.25, 7.5)$, because $5.25 = (4.5 * 3 + 7.5)/(3 + 1), 7.5 = (7.5 * 3 + 7.5)/(3 + 1)$.

Suppose, when the clustering algorithm in Figure 4.2 ends, *i.e.*, the clusters converge, the two clusters are $cluster_1$=$\{B_5, B_8\}$ and $cluster_2$=$\{B_0, B_1, B_3\}$. The union of vec_{B_5} and vec_{B_8} thus gives us the members of $cluster_1$. That is , $vec_{cluster_1}$ = $vec_{B_5} \mid vec_{B_8}$ = 1000100001 | 0000000100 = 1000100101. Therefore $cluster_1$ contains 4 tuples, t_1, t_5, t_8, and t_{10}. The members for $cluster_2$ can be similarly obtained, as $vec_{cluster_2}$=0111011010. ■

Compared with clustering the original tuples, the summary-based clustering has clear advantages, as only one virtual point is needed for a large number of tuples in the same bucket. The number of virtual points can be much smaller than the number of original tuples. This reduction of data size saves not only the CPU cost in assigning tuples to clusters, but also more importantly the I/O cost in scanning the tuples from base tables or intermediate relations. More importantly, such a summary-based method allows us to integrate clustering and ranking seamlessly, as we shall see in Section 4.4.2.

Summary-Based Ranking

The structure of summary grid can be used in ranking as well. The essence of the idea is that we can prune most of the tuples that are bound to be outside of the top k tuples and zoom into the candidate tuples, based on the upper-bound and lower-bound scores of the tuples within each bucket. For a bucket, such bounds are derived from the corresponding ranges of the partitioning attributes on the bucket. The details are given below.

Given a set of tuples T to be ranked and the ranking function $\mathcal{F}$ over the ranking attributes R=$\{r_1, \ldots, r_n\}$, we obtain the summary grid $S_{\mathcal{G}(T,R)}$ using R as the partitioning attributes. The bit

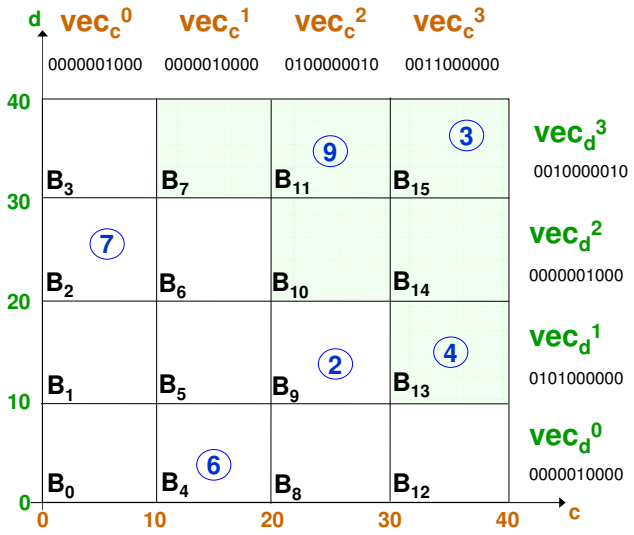


Figure 4.3: Summary-based top-k ranking.

vector for each bucket in the grid is given by intersecting the bit vectors corresponding to the ranges on the ranking attributes. The resulting vectors give us the tuple IDs in each bucket. Moreover, by counting the set bits in a vector, we obtain the cardinality of the corresponding bucket.

In addition to the cardinality, we can obtain the upper-bound and lower-upper scores for tuples in each bucket. As mentioned in Section 4.3.2, we focus on ranking functions that are monotonic with respect to the ranking attributes. Therefore, given a bucket, the highest (lowest) possible score of the tuples in that bucket is reached when the values of ranking attributes are equal to the right (left) endpoints of the corresponding ranges on these attributes. More formally, given B_{id}=<$range_1^{id_1}$,...,$range_n^{id_n}$> =<$[r_1^{id_1}, r_1^{id_1+1})$, ..., $[r_n^{id_n}, r_n^{id_n+1})$>, the upper-bound score for tuples in B_{id} is $upper_{B_{id}}$=$\mathcal{F}(r_1^{id_1+1}, \ldots, r_n^{id_n+1})$ and the lower-bound score is $lower_{B_{id}}$=$\mathcal{F}(r_1^{id_1}, \ldots, r_n^{id_n})$. That is, $\mathcal{F}[t]$=$\mathcal{F}(t.r_1, \ldots, t.r_n) \in [lower_{B_{id}}, upper_{B_{id}}), \forall t \in B_{id}$.

Example 16 (Upper- and Lower-Bounds for Buckets): Continuing with our running example, suppose we rank the 6 tuples of $cluster_2$ in Figure 4.1 and obtain the top 2 tuples, with the ranking function c+d. Figure 4.3 illustrates a summary grid for the tuples in $cluster_2$, using the ranking attributes c and d as the partitioning attributes. For the grid, $ranges_c$={$[0, 10)$, $[10, 20)$, $[20, 30)$, $[30, 40)$}, and $ranges_d$={$[0, 10)$, $[10, 20)$, $[20, 30)$, $[30, 40)$}. Thus there are 16 buckets. For in-

stance, B_{13}=<$range_c^3$, $range_d^1$> = < $[30, 40)$, $[10, 20)$>. The scores of the tuples in B_{13} are bounded by $[30 + 10, 40 + 20)$. That is, $lower_{B_{13}}$=40 and $upper_{B_{13}}$=60. Similarly we can obtain the bounds for other buckets. ■

Based on the upper-bounds and lower-bounds of the buckets, we can derive a set of candidate buckets that are guaranteed to contain *all* the top k tuples in the grid. Correspondingly the rest of the buckets can be safely pruned as the tuples in these buckets are guaranteed to be ranked lower than top k. By performing union (|) of the vectors for the candidate buckets, we can thus retrieve tuples in the candidate buckets to obtain their exact scores. The top k tuples in these candidate buckets form the top k tuples in the grid as well. The intuition is demonstrated in the following example.

Example 17 (Pruning Based on Bounds): The upper-bounds for the buckets in the non-shaded region of Figure 4.3 are at most 50. Note that the lower-bounds for buckets B_{11} and B_{15} are at least 50 and there are already 2 tuples in these buckets. Therefore we conclude that the tuples in the non-shaded region have no chance to be top 2. On the other hand, we are not able to obtain the same conclusion for any of the buckets in the shaded region, which thus constitute the candidate buckets. Therefore we union the vectors of those non-empty candidate buckets, resulting in $vec_{candidate}$=$vec_{B_{11}}$ | $vec_{B_{13}}$ | $vec_{B_{15}}$ = 0011000010. Thus the candidate tuples are t_3, t_4, and t_9. We retrieve these tuples by random I/O access and identify t_3 and t_9 as the top 2 answers in $cluster_2$. ■

The detailed algorithm for ranking is shown in Figure 4.4. Formally, we prove both the correctness and the optimality of the algorithm, *i.e.*, we prune all and only those buckets that can be pruned.

Property 6: With respect to a relation T, a ranking function $\mathcal{F}(R)$, and k, suppose the top k tuples are T_k. The set of candidate buckets $\mathcal{B}'$ obtained by the algorithm in Figure 4.4 is both *correct*: $\mathcal{B}'$ contains all the top k tuples, *i.e.*, $T_k \subseteq T_{\mathcal{B}'}$; and *optimal*: $\mathcal{B}'$ is the smallest set of buckets that contain T_k, *i.e.*, $\forall \mathcal{B}''$ *s.t.* $\exists B_i \in \mathcal{B}'$ and $B_i \notin \mathcal{B}''$, there exists an instance of T *s.t.* $T_k \nsubseteq T_{\mathcal{B}''}$. ■

Procedure

/* table: T; ranking attributes: R; ranking function: $\mathcal{F}(R)$; summary grid: $S_{\mathcal{G}(T,R)}$; candidate buckets: $\mathcal{B}'$ */

begin

1: $\mathcal{B}' \leftarrow \phi$
2: **for** each bucket $B_i \in S_{\mathcal{G}(T,R)}$ **do**
3: $\quad total \leftarrow 0$
4: $\quad$ **for** each bucket $B_j \in S_{\mathcal{G}(T,R)}$ **do**
5: $\quad\quad$ **if** $lower_{B_j} \geq upper_{B_i}$ **then**
6: $\quad\quad\quad total \leftarrow total + \#vec_{B_j}$
7: $\quad$ **if** $total < k$ **then**
8: $\quad\quad \mathcal{B}' \leftarrow \mathcal{B}' \cup \{B_i\}$ /* candidate buckets */
9: $vec_{\mathcal{B}'} \leftarrow |_{B_i \in \mathcal{B}'} vec_{B_i}$ /* union of the vectors */
10: $T_{\mathcal{B}'} \leftarrow$ retrieve tuples whose bits are set in $vec_{\mathcal{B}'}$ /* candidate tuples */
11: sort $T_{\mathcal{B}'}$ based on $\mathcal{F}(R)$
12: **return** the top k tuples in $T_{\mathcal{B}'}$

end

Figure 4.4: Summary-based top-k algorithm.

So far we have assumed that we are given the set of tuples to be ranked. Intersecting the bit vectors for the ranking attributes results in a grid over all the tuples, by using the ranking attributes as the partitioning attributes. However, in our queries, we are required to obtain the top k results in each cluster. In other words, a grid must be constructed for the tuples in each cluster. We realize this easily by intersecting the vectors for the buckets in the aforementioned grid with the bit vector for each cluster, as obtained in Section 4.4.2. An example is shown below.

Example 18 (Integrating Ranking with Clustering): Continue Example 16 and Figure 4.3. The summary grid in Figure 4.3 was obtained by assuming that the bitmap indices on ranking attributes c and d are built for the tuples in $cluster_2$ only. While in reality, we can only build bitmap indices for the whole table T, without knowing what will be the clusters for users' dynamic queries. Below is how we can obtain the summary grid in Figure 4.3 in reality.

According to Example 15, $vec_{cluster_2}$=0111011010, which gives the 6 tuples in $cluster_2$. For T, the bitmap index on c has vectors $real\_vec_c^0$, $real\_vec_c^1$, $real\_vec_c^2$, $real\_vec_c^3$. Suppose $real\_vec_c^1$ =1000010000. We intersect $real\_vec_c^1$ with $vec_{cluster_2}$ to obtain the tuples in $cluster_2$ that fall in

$range_c^1$. Thus we have $real\_vec_c^1$ & $vec_{cluster_2}$=1000010000 & 0111011010 = 0000010000, which is the vec_c^1 in Figure 4.3. We can similarly obtain all the vectors, thus obtain the summary grid in Figure 4.3. ■

Dealing with Boolean Conditions

We have assumed that we cluster the original relation T without considering Boolean conditions. However, the tuples to be clustered are actually the result of Boolean conditions, *i.e.*, $\sigma_{\mathcal{B}}(T)$ [7]. Therefore before constructing the summary grid in Figure 4.1, the vectors over the clustering attributes must take into consideration the filtering effect of the Boolean conditions. If a tuple does not belong to $\sigma_{\mathcal{B}}(T)$, the corresponding bits in the vectors must be set to 0. Bit vector operations smoothly allow such processing of clustering together with Boolean conditions, as explained below.

Selections: Suppose our query has a set of conjunctive range selection conditions $v_1^1 \leq b_1 \leq v_1^2, \ldots, v_p^1 \leq b_p \leq v_p^2$. We first obtain a vector $vec_{\mathcal{B}}$, which contains the tuples that satisfy all the selection conditions [8]. Then given the grid on T over the clustering attributes C, $\mathcal{G}(T, C)$, we intersect $vec_{\mathcal{B}}$ with each vector for the individual range on every attribute (the $vec_a^0, vec_a^1, vec_a^2, vec_b^0, vec_b^1, vec_b^2$ in Figure 4.1), to obtain the grid on $\sigma_{\mathcal{B}}(T)$, before generating virtual data points and applying the weighted clustering algorithm.

There is a vast literature on using bitmap index to answer Boolean queries, which contains the details about how to obtain $vec_{\mathcal{B}}$. Briefly, for each condition $v_i^1 \leq b_i \leq v_i^2$, given a bitmap index over b_i, we can use bitmap operations to obtain a vector $vec_{v_i^1 \leq b_i \leq v_i^2}$ (in simplified form vec_{b_i}) that contains the tuples satisfying the condition. By intersecting the vectors for all the individual conditions, we obtain $vec_{\mathcal{B}}$. The bitmap index may need to be encoded in some way so that a very small number of bitmap operations can allow us to obtain such vec_{b_i}. There are many encoding schemes in the literature(*e.g.*, [16, 97, 96]). For instance, some scheme requires only

[7] Here $\mathcal{B}$ denotes Boolean conditions, not the buckets in Section 4.4.1- 4.4.2.

[8] More strictly speaking, the corresponding bits for the satisfying tuples in T are all set in the vector.

one bitmap operation for any one-side range selection condition and some requires two for any two-side condition.

Note that even without using bitmap index to handle the selection conditions, we can construct the bit vector $vec_{\mathcal{B}}$ upon $\sigma_{\mathcal{B}}(T)$ that is obtained using any conventional query processing techniques.

Join Queries: Under the assumption of snowflake-schema made in Section 4.3.2, our technique can be easily extended to handle join queries. Such join queries are so-called "star-joins" under star-schema, a special case of snowflake-schema. Consider a simple case with only two tables, where S is the fact table and R is the dimension table, $j1$ is a key of R and $j2$ is the corresponding foreign key in S. Due to the foreign key constraint, there exists one and only one tuple in R joining with each and every tuple $s \in S$. Therefore for a join condition $R.j1$=$S.j2$, virtually all the join results are in S, with some attributes in S and some other in R. Therefore, for each attribute a in the schema of R except $j1$ (since $R.j1$=$S.j2$ and we already have $j2$ in S), we can construct a bitmap index on a for the tuples in S, even though a is not an attribute of S. In general, we can follow this way to construct bitmap index for the tuples in the single fact table, on all relevant attributes in the dimension tables. Thus the Boolean selection conditions involving these attributes can be viewed as being applied on the fact table only. A join query can then be processed like a single table query. More details about such bitmap join index are in [80].

Example 19 (Handling Boolean Conditions): Continue Example 14 and Figure 4.1. Suppose our query has a condition $10 \leq e \leq 20$. By operations on the bitmap index over e, suppose we obtain vec_e=1100101000, indicating tuples t_1, t_2, t_5, and t_7 satisfy the condition. After intersecting with the vectors on the ranges, we get $(vec_a^1)'$=vec_a^1 & vec_e=1100110011 & 1100101000=1100100000, $(vec_b^2)'$=vec_b^2 & vec_e=1000100101 & 1100101000=1000100000. Thus $(vec_{B_5})'$ = $(vec_a^1)'$ & $(vec_b^2)'$ = 1100100000 & 1000100000 = 1000100000, indicating the new bucket B_5'=$\{t_1, t_5\}$. ■

4.5 Optimization Heuristics

In this section, we present the optimization heuristics in summary-based clustering (Heuristic 1 and 2) and ranking (Heuristic 3).

Heuristic 1– Pruning Underpopulated Buckets in Grid Construction for Clustering:
For clustering on high dimensions, there are potentially huge number of buckets in the summary grid even if the number of ranges per attribute is small. However, likely many of the buckets are empty (if there exist clusters at all). During construction of the grid, we get rid of the empty intermediate buckets before the vectors from all the attributes are intersected. More generally, we prune the buckets whose cardinality is under certain threshold, *i.e.*, the underpopulated buckets that likely will result in many empty buckets if they further intersect with the remaining attributes. The pruned buckets do not participate in clustering. After clustering the non-pruned buckets in the grid, we need to use random access to retrieve the tuples belonging to the pruned buckets. (The IDs of these pruned tuples are obtained by *bit-negation* (*i.e.*, $\sim$) of the union of vectors for all the clusters.) The pruned tuples are then assigned to their closest clusters, whose vectors are modified by setting the bits corresponding to the pruned tuples.

Heuristic 2– Dynamically Selecting Partitioning Ranges:
To construct the summary grid in Section 4.4, the bitmap index on each partitioning attribute must have a set of bit vectors, one per range of the attribute values. However, we may not know in prior what is the appropriate number of ranges, *i.e.*, the number of vectors to build. On the one hand, too many ranges result in too many buckets in the grid, thus large number of bitmap intersections. On the other hand, insufficient number of buckets due to too few ranges result in poor clustering quality or pruning power, for summary-based clustering or ranking, respectively.

We address this problem by starting with large buckets (*i.e.*, small number of buckets) and splitting buckets dynamically. At the beginning, we start with 2 ranges on each attribute, resulting in 2^n buckets after intersecting the vectors from all the n attributes, among them x_2 buckets are nonempty, thus x_2 bucket vectors. If more ranges are necessary, we split each range into 2 ranges.

For each of the x_2 bucket vectors, on each attribute, we intersect the bucket vector with the 2 vectors for the smaller ranges within the original range corresponding to the bucket. We stop splitting an individual bucket if its cardinality is under certain threshold. After this step, totally we obtain x_4 vectors for the smaller nonempty buckets. We stop the whole splitting procedure when the number of nonempty buckets is over another threshold.

Such binary splitting of ranges (vectors) can be easily supported by bitmap encoding scheme such as bit-sliced index (BSI) [81]. A BSI on an attribute a consists of m+1 vectors vec_0, ..., vec_m, where the i^{th} bit of vec_j is set if the j^{th} bit is set in the binary representation of $t_i.a$. Thus these vectors together form the binary representation of the attribute values in all the tuples. Therefore vec_m and $\sim vec_m$ provide the 2 vectors for the initial 2 ranges on a. Similarly $vec_m \& vec_{m-1}$, $vec_m \& (\sim vec_{m-1})$, $(\sim vec_m) \& vec_{m-1}$, and $(\sim vec_m) \& (\sim vec_{m-1})$ give the 4 vectors when the ranges are split, and so on. We can easily adopt BSI for ranges. The idea is to partition the value domain of an attribute into a sufficiently large number (2^{m+1}) of "minimal" ranges, and number the ranges with 0, ..., 2^{m+1}-1, ordered from the range with the lowest value to the highest. We then use BSI to capture the binary representation of the numbers corresponding to the minimal ranges, thus allow various sizes of the dynamic ranges during splitting.

According to our experiments (discussed in Section 4.6), empirically a small number of ranges such as 10 is sufficient for 2 or 3 clustering attributes, 5 is sufficient for 4 to 6 attributes, and even only 3 ranges is sufficient for 8 or more attributes. Intuitively, with large number of clustering attributes, two tuples are unlikely to be in the same range on many attributes, therefore large range is sufficient to differentiate the tuples for the clustering. The grid has 3^8 buckets when there are 3 ranges on each of the 8 attributes, which can be sufficient.

Heuristic 3– Incrementally Constructing Grid for Ranking:

Directly following Section 4.4.2, we would have to fully construct the summary grid for ranking. However, there is no need for such full grid, since most of the buckets can be pruned even before they are actually constructed. Following this intuition, we construct the summary grid for ranking in lock-step fashion, similar to the *NRA* top-k algorithm [32]. Detailed algorithm is omitted.

Instead, We explain the algorithm using an example. For simplicity, suppose there are 2 ranking attributes a and b, each of which has 4 ranges, $ranges_a$ = $\{range_a^0, range_a^1, range_a^2, range_a^3\}$ = $\{[a^0, a^1), [a^1, a^2), [a^2, a^3), [a^3, a^4)\}$ and $ranges_b$ = $\{range_b^0, range_b^1, range_b^2, range_b^3\}$ = $\{[b^0, b^1), [b^1, b^2), [b^2, b^3), [b^3, b^4)\}$.

At step–1, we start by intersecting the first range from each ranking attribute, *i.e.*, $range_a^3$ and $range_b^3$, resulting in the single bucket with the highest upper-bound score in the whole grid, *i.e.*, B_{15} = $< range_a^3, range_b^3 >$. Then at each following step–i, we use the next range on every attribute (in our example $range_a^{4-i}$ and $range_b^{4-i}$) and intersect them with previous ranges. The seen ranges classify the buckets in the full grid into three types: (1) *completely seen buckets* or *csb*, whose corresponding ranges are seen on every attribute in previous steps; (2) *partially seen buckets* or *psb*, whose ranges are seen on some attributes; and (3) *unseen buckets* or *usb*, whose ranges are unseen on every attribute. The upper-bound and lower-bound scores for *csb* are computed from the corresponding ranges. For one *psb* B, on attribute a, if the corresponding range is $range_a^j$, where $j < 4 - i$ (*i.e.*, $range_a^j$ is unseen yet), we use $[a^{4-i-1}, a^{4-i})$ as the range of B on a, otherwise we use the corresponding seen range if $j \geq 4 - i$. We thus obtain the bounds for *psb*. Similarly we obtain the bounds for *usb*, as it is a special case of *psb*. At some step, if there is a subset of *csb* containing at least k tuples in total such that their lower-bounds are higher than or equal to the upper-bounds of all the *psb* and *usb*, then the top k tuples in the *csb* are the top k answers and our algorithm terminates.

4.6 Experiments

The framework and algorithms are implemented in C++. Moreover, the bitmap index implementation is based on [89], which builds multiple bitmap indices at different domain resolutions and compresses them using the WAH compression method [96]. The weighted *K-means* is built upon a publicly available implementation of *K-means* algorithm from [34]. For the straightforward approach of materialize-cluster-rank, we apply this *K-means* implementation on real tuples instead

parameter	meaning	values
s	# tuples	80K, 400K, 800K, 4M, 8M
t	# clusters	2,4,6,8,10,20,50,100
c	# clustering attributes	$[2, 8]$
p	# ranges per clustering attribute	5, 10, 20, 30, 40
k	retrieval size per cluster	1, 5, 10, 50, 100
r	# ranking attributes	$[2, 5]$
p'	# ranges per ranking attribute	10, 20

Table 4.1: Configuration parameters for experiments on integrating clustering and ranking.

of the virtual tuples from summary, and use an implementation of external merge-sort for ranking.

To verify its effectiveness, we conducted experiments to compare the proposed framework (denoted as *ClusterRank*) with the straightforward approach (denoted as *StraightFwd*), on both efficiency and quality. Moreover, we investigated how they are affected by important factors under various configurations. The detailed experimental results are presented below. Section 4.6.1 describes the settings of experiments. Regarding *efficiency*, the experimental results show that *ClusterRank* is orders of magnitude more efficient than *StraightFwd* (Section 4.6.2). Regarding *quality*, the results indicate that clustering based on the summary gird achieves close to the same quality of results as clustering on the full data does (Section 4.6.3).

4.6.1 Experimental Settings

Our experiments were conducted over a synthetic table with a set of 4-byte integer clustering attributes, a set of 4-byte floating number ranking attributes, and other attributes to pad the clustering and ranking attributes to form 100-byte per tuple. The tuple values of these two sets of attributes are independently created. The values of the ranking attributes are independently generated by various distributions, including uniform, Gaussian, and cosine distributions. The values of the clustering attributes are produced by a data generator for clustering algorithms from [34]. The generator creates values based on underlying data models, one model per cluster. A model specifies, for the corresponding cluster, the mean and standard deviation of each attribute individually.

The values on an attribute are generated by following the Gaussian distribution with the specified mean and standard deviation.

Each query used in our experiments clusters the tuples by all the clustering attributes and uses the summation of the ranking attributes as the ranking function. Note that we do not experiment with Boolean selection and join conditions. The synthetic table can be viewed as the results after such conditions are applied. To obtain the results, the approach of using bitmap index has been well-studied and is shown to be very efficient for range selections and star-joins [79, 80, 81, 16, 97]. In Section 4.4.2 we have discussed how to integrate with such techniques. To focus on the performance study of the new clustering and ranking methods proposed, we do not mix with the performance measurements on Boolean conditions, whose results are well-known in the literature.

The experiments were run on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 1MB cache), 2GB RAM, and a RAID5 array of 3 146GB SCSI disks, running Linux 2.6.15.

4.6.2 Efficiency

We evaluated the performances of *ClusterRank* and *StraightFwd* and studied how they are affected by several important configuration parameters, which are summarized in Table 4.1.

The Efficiency of Clustering:

To evaluate the performance of clustering, we conducted experiments under groups of configurations by the value combinations of the four relevant parameters, s, t, c, and p. In each group of experiments, we varied the value of one parameter and fixed the values of the remaining three. We then run *ClusterRank* and *StraightFwd*, and studied how their performances are affected as the value of the varying parameter changed. The results on wall-clock execution time under four sample groups of experiments are shown in Figure 4.5. In the figure, for *ClusterRank*, we use *CR-build* to represent the time for building summary grid and *CR-cluster* for clustering using the summary. For *StraightFwd*, we use *SF-scan* to denote the time for scanning the table and *SF-cluster* for

directly clustering the original tuples instead of using the summary.

Overall, the values of these four time measurements are in general in the order of *CR-cluster*<*CR-build*<*SF-scan*<*SF-cluster*. Due to the small number of virtual tuples in the summary grid, *CR-cluster* is several orders of magnitude smaller than others, therefore is almost negligible. While on the other hand, *SF-cluster* is orders of magnitude larger than others. *SF-scan* is normally larger than *CR-build*, with the differences often being an order of magnitude. In summary, Figure 4.5 shows that with respect to the efficiency in clustering, our approach (*CR-build* + *CR-cluster*) is much more efficient than the straightforward one (*SF-scan* + *SF-cluster*).

To understand how the parameters affect the performance, below we further analyze the individual graphs in Figure 4.5.

(a) s=4**M,** t=10**,** c=3: Varying the number of ranges per attribute in constructing the grid, 4 million tuples are partitioned into 10 clusters by 3 clustering attributes. As the number of ranges (p) increases, the less efficient grid construction (*CR-build*) is due to more bitmap operations for intersecting the ranges. Moreover, as p increases, there are also more nonempty buckets (thus more virtual tuples) in the summary grid when it is constructed, therefore *CR-cluster* takes longer. This is further verified by Figure 4.6, which shows the number of bitmap operations and the number of nonempty buckets in the grid. Figure 4.6(a)-(d) illustrate the results under four groups of configurations, individually corresponding to the configurations in Figure 4.5(a)-(d). Although the cost of *ClusterRank* increases as p increases, usually a small value of p such as 10 or 5 is sufficient for good quality of clustering results.

(b) s=4**M,** t=10**,** p=5: In this configuration c is changing and other parameters are fixed. We can see that the effect of c is similar to that of p, as its increasing results in more bitmap intersections and more nonempty buckets (*cf.* Figure 4.6(b)), thus more expensive *CR-build* and *CR-cluster*. *ClusterRank* enjoys clear advantages over *StraightFwd* until the number of clustering attributes goes beyond 8, which we believe is sufficiently large in our target applications.

(c) s=4**M,** c=8**,** p=5: As expected, the more clusters to produce, the less efficient the clustering is, thus longer execution time for both *CR-cluster* and *SF-cluster*.

(d) t=10**,** c=3**,** p=10: As expected, increasing the number of tuples increases the cost of everything.

The Efficiency of Ranking:

We conducted experiments under configurations of the four relevant parameters, s, t, r, and k. Similar to the experiments in clustering efficiency, in each group of experiments, we changed the value of one parameter and fixed the remaining ones. Note that the number of ranges per ranking attribute (p') with value 20 works quite well in general in all the configurations. Therefore we do not present the results with respect to various p' values. The wall-clock execution time under four sample groups of experiments is shown in Figure 4.8, where we use *CR-rank* to denote the time for grid-based ranking in *ClusterRank* and *SF-sort* for the sorting in *StraightFwd*.

Overall, *SF-sort* is one order of magnitude more expensive than *CR-rank*. We further analyze the individual graphs. Figure 4.8(a) shows that *CR-rank* only increases slowly as k increases, thus *ClusterRank* is effective for sufficiently large retrieval size within each cluster; Figure 4.8(b) shows that *CR-rank* increases as the number of ranking attributes (r) increases, and becomes close to *SF-sort* when r=5. However, as commonly acknowledged in the literature of ranking queries (*e.g.*, [32, 13, 30, 19, 4, 57, 21, 67, 55]), in many cases a very small number of ranking attributes suffice. We believe this is especially true in our motivating applications, where users are not expected to articulate too complicated ranking criteria involving more than 5 attributes; Figure 4.8(c) indicates that *CR-rank* increases as the number of clusters (t) increases. This is because *ClusterRank* constructs a grid for each cluster and performs ranking within each grid. Although *SF-sort* is not affected by t, it may be smaller than *CR-rank* only when there are a very large number of clusters. We argue the number of clusters t is small in our target applications, because clustering is used to organize large query results for users and such large t is not helpful and thus unnecessary. Finally, Figure 4.8(d) shows that both *CR-rank* and *SF-sort* increase as the number of tuples increases, as expected.

The Overall Efficiency:

We compared *StraightFwd* and *ClusterRank* with the execution time for clustering and ranking

combined. Consider both Figure 4.5 and 4.8, we see that *SF-sort* in general is close to *SF-cluster*. Thus these two costs together dominate the execution time of *StraightFwd*, which is orders of magnitude more than the time of *ClusterRank*. In Figure 4.9 we show such comparisons under two sample configurations. In Figure 4.9(a) we vary the number of clusters and fixed the values of others, while in Figure 4.9(b) we vary the number of tuples.

4.6.3 Quality

We compared res_{CR}, the clustering results from the weighted *K-means* on the summary grid (*ClusterRank*), with res_{SF}, the results from the conventional *K-means* on the original tuples (*StraightFwd*). We measured how close res_{CR} is to the ground truth res_{SF}, *i.e.*, $close(res_{SF}, res_{CR})$. This metric is defined below.

Suppose there are two methods that generate two different sets of clusters res= $\{c_1, \ldots, c_t\}$ and res'=$\{c'_1, \ldots, c'_t\}$, respectively, where each c_i and c'_j is a set of tuples. The *closeness* of res' to the ground-truth res is

$$close(res, res') = \frac{\sum_i (|c_i| \times max_j(sim(c_i, c'_j)))}{\sum_i |c_i|},$$

where

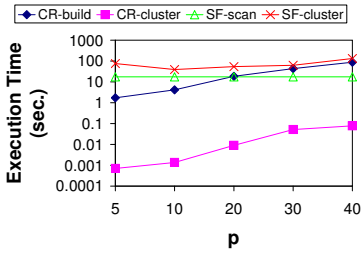
$$sim(c_i, c'_j) = 2\frac{|c_i \cap c'_j|}{|c_i| + |c'_j|}.$$

This metric is asymmetric. Since $close(res, res')$ measures how well the clusters in res are captured by the clusters in res', it should be used when res instead of res' is the ground-truth. Its value range is $[0, 1]$, as 1 indicates identical results and 0 indicates totally different results. The metric sim has been used in comparing clustering results, *e.g.*, in [63] and [36]. It is equivalent to the *F-measure* for *precision/recall* in IR literature.

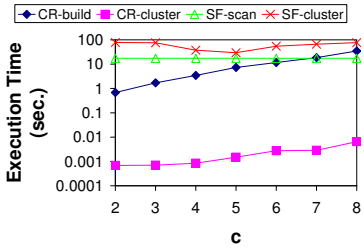
K-means algorithm is known to be unstable and its behavior depends on the initial centroids chosen [60]. Even running *K-means* twice on the same data may not give us very high closeness

between the two results. Therefore instead of interpreting the value of $close(res_{SF}, res_{CR})$ directly, we compare it with $close(res_{SF}, res_{SF})$, which is the average closeness among the results from multiple runs of *StraightFwd*. If $close(res_{SF}, res_{CR})$ is close to $close(res_{SF}, res_{SF})$, we are confident that the quality of the results from *ClusterRank* is comparable to that from *StraightFwd*.

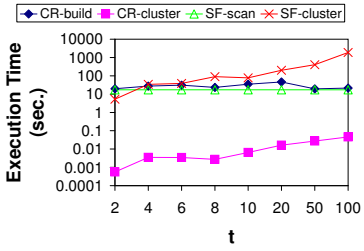
Figure 4.7(a)-(d) show $close(res_{SF}, res_{CR})$ and $close(res_{SF}, res_{SF})$ under four groups of configurations, corresponding to the configurations in Figure 4.5(a)-(d) and Figure 4.6(a)-(d). The figures show that the quality of clustering results from *ClusterRank* is often close to the quality from *StraightFwd*. The quality increases as the number of ranges per clustering attribute (p) increases (Figure 4.7(a)), because the summary grid becomes more and more fine-grained. However, we observe that a relatively small p such as 5 and 10 usually is sufficient. As Figure 4.7(b) shows, for the same p, the more attributes, the higher quality. This is simply because it is easier to partition data when they have more dimensions to compare with each other. Therefore with 8 clustering attributes, p=5 is sufficient under various number of clusters requested (Figure 4.7(c)), and with 3 clustering attributes, p=10 is fairly sufficient (Figure 4.7(d)). Moreover, Figure 4.7(c) verifies that it is more difficult to perform clustering when more clusters are quested.



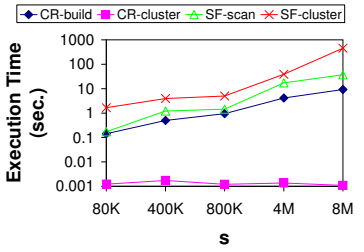
(a) s=4M, t=10, c=3



(b) s=4M, t=10, p=5

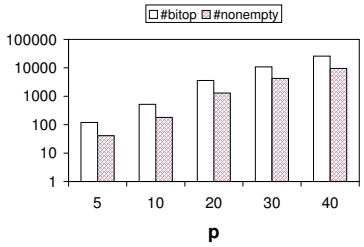


(c) s=4M, c=8, p=5

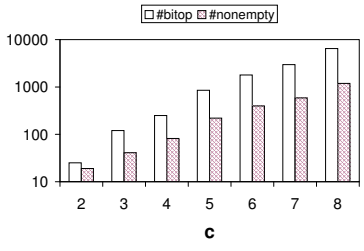


(d) t=10, c=3, p=10

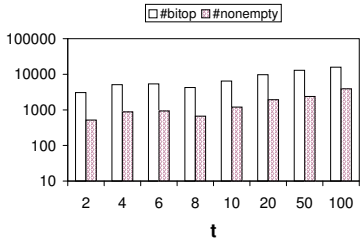
Figure 4.5: Clustering efficiency.



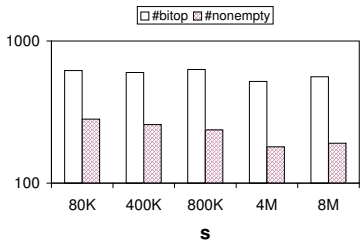
(a) s=4M, t=10, c=3



(b) s=4M, t=10, p=5

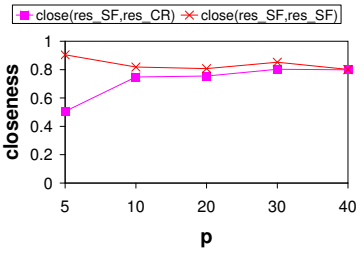


(c) s=4M, c=8, p=5

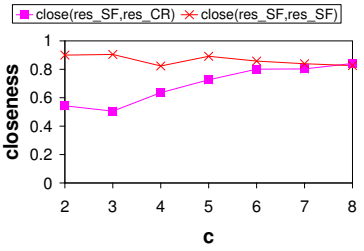


(d) t=10, c=3, p=10

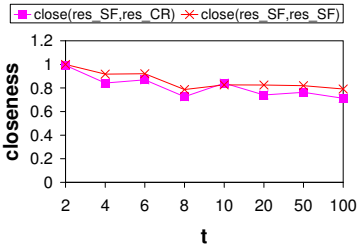
Figure 4.6: Number of bitmap operations and nonempty buckets.



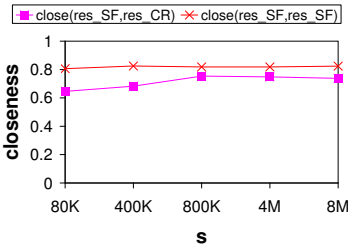
(a) s=4M, t=10, c=3



(b) s=4M, t=10, p=5

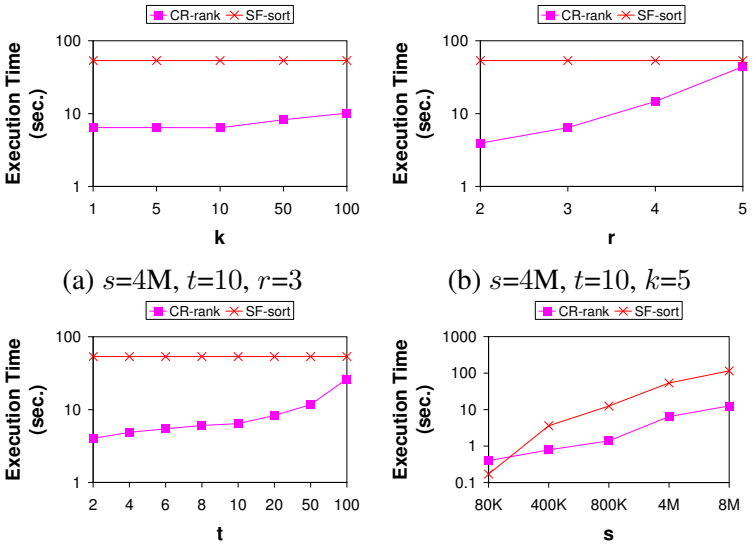


(c) s=4M, c=8, p=5



(d) t=10, c=3, p=10

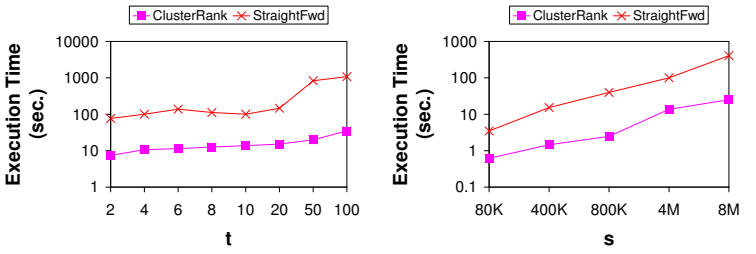
Figure 4.7: Clustering quality.



(a) s=4M, t=10, r=3 (b) s=4M, t=10, k=5

(c) s=4M, r=3, k=5 (d) t=10, r=3, k=5

Figure 4.8: Ranking efficiency.



(a) s=4M, c=5, p=5, r=3, k=5 (b) t=10, c=5, p=5, r=3, k=5

Figure 4.9: Total execution time.

Chapter 5

Beyond Ranking: Inverse Ranking Queries

We identify a novel and interesting type of *inverse ranking queries*, symmetrical to ranking. Such a query obtains the rank of a data record within a certain context given in the query. Inverse ranking queries are useful in many places. For instance, a credit card company may be interested in the standing of a new customer among her peers, in order to determine her credit line. As another example, we may want to compare a newborn baby to others with respect to their heights, weights, *etc.* While ranking has recently gained significant attention from the research community, inverse ranking query has not been studied so far, in contrast to its usefulness. This thesis thus proposes and defines the query model and SQL language extension to enable expressing such queries.

With the current query processing techniques, a straightforward *exhaustive* approach of processing inverse ranking queries is to fully materialize the results of a Boolean query, *i.e.*, the context of the ranking, and then count the number of tuples whose ranking scores are higher than the score of the object in question. The rank of the given object is thus obtained as a post-processing step. Such an exhaustive approach can be inefficient, as the query only asks for the rank of a certain tuple, while the full Boolean results are indeed made. Observing the symmetry between inverse ranking queries and top-k queries, it may appear that the recently developed top-k query algorithms can be adopted. However, top-k algorithms are explicitly optimized for retrieving very small number (k) of top answers. As k increases, the performance of these algorithms degrades and eventually becomes even worse than the straightforward materialize-then-sort approach (Chapter 2). Unfortunately, the tuple in question in an inverse ranking query can be ranked anywhere such as in the middle or at the bottom, corresponding to fairly large k. Therefore we must look for novel and efficient processing techniques.

We design a *partition-and-prune* framework for processing inverse ranking queries. The framework starts by partitioning the space of tuples into buckets. The upper and lower bounds of ranking scores for tuples within each bucket are derived. These bounds determine which are the candidate buckets whose tuples rank near the given tuple in the query. After computing their cardinalities (number of tuples), the non-candidate buckets can be safely pruned because, for any such bucket, its tuples are all ranked higher (lower) than the query tuple. Therefore we only need to retrieve the tuples in the candidate buckets in order to obtain the ranking position of the query tuple. To realize this general framework, we introduce several partition schemes and implementation methods. Some of these methods exploit common data structures in database systems, while some others utilize bitmap index built over ranking functions. Our experimental study shows that our algorithms can be significantly more efficient than the exhaustive method.

While inverse ranking query is compelling by itself, we find that its dual form, *quantile query*, is also important. A quantile query returns the results at certain ranking positions according to a ranking function. More specifically, the q quantile of a given set of data is the value v_q such that the fraction q of the data are higher than v_q. Quantile queries can be useful in non-traditional data retrieval and exploration since they provide a mechanism of fast-forwarding over the data, as an analogy to such functionalities in video and audio equipments. They locate the quantiles as a sketch of the query results, which give users a quick feeling about both the data and the ranking function, and thus help in continuously exploring the data. Moreover, quantile points have significant statistical meanings. For example, the 50% quantile is the mean of a dataset; the *quantile-quantile* (*q-q*) plot is a graphical method used in statistics to compare the distributions of two sets of data and determine if they come from populations with a common distribution.

This is also the first that studies such quantile queries in the general context of querying databases. Previous works on computing quantiles [3, 5, 37, 71, 6, 28, 27, 101] focus on the quantiles for a set of data values, whereas we study the quantiles among database records that are ranked by functions combining multiple criteria (attributes). More importantly, the context of ranking in this study is general database queries with Boolean conditions (selections and joins),

which are not considered by previous works.

The duality between inverse ranking queries and quantile queries allows us to apply the same framework and similar algorithms in processing them. However, we focus on inverse ranking queries, and do not discuss such adaptation.

In summary, this work makes the following contributions:

- **Concept: inverse ranking queries and quantile queries.** We identify the query models and propose the SQL extensions for defining these queries. To the best of our knowledge, ours is the first in the literature to study inverse ranking queries, and the first to investigate quantile queries in general database query context.
- **Framework: partition-and-prune approach.** We design this general framework to avoid costly exhaustive approach. We also develop a preliminary cost model to assist the analytical comparisons of various implementation methods for the framework.
- **Implementations.** We develop several methods to instantiate the framework, utilizing existing and new data structures. We analyze the pros and cons of these methods, based on the cost model. Our empirical study verifies that some of our methods can be significantly more efficient than the exhaustive approach.

The rest of the chapter is organized as follows. In Section 5.1, we introduce the SQL extension to define inverse ranking and quantile queries and illustrate the example queries. Section 5.2 presents the partition-and-prune framework, focusing on inverse ranking queries. Several implementation methods for the framework are introduced in Section 5.3. We experimentally evaluate the proposed framework and algorithms in Section 5.4.

5.1 Defining Inverse Ranking Queries and Quantile Queries

In this section, we propose extensions to SQL language for expressing inverse ranking and quantile queries, and use examples to motivate their applications.

To specify an inverse ranking query, we overload the OLAP function *rank*() in SQL[1], as shown below:

select	..., *rank*() **in** (	**select**	...
		from	R_1, ..., R_n
		where	$\mathcal{B}(c_1, \ldots, c_j)$)
from	R'_1, ..., R'_h		
where	$\mathcal{B}'(c'_1, \ldots, c'_l)$		
order by	$\mathcal{F}(p_1, \ldots, p_m)$		

In an inverse ranking query Q, the product of the base relations $R'_1 \times \ldots \times R'_h$, filtered by a *Boolean function* $\mathcal{B}'(c'_1, \ldots, c'_l)$ (*e.g.*, $\mathcal{B}'=c'_1 \wedge c'_2 \wedge c'_3$), constitutes the *query tuples*. Following *rank*() **in**, another Boolean function $\mathcal{B}(c_1, \ldots, c_j)$, applied over $R_1 \times \ldots \times R_n$, supplies the *context tuples*. A *ranking function* $\mathcal{F}$ over the *ranking attributes* p_1, ..., p_m (*e.g.*, $\mathcal{F}=p_1+p_2+p_3$) computes the *ranking scores* of context and query tuples. The query returns the ranks of the query tuples among the context tuples, by the descending order of their scores[2]. Note that in order to make the ranking function applicable, the schema of both context and query tuples should contain the ranking attributes.

Formally, Q returns query tuples $R_{\mathcal{B}'}=\sigma_{\mathcal{B}'(c'_1,\ldots,c'_l)}\ (R'_1 \times \cdots \times\ R'_h)$ with their ranks, determined as follows. Each tuple t has a *ranking score* $\mathcal{F}[t]$. For a tuple $t_q \in R_{\mathcal{B}'}$, its rank is the number of context tuples t_c that have higher scores than t_q (plus 1), *i.e.*, *rank*(t_q)=$1+|\{t_c|\mathcal{F}[t_c]>\mathcal{F}[t_q], t_c \in R_{\mathcal{B}}\}|$, where $R_{\mathcal{B}}=\sigma_{\mathcal{B}(c_1,\ldots,c_j)}(R_1 \times \cdots \times R_n)$ is the *context relation*. When there are ties in scores, an arbitrary *deterministic* "tie-breaker" function can be used to determine an order, *e.g.*, by tuple IDs.

The aforementioned query obtains query tuples by Boolean selection and join conditions, thus it requires the existence of query tuples in the Boolean results. However, we may be interested in the ranks of virtual tuples that do not necessarily exist. Therefore we propose the following alternative syntax that requests the rank of a virtual tuple $p_1 = v_1, \ldots, p_m = v_m$.

[1] OLAP functions, introduced in SQL99 and supported by major DBMSs, are for different purpose than inverse ranking.

[2] We assume **order by asc**|**desc** uses descending order (**desc**) as default, although **asc** is the default in some systems.

select	..., *rank*() **in** (	**select**	...
		from	R_1, ..., R_n
		where	$\mathcal{B}(c_1, \ldots, c_j)$)
values	$(p_1 = v_1, \ldots, p_m = v_m)$		
order by	$\mathcal{F}(p_1, \ldots, p_m)$		

Example 20: Consider a credit card company. To decide whether to approve the increase of credit line upon the request from a customer (with customer id 1001), the following query gives the rank of the customer among the customers in the same area. Various ways can be explored in computing the ranking scores. For example, a weighted average of the customer's income, age, and credit history is used in the query.

select	cid, *rank*() **in** (	**select**	*
		from	$Customer$
		where	$zipcode$=12345)
from	$Customer$		
where	cid=1001		
order by	$w_1 \times income + w_2 \times age + w_3 \times credit$		

Alternatively, we might want to use the following query to determine the rank of a user who is not an existing customer.

select	cid, *rank*() **in** (	**select**	*
		from	$Customer$
		where	$zipcode$=12345)
values	($income$=50000, age=30, $credit$=600)		
order by	$w_1 \times income + w_2 \times age + w_3 \times credit$		

■

To specify a quantile query, we extend the syntax of SQL language by adding a **quantiles at** clause, as shown below:

select	...
from	R_1, ..., R_n
where	$\mathcal{B}(c_1, \ldots, c_j)$
order by	$\mathcal{F}(p_1, \ldots, p_m)$
quantiles at	q_1, ..., q_k

The semantics of such a query Q is that, among the filtered Boolean results $R_{\mathcal{B}}$, ranked by $\mathcal{F}(p_1, \ldots, p_m)$, those tuples ranked at the given quantile positions $q_1, \ldots, q_k$ are returned. Formally,

Q returns k tuples[3] $\{t_1, \ldots, t_k\}$ from $R_{\mathcal{B}}$ at the quantiles $q_1, \ldots, q_k$, such that $rank(t_i)$=$\lceil q_i \times |R_{\mathcal{B}}| \rceil$ (for $0 \leq q_i \leq 1$, representing percentile) or $rank(t_i)$=$\lceil q_i \rceil$ (for $q_i > 1$, representing absolute ranking position).

Example 21: Consider a real estate company that is interested in identifying their representative houses at several ranking positions, according to the perspective of a certain type of customers. The houses are ranked by size, price, and so on. The quantile query is shown below.

select	$*$, $size/price$ **as** $score$
from	$Houses$
where	$zipcode$=12345
order by	$score$
quantiles at	$0.1, 0.5, 0.7, 0.95$

■

5.2 Partition-and-Prune Framework

In this section, we present a general framework for processing inverse ranking queries. To answer such queries, with the ranking scores of the given query tuples, we must locate where the query tuples stand among the context tuples. As discussed at the beginning of Chapter 5, the problems with the straightforward exhaustive approach is that it fully materializes the context tuples. Therefore the key of a more efficient solution lies in avoiding such full materialization. To be more specific, we want to prune irrelevant context tuples and quickly zoom into the regions containing tuples with scores close to the query tuples.

The framework is simple and intuitive. We partition the space of tuples into buckets and compute the upper-bound and lower-bound of ranking scores of the tuples within each bucket. These bounds classify the buckets into three categories, with respect a given query tuple. The buckets with lower-bounds higher than the score of the query tuple contain context tuples ranked higher than the query tuple; the buckets with upper-bounds lower than the query tuple score contain lower-ranked context tuples; and the context tuples in the rest of the buckets, the *candidate tuples*, may

[3] More rigorously, it returns $min(k, |R_{\mathcal{B}}|)$ tuples.

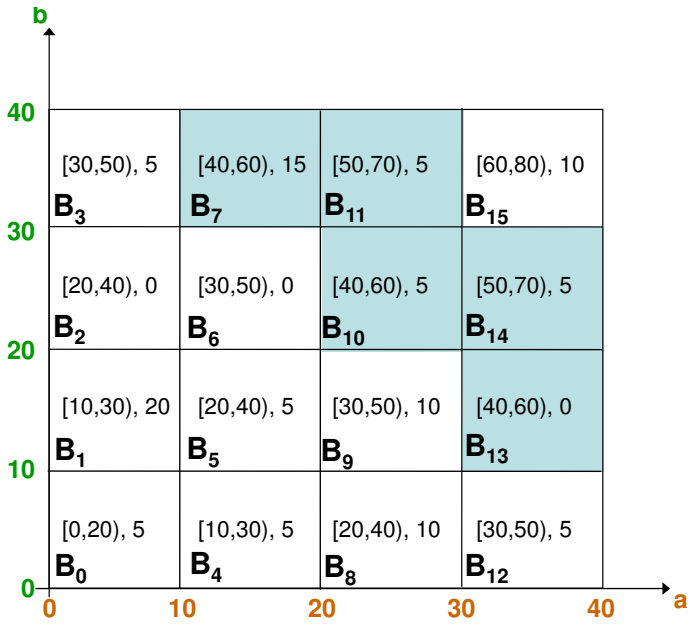


Figure 5.1: Buckets in a 2-dimensional tuple space.

be ranked higher or lower than the query tuple. Therefore by counting the cardinalities (number of tuples) of the buckets, we know how many context tuples are guaranteed to be ranked higher (lower) than the query tuple, and we only need to look up the scores of the candidate tuples to obtain the rank of the query tuple. We illustrate the idea in Example 22, as our running example.

Example 22: Consider the following inverse ranking query, which asks for the ranks of those tuples in R with a=35 and b=20, ranked by a+b.

select *, rank() **in** (**select** * **from** R)
from R
where a=35 **and** b=20
order by a+b

Figure 5.1 shows the two-dimensional tuple space (on attributes a and b) partitioned into 16 buckets. The ranges of a and b for each bucket are shown. For instance, the bucket in the left lowermost corner (B_0) has ranges $0 \leq a < 10$ and $0 \leq b < 10$. The ranges on a and b determine the upper-bound and lower-bound of tuple scores for buckets. We show the bounds and the cardinality inside each bucket. For instance, B_0 contains 5 tuples, which have the lower-bound and upper-bound score 0 and 20, respectively.

Among these buckets, the shaded ones are candidate buckets and others are pruned buckets. Bucket B_{15} is pruned because the scores of tuples inside it are at least 60, thus higher than the score of the query tuples, 35+20=55. The 10 left lower buckets are also pruned because their tuple scores are lower than 55. The tuples in the candidate buckets may score higher or lower than 55. There are 10 tuples in B_{15} and totally 30 tuples in the candidate buckets. Therefore the ranks of the query tuples are between 11 and 40. To get their ranks, we must obtain the tuples in the candidate buckets and resolve their orders to the query tuples. ■

Based on the intuition from the above example, below we formally present the framework. Section 5.2.1 defines the partitioning and pruning of tuple space, which are the basis of the procedural general algorithm in Section 5.2.2. Section 5.2.3 discusses the cost model for the algorithm. The cost model helps us in designing and analyzing various schemes of partitioning in Section 5.2.4, and is the guideline in realizing the partitioning schemes and the algorithm, in Section 5.3.

Throughout the discussion, we assume there is only one single table, without Boolean conditions over context tuples. In other words, we assume the context relation $R_{\mathcal{B}}$ is simply one base table. In Section 5.3, we discuss how to extend the techniques to handle join and selection conditions. Moreover, we will not discuss how to obtain the query tuples $R_{\mathcal{B}'}$. Note that such tuples are given when the query is about virtual tuples, using **values** (proposed in Section 5.1). In other cases, we need to obtain the query tuples and cannot avoid the overhead. Certainly there are situations when the cost of obtaining the query tuples themselves dominates that of getting their ranks.

5.2.1 Tuple Space Partitioning and Pruning

Definition 2 (Partition, Bucket, Constraint): A relation, $R(a_1, a_2, \ldots)$=$\{t_1, t_2, \ldots\}$, is a set of tuples $\{t_1, t_2, \ldots\}$ with the schema A=$\{a_1, a_2, \ldots\}$. A *partition* $\mathcal{P}_{\mathcal{R}}$=$\{b_1, b_2, \ldots\}$ is a set of subsets of R such that $\cup b_i$=R, $b_i \neq \phi$, and $b_i \cap b_j$=ϕ, $\forall b_i, b_j \in P_R$. Each subset b_i is a *bucket*. Given a bucket b_i, its cardinality $|b_i|$ is the number of tuples belonging to b_i. That is, b_i=$\{t_{i_1}, \ldots, t_{i_{|b_i|}}\}$, where $t_{i_j} \in R$, $\forall 1 \leq j \leq |b_i|$. Each bucket b_i is associated with a set of constraints C_i=$\{c_{i_1}, c_{i_2}, \ldots\}$.

Each constraint is of the form $l \leq g(A) < u$, where $g(A)$ is a function over A. Given any tuple $t_{i_j} \in b_i$, all the constraints associated with b_i are satisfied. ■

Example 23: Continue Example 22. Figure 5.1 is a partition of R, containing 16 buckets. The cardinalities of the buckets are shown in Figure 5.1. Every bucket is associated with two constraints. For instance, bucket B_0 has constraints $\{0 \leq a < 10,\ 0 \leq b < 10\}$. All the tuples in B_0 thus have both attribute a and b in the range $[0, 10)$. Note that the constraints associated with these buckets are in the form of very simple function– a single attribute. ■

Definition 3 (Upper-Bound, Lower-Bound): Given a bucket b, an upper-bound score $\lceil b \rceil$ is a value that is larger than the highest score among tuples in b. That is, $\lceil b \rceil > \mathcal{F}[t], \forall t \in b$. Similarly, a lower-bound score $\lfloor b \rfloor$ is a value that is smaller than or equal to the lowest score among tuples in b. That is, $\lfloor b \rfloor \leq \mathcal{F}[t], \forall t \in b$.[4] [5] ■

Definition 4 (Pruned and Candidate buckets): With respect to a query tuple t_q, a bucket b is a *pruned bucket* if $\mathcal{F}[t_q]$, the score of t_q, is an upper-bound of b, or if $\mathcal{F}[t_q]$ is a lower-bound of b and there is no tuple in b with score equal to $\mathcal{F}[t_q]$. Formally, given a partition $\mathcal{P}_\mathcal{R}$, the set of pruned buckets with respect to t_q is $pruned(\mathcal{P}_\mathcal{R}, t_q) = pruned^+(\mathcal{P}_\mathcal{R}, t_q) \cup pruned^-(\mathcal{P}_\mathcal{R}, t_q)$, where $pruned^+(\mathcal{P}_\mathcal{R}, t_q) = \{b | b \in \mathcal{P}_\mathcal{R}$ and $\mathcal{F}[t] > \mathcal{F}[t_q], \forall t \in b\}$ are the *dominating buckets* of t_q and $pruned^-(\mathcal{P}_\mathcal{R}, t_q) = \{b | b \in \mathcal{P}_\mathcal{R}$ and $\mathcal{F}[t] < \mathcal{F}[t_q], \forall t \in b\}$ are the *dominated buckets* of t_q. The set of candidate buckets is $candidate(\mathcal{P}_\mathcal{R}, t_q) = \mathcal{P}_\mathcal{R} - pruned(\mathcal{P}_\mathcal{R}, t_q)$. ■

Example 24: Continue Example 23. The bucket B_0 has constraints $\{0 \leq a < 10,\ 0 \leq b < 10\}$. Thus the tuples in B_0 can score at most 10+10=20 (without equality), and as low as 0, *i.e.*, $\lceil B_0 \rceil$=20 and $\lfloor B_0 \rfloor$=0.[6] Similarly, we can obtain the bounds of other buckets. The query tuple has score 55, therefore the white buckets in Figure 5.1 are pruned buckets and the shaded buckets are candidate buckets, based on Definition 4. ■

[4] Without loss of generality, we require $\lceil b \rceil$ to be open-ended while $\lfloor b \rfloor$ to be close-ended. Correspondingly the constraints in Definition 2 are left-end closed and right-end open.

[5] Note that there is an infinite number of upper-bounds and lower-bounds for any bucket.

[6] More strictly, $\lceil B_0 \rceil$=20 means 20 is one known upper-bound for b. Similar statement applies for $\lfloor B_0 \rfloor$.

Procedure Partition-and-Prune Inverse Ranking
/* relation: R, with schem A; partition: $\mathcal{P}_\mathcal{R}$; ranking function: $\mathcal{F}(A)$; query tuple: t_q */
begin
1: /* *1. Partitioning Space* */
2: determine the number of buckets, n
3: **for** each bucket b_i **do**
4: determine constraints C_i=$\{c_{i_1}, c_{i_2}, \ldots\}$, where c_{i_j} is: $l_{i_j} \leq g_i(A) \leq u_{i_j}$
5: /* *2. Deriving Bounds* */
6: **for** each bucket b_i **do**
7: /* solve the following optimization problem */
8: $\lceil b_i \rceil \leftarrow$ the value maximizes $\mathcal{F}(A)$ in b_i
9: $\lfloor b_i \rfloor \leftarrow$ the value minimizes $\mathcal{F}(A)$ in b_i
10: /* *3. Computing Cardinalities* */
11: **for** each bucket b_i **do**
12: $|b_i| \leftarrow$ compute the number of tuples in b_i
13: /* *4. Classifying Buckets* */
14: $pruned^-(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* dominated buckets */
15: $pruned^+(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* dominating buckets */
16: $candidate(\mathcal{P}_\mathcal{R}, t_q) \leftarrow \phi$ /* candidate buckets */
17: **for** each bucket b_i **do**
18: **if** $\lceil b_i \rceil \leq \mathcal{F}[t_q]$ **then**
19: $pruned^-(\mathcal{P}_\mathcal{R}, t_q) \leftarrow pruned^-(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
20: **else if** $\lfloor b_i \rfloor > \mathcal{F}[t_q]$ **then**
21: $pruned^+(\mathcal{P}_\mathcal{R}, t_q) \leftarrow pruned^+(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
22: **else**
23: $candidate(\mathcal{P}_\mathcal{R}, t_q) \leftarrow candidate(\mathcal{P}_\mathcal{R}, t_q) \cup b_i$
24: /* *5. Retrieving Candidates* */
25: $R' \leftarrow \cup_{b_i \in candidate(\mathcal{P}_\mathcal{R}, t_q)} b_i$ /* candidate tuples */
26: retrieve tuples in R'
27: $rank_{R'}(t_q) \leftarrow$ the rank of t_q in R'
28: $rank(t_q) \leftarrow rank_{R'}(t_q) + \sum_{b \in pruned^+(\mathcal{P}_\mathcal{R}, t_q)} |b|$
29: **return** t
end

Figure 5.2: The outline of the algorithm.

In determining the rank of the query tuple, we can safely prune the pruned buckets and we only need to look up the tuples in the candidate buckets. The idea is already illustrated in Example 22. More formally, we have the following Property 7. The straightforward proof is omitted.

Property 7 (Bucket Pruning Property): Given a relation R and its partition $\mathcal{P}_\mathcal{R}$, the rank of a query tuple t_q is

$rank(t_q)$=$1 + \sum_{b \in pruned^+(\mathcal{P}_\mathcal{R}, t_q)} |b| + |\{t_c | \mathcal{F}[t_c] > \mathcal{F}[t_q] \wedge t_c \in b \; s.t. \; b \in candidate\ (\mathcal{P}_\mathcal{R}, t_q)\}|$. ■

5.2.2 General Algorithm

Based on Property 7, we design the general algorithm for answering inverse ranking queries, as outlined in Figure 5.2. The algorithm takes the following steps in sequence:

1. Partitioning Space: The algorithm partitions the tuple space into buckets. It needs to decide the number of buckets and the constraints associated with each bucket. The constraints directly determine the geometrical shape and area of a bucket.

2. Deriving Bounds: The algorithm derives the upper-bound and lower-bound of every bucket, based on the associated constraints. For a set of general constraints, deriving the bounds of a ranking function is a *nonlinear programming* (NLP) problem. While general NLP problem is very hard, there are methods for special cases [9].

We concentrate on *monotonic* ranking functions, as commonly studied in top-k queries (*e.g.*, [32]). Examples of such functions include *sum*, *weighted average*, *Lp-norm distance* such as *Manhattan* and *Euclidean distance*, and so on. With single-attribute constraints in the form of $l \leq a < u$, the bounds of such monotonic ranking function can be straightforwardly determined by the ranges on the attributes in the bucket. Therefore given a partitioning scheme using only single-attribute constraints, the algorithm can handle any monotonic ranking functions.

More specifically, deriving the bounds becomes a *linearly constrained optimization* problem when all the constraints are linear functions over the attributes, and further a *linear programming* (LP) problem [10] when the ranking function is a linear function as well. There are well-studied algorithms for solving LP problems, such as the Simplex method [10]. Therefore given a partitioning scheme using linear constraints, the algorithm can process linear ranking functions, a subset of monotonic functions. [7] [8]

3. Computing Cardinalities: The algorithm computes the cardinality of every bucket according to the associated constraints.

[7] Note that the above single-attribute constraint is an extreme case of linear constraint.

[8] Linear function such as $2p_1 - 3p_2$ is monotonic on p_1 and $-p_2$.

4. Classifying Buckets: The bounds from step 2 and cardinalities from step 3 are used to identify the pruned and candidate buckets, following Definition 4.

5. Retrieving Candidates: The algorithm retrieves the tuples in the candidate buckets, evaluates their scores, and obtains the ranks of the query tuples.

Among the five steps, step 2 and 4 are described above and shown in Figure 5.2. They are not further discussed. Step 1 is the basis of the algorithm, since the partitioning scheme determines what type of ranking functions can be handled and which implementation methods for other steps are applicable. An appropriate partitioning scheme is thus key to the efficiency of the algorithm. In Section 5.2.3 we describe an analytical cost model of the algorithm. Guided by the analysis, we discuss several partitioning schemes in Section 5.2.4. There are different methods in implementing the partitioning scheme and thus the step 3 and 5. Such choices directly affect the efficiency of the algorithm. We present implementation details in Section 5.3.

5.2.3 Cost Model

Given the variety of implementations for the above algorithm steps, there exists an optimization space in choosing an efficient evaluation method for inverse ranking queries. The primitive cost model in this section is for the purpose of analyzing and comparing the choices in realizing our framework. In query optimization, a cost model is critical for estimating the costs of query plans. However, a complete cost model for inverse ranking queries, incorporated into query optimizer, is out of our focus.

The cost model has the following components:

Cost factors: The cost of our algorithm is determined by several factors, which are the partition, the data distribution, the data size, and the query.

Cost parameters: The cost is a function of several important parameters, including the number of buckets ($|\mathcal{P}_\mathcal{R}|$), the score bounds of the buckets ($\lfloor b \rfloor$ and $\lceil b \rceil$), the cardinalities of the buckets ($|b|$), and the candidate buckets ($candidate(\mathcal{P}_\mathcal{R}, t_q)$). These cost parameters are determined by the

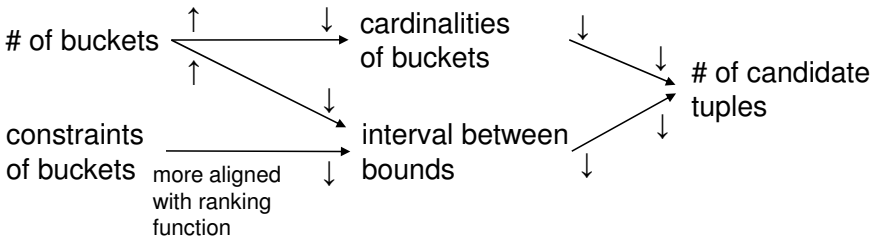


Figure 5.3: The relationship among cost parameters.

cost factors above. Specifically, the partition determines the number of buckets and the bounds (constraints) of the buckets, the data distribution and data size determine the cardinalities, and the query determines the candidate buckets, together with the partition, data distribution and size.

Cost formula: The cost formula in terms of time, as shown below, is the sum of the CPU cost, the I/O cost for obtaining cardinalities (step 3 in Section 5.2.2) and the I/O cost for retrieving candidate tuples (step 5).

$$\mathcal{C} = \mathcal{C}_{CPU} + \mathcal{C}3_{I/O} + \mathcal{C}5_{I/O} \tag{5.1}$$

Conventional query algorithms are I/O-bound rather than CPU-bound, therefore the common practice in investigating the costs of query plans is to focus on disk I/O cost. However, we shall see that some of the methods in Section 5.3 involve CPU costs that cannot be ignored. Among the five steps in Figure 5.2, step 1, 2, and 4 do not involve disk I/O. To obtain the cardinalities of buckets in step 3, some of our methods require I/O access to auxiliary data structures. For step 5, the retrieving of candidate tuples involves disk access.

Nevertheless, in most cases, the most significant component in the above formula is the disk I/O cost of step 5, for which a good metric is the number of candidate tuples. That is,

$$\mathcal{C}5_{I/O} = f_{retrieve} \times \sum_{b \in candidate(\mathcal{P}_{\mathcal{R}}, t_q)} |b| \tag{5.2}$$

where $f_{retrieve}$ is a factor. In principle the more candidate tuples, the higher cost, although the exact $f_{retrieve}$ depends on the specific method of retrieving tuples.

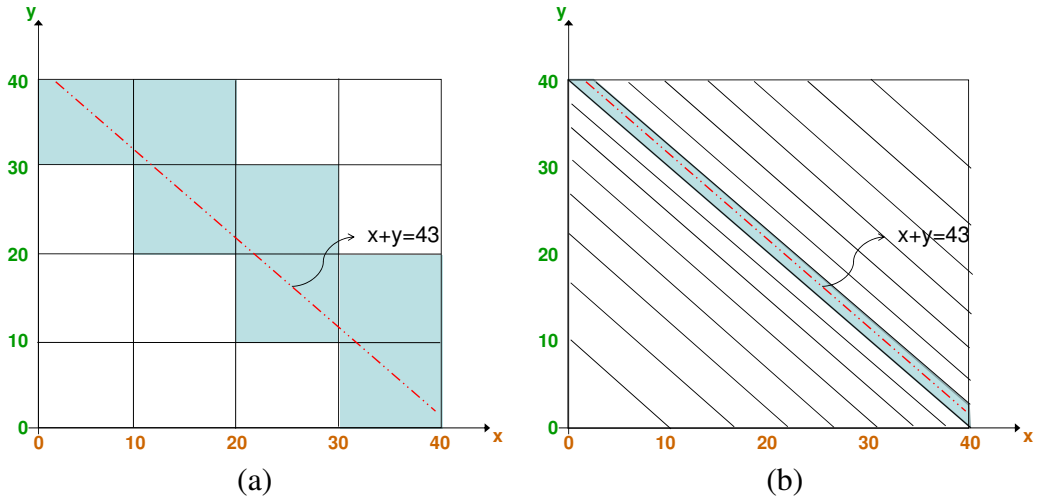


Figure 5.4: The relationship between constraints and bounds.

Figure 5.3 summarizes the relationships among the cost parameters, in determining the number of candidate tuples. We use the up-arrow and down-arrow to represent "increase in amount" and "decrease in amount", respectively. *First*, the number of candidate tuples is directly determined by two cost parameters, the bounds and the cardinalities. To be more specific, the more tuples in each candidate bucket, the more candidate tuples; and the bigger interval between the upper-bound and the lower-bound ($\lceil b \rceil - \lfloor b \rfloor$) of each bucket, the more candidate buckets (since the chance of subsuming the query tuple score is bigger), thus the more candidate tuples.

Second, the cardinalities and bounds are determined by the partition, *i.e.*, the number of buckets and the constraints of each bucket. The relationship between the number of buckets and the cardinalities/bounds is clear. The more buckets, the less tuples in each bucket, and the smaller intervals between the upper and lower bounds. This seems to indicate that we should have as many buckets as possible, since that can result in less candidate tuples. However, with more buckets, the cost of constructing the buckets and computing their cardinalities ($\mathcal{C}3_{I/O}$ and part of $\mathcal{C}_{CPU}$) can be significant.

In addition to the number of buckets, the constraints also determine the bounds. This relationship is illustrated by the following example.

Example 25: Figure 5.4 shows a two-dimensional space over attributes x and y of relation R.

Consider ranking function x+y. We are looking for the rank of a tuple with score 43. The tuples with the same ranking score locate on a unique contour line, one for each score value. For instance, the contour line for x+y=43 is shown as a dashed line in Figure 5.4(a) and (b).

Different constraints can result in very different bounds. Figure 5.4(a) and (b) show two partitions of the same space, where the solid lines are the boundaries of the buckets. The partition in (a) uses constraints of the form $\{x_1 \leq x < x_2,\ y_1 \leq y < y_2\}$, while the partition in (b) has constraints of the form $\{l \leq x+y < u\}$, *i.e.*, the constraints are parallel to the contour lines of the ranking function. Both (a) and (b) partition the space into 16 buckets with the same area size, thus roughly the same cardinality under the assumption of uniform data distribution. The figures show that, although containing about the same number of tuples, the buckets in (b) have much smaller intervals between bounds than the buckets in (a) have. Therefore there are 7 candidate buckets (in shade) in (a), while (b) only has 1 candidate bucket. ■

The above example illustrates that which constraint results in the smallest intervals between bounds depends on the ranking function itself. For instance, the contour lines for $2x$+y have different slope than that in Figure 5.4, thus the constraints parallel to the contour lines are also different.

5.2.4 Partitioning Schemes

The analysis in Section 5.2.3 indicates that the most significant cost component, the number of candidate tuples, is determined by the partitioning scheme, which consists of the number of buckets and the constraints, by definition. The constraints of the buckets specify the way to partition, while the number of buckets indicates the granularity of the partition in that way. Below we present several partitioning schemes, *i.e.*, various types of constraints. These schemes are implemented by the methods in Section 5.3.

Single-Attribute Constraints:

A straightforward approach of partitioning is to associate with the buckets the simplest constraints,

which are intervals (ranges) over individual attributes. That is, a constraint has the form $l \leq a < u$, where a is one attribute, and l and u are some constant values. In this partition, the boundaries between buckets are parallel to the dimensions, *i.e.*, attributes. One advantage of using single-attribute constraints is that they can support any monotonic functions on these attributes. Moreover, it is easy to conduct satisfaction test for these constraints. For instance, a B-tree on a may be used in obtaining those tuples satisfying constraint $l \leq a < u$.

Function Constraints:

Partitioning based on single-attribute constraints can be sub-optimal, depending on the ranking functions. As discussed in Section 5.2.3, the constraints should be aligned with the contour lines of the ranking function, in order to achieve small number of candidate tuples. Following this intuition, we propose a partitioning scheme based on constraints as functions.

In this scheme, each constraint is of the form $l \leq g < u$, where g is a linear function. Given a linear ranking function f, if g is "close" to f (in other words, g is aligned with f), the number of candidate buckets and tuples can be small. Such closeness can be measured by the angle between f and g, *i.e.*, the cosine similarity of their coefficients. Note that this scheme is applicable when we consider only linear ranking functions and linear constraints. The problem of computing the bounds of such ranking functions, given the constraints, is a linear programming problem, as we discussed in Section 5.2.2.

Satisfaction test for such function constraints requires us to build auxiliary data structures. That is, we need to build an index over g (instead of single attribute a).

Workload-Based Function Constraints:

For the above scheme using function constraints, in order to achieve small number of candidate tuples, the function g used in the constraint should be close to f. In other words, we must have indices built for various different g, so that among them we can find one that is close to the dynamic ranking function f in the query. However, as the number of attributes involved in ranking functions increases, the necessary number of indices for the g may become prohibitively large.

Our idea in tackling this challenge is use query workload to guide the selection of functions g to build index for. The indexed functions are chosen such that they are "close" to many previous queries according to the query workload. With the reasonable expectation that future queries share the same characteristics with the workload, the indexed functions can capture many queries in the future. [9]

5.3 Implementation Methods

In this section, we present several implementation methods of the partition-and-prune framework in Section 5.2. These methods utilize a variety of data structures in DBMSs including histogram, B-tree, multi-dimensional index, and bitmap index. When introducing each method, we first describe the details of partitioning space (step 1), computing cardinalities (step 3), and retrieving candidates (step 5), respectively. Then we analyze the pros and cons of the method. We first introduce those methods realizing partitioning schemes based on single-attribute constraints (Section 5.3.1), then propose methods for schemes using function constraints and workload-based function constraints (Section 5.3.2). Finally, we discuss how to deal with Boolean selection and join conditions in the queries (Section 5.3.3).

5.3.1 Implementations of Partitioning by Single-Attribute Constraints

Using Multi-Dimensional Histogram

Histogram is commonly used in query optimization (for selectivity estimation) and approximate query answering in database systems [59]. A multi-dimensional histogram naturally partitions the tuple space into buckets, with the cardinality of every bucket stored.

Partitioning Space: In a histogram, each bucket is defined by intervals (ranges) over individual attributes. For instance, the partition in Figure 5.1 can indeed be a two-dimensional histogram.

[9] By the same intuition, in other works, query workload is also used in choosing conventional indices to construct and views to materialize.

Computing Cardinalities: The advantage of using multi-dimensional histogram is that the cardinalities of buckets are pre-computed.

Retrieving Candidates: The histogram maintains cardinality information, but cannot provide access to individual tuples. Therefore, we must use SQL range queries, one for each bucket, to retrieve the tuples in the candidate buckets.

Example 26: Suppose Figure 5.1 is a histogram. The range queries corresponding to the candidate buckets are shown below. Note that the multiple range queries are concatenated by **union** in a single SQL query. An alternative is to use disjunctive conditions in the **where** clause, *i.e.*, ($10 \leq a$ and $a < 20$ and $30 \leq b$ and $b < 40$) or $\cdots$ or ($30 \leq a$ and $a < 40$ and $20 \leq b$ and $b < 30$).

(**select** * **from** R
where $10 \leq a$ and $a < 20$ and $30 \leq b$ and $b < 40$)
union
...
union
(**select** * **from** R
where $30 \leq a$ and $a < 40$ and $20 \leq b$ and $b < 30$)

■

Although using multi-dimensional histogram provides cardinality automatically, this method has serious disadvantages. *First*, the transformed SQL query is inefficient to evaluate, as the multiple range queries may require the access to the full domain of every attribute. To illustrate, consider Figure 5.5. Suppose the ranking function is x+y+z, and the query asks for the rank of a tuple with score 16. The candidate buckets must at least contain the gray plane x+y+z=16, which spans through the whole domain of x, y, and z, respectively. Note that multi-dimensional histogram was also used in answering top-k queries [19]. However, only one range query is needed for one top-k query, because the candidates of top k answers are located in a small area around the query point.

Second, the dimensions in the histogram may not match the attributes in the ranking function, resulting in loose upper-bound and lower-bound. The loose bounds further produce significant overlapping among the buckets' bounds, thus large number of candidate buckets. For instance, suppose the histogram in Figure 5.1 is used to answer another inverse ranking query with ranking

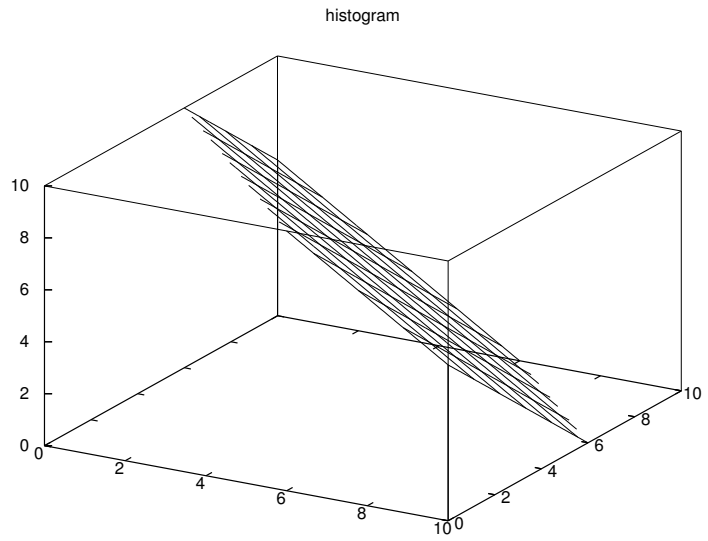


Figure 5.5: Inverse ranking query for a tuple with score $x+y+z$=16, using histogram.

function a+b+c, instead of a+b. The attribute c has domain $[0, 40)$. Since the histogram only uses a and b to partition the space, a constraint, $0 \leq c < 40$, is implicitly given for every bucket. With such an identical loose constraint, all the buckets may become candidates.

Traversing Multi-Dimensional Index

Similar to multi-dimensional histogram, multi-dimensional index is also a partition scheme where buckets are specified by intervals over individual attributes. Differently, multi-dimensional index provides access to tuples, but does not contain cardinality information.

Partitioning Space: The index nodes can be viewed as the buckets. There exists a hierarchy in the index tree, thus a hierarchy for the buckets as well. Common multi-dimensional index can be tuple-partitioning (*e.g.*, R-tree [46]) where the buckets may overlap although one tuple only belongs to one leaf node, or space-partitioning (*e.g.*, grid file [78]) where the buckets do not overlap. For instance, the partition in Figure 5.1 can be a grid file.

Computing Cardinalities: To compute the cardinalities of all the buckets, we have to count the tuples in the corresponding index nodes, resulting in a full traversal of the index tree. This cost can be avoided by augmenting each index node with the number of tuples in the node (and the

descendant nodes). Such a variant was briefly discussed in [59]. Recording the cardinality takes several extra bytes per node, resulting in smaller number of pointers that can be stored in each node. Specifically, the consequence in R-tree is smaller fan-out of tree nodes, thus deeper index tree [59].

Retrieving Candidates: The leaf index nodes in a multi-dimensional index provide pointers to individual tuples in the corresponding buckets.

In answering inverse ranking queries, multi-dimensional index has some problems similarly existing for multi-dimensional histogram. For instance, the attributes in the index may not match the attributes of the ranking function. Therefore the score bounds can be loose, resulting in a large number of candidate buckets, as we analyzed in Section 5.3.1. The problem of mismatching between indexed attributes and ranking attributes seriously limits the applicability of this method, since it is not affordable to exhaustively build indices for all possible combinations of dimensions.

Moreover, for multi-dimensional index with tuple-partitioning (*e.g.*, R-tree), the boundary of an index node (*e.g.*, minimal bounding rectangle or *MBR*, in R-tree) is determined towards the efficiency of the insertion/deletion operations over index, which may conflict with the efficiency in answering inverse ranking queries. For instance, the *MBR* in R-tree may stretch over a big range along one dimension, resulting in a large interval between the upper-bound and lower-bound of the corresponding bucket. The consequence is a large number of candidate buckets.

Intersecting B-tree Indices

While the method in Section 5.3.1 relies on the existence of multi-dimensional index, we can also partition the space by intersecting the indices (such as B-trees) over individual attributes. Such index intersection was used in evaluating selection queries [72].

Partitioning Space: A B-tree index naturally provides a partition of an attribute domain into multiple intervals. Therefore by intersecting multiple B-trees, we obtain a tuple-space partition.

Computing Cardinalities: Index intersection supplies the list of tuple pointers in each bucket, and thus the cardinality.

Retrieving Candidates: Again, the results of index intersection provide the (pointers to) tuples in each bucket.

The advantage of using index intersection is that we can handle any combination of ranking attributes as long as the individual indices are available. However, unlike multi-dimensional index where the space is readily partitioned, index intersection requires explicit operations to partition the space. The main overhead of this method thus lies in the partitioning, where the B-tree over each attribute must be fully traversed, and the tuple lists are intersected. The result of partitioning is the tuple lists for all the buckets. The traversal on each index may repeat multiple times if the memory buffer cannot hold all the index nodes from all the indices.

Intersecting Bitmap Indices

The method proposed in this section intersects bitmap indices instead of B-trees. While the essential methodology is the same for intersecting both types of index, the key is that bitmap index is more efficient to intersect. A brief review of bitmap index was given in Section 4.3.3. Below we discuss how to use such index to answer inverse ranking queries.

Partitioning Space: Similar to the aforementioned several methods, the buckets are specified by intervals on individual attributes.

Computing Cardinalities: The IDs of tuples within one specific interval on an attribute are given by the corresponding bit vector. Tuples inside a bucket can thus be obtained by intersecting (**and** operation) all the relevant bit vectors. A 1 bit in the resulting vector indicates that the corresponding tuple belongs to the bucket. Computing the cardinality of the bucket is thus a **count** operation on the resulting vector.

Retrieving Candidates: The union (**or** operation) of the vectors for all the candidate buckets gives us a single vector, where the 1 bit corresponds to a candidate tuple. Thus we get the IDs of all candidate tuples.

We illustrate the idea using the following example.

TID	B_a^1	B_a^2	B_a^3	B_a^4
r_1	1	0	0	0
r_2	0	0	1	0
r_3	1	0	0	0
r_4	0	1	0	0
r_5	0	0	0	1
r_6	1	0	0	0
r_7	0	0	1	0
r_8	0	0	0	1
r_9	0	0	1	0
r_{10}	0	1	0	0

(a) bitmap index on $R.a$

TID	B_b^1	B_b^2	B_b^3	B_b^4
r_1	0	0	0	1
r_2	0	1	0	0
r_3	0	0	1	0
r_4	0	0	1	0
r_5	1	0	0	0
r_6	0	1	0	0
r_7	0	0	1	0
r_8	0	0	0	1
r_9	0	1	0	0
r_{10}	0	0	1	0

(b) bitmap index on $R.b$

Figure 5.6: Example of bitmap indices.

Example 27: Figure 5.6(a) and (b) show the bitmap index on attribute a and b over relation R. The values of a and b are grouped into 4 intervals, respectively, following the partition in Figure 5.1. Therefore there are 4 bit vectors on a and 4 on b as well. For instance, the 4th bit of vector B_a^2 is 1, which indicates that the value of attribute a for tuple r_4 (*i.e.*, the tuple with ID 4) is in the 2nd interval, *i.e.*, $[10, 20)$. Performing **and** operation on each pair of vectors (one for a and another for b), we obtain the resulting vectors for the 16 buckets. For instance, the result of B_a^2 **and** B_b^3 is 0001000001, which contains two 1-bits, thus the cardinality of bucket $\{10 \leq a < 20,\ 20 \leq b < 30\}$ is 2. (Note that we use the same partition as Figure 5.1, but different cardinalities, for simplicity in illustrating the idea.) ■

The advantage of using bitmap index is that the bit operations in getting the vectors for buckets are much more efficient than traversing the B-tree nodes and intersecting the verbatim list of tuple IDs. However, in multi-dimensional space with many intervals on each dimension, the number of buckets and thus the number of bitmap operations can be fairly high, resulting in prohibitive cost of computing cardinalities.

5.3.2 Implementations of Partitioning by Function Constraints

Intersecting Bitmap Indices on Functions

Following the intuition of using function constraints (Section 5.2.4), we propose a method of intersecting bitmap indices on functions. Different than the method in Section 5.3.1, the bitmap indices intersected in this approach are built upon ranking functions instead of individual attributes. The motivation in building bitmap index instead of B-tree index on the functions is that, intersecting bitmap index is more efficient than intersecting B-trees.

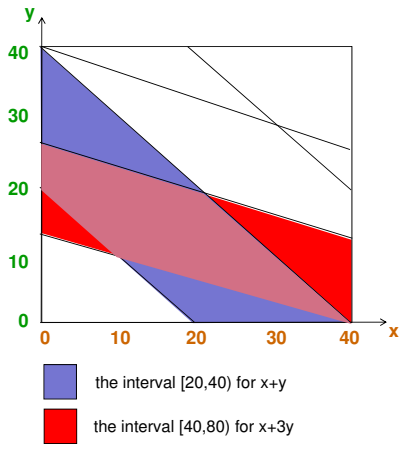
During index construction, a bitmap index is created for each selected ranking function. It consists of several bit vectors, each of which corresponds to an interval of ranking scores over the ranking function. For a database tuple, its corresponding bit in the vector for the interval subsuming its ranking score is set to 1, and the same bits in other vectors are 0. During query answering, we select one or more indices (functions) that are close to the ranking function in the query. The constraints of the buckets are specified by intervals of scores over the chosen functions. After that, the procedures of computing cardinalities and retrieving candidates are essentially the same as in Section 5.3.1. Note that the ranking functions and the indexed functions must be linear functions in order to make this method applicable.

Example 28: Consider ranking functions $w_1 \times a_1 + ... + w_n \times a_n$, where each a_i is an attribute and w_i is the corresponding weight. Given a specific combination of $(w_1, ..., w_n)$, the tuples in the space are ranked in the order of the contour lines $w_1 \times a_1 + ... + w_n \times a_n = s$, where s is the ranking score. Therefore we can construct a bitmap index for the given function, *i.e.*, the $(w_1, ..., w_n)$, with several bitmaps. Each bitmap of the index corresponds to a score interval.

Figure 5.7(a) shows a relation R with its tuples and the attribute values. Suppose the ranking function in the query is x+$2y$. Among the functions with corresponding bitmap indices constructed, the two functions x+y and x+$3y$ are chosen for answering the query, since they are close to x+$2y$. Their indices are shown in Figure 5.7(c) and (d), respectively. Specifically, the index for x+y consists of 4 bitmaps, corresponding to the score intervals $[0, 2)$, $[2, 4)$, $[4, 6)$, and $[6, 8)$.

TID	x	y
r_1	1	0
r_2	3	3
r_3	2	4
r_4	0	3
r_5	2	1
r_6	4	3
r_7	0	0
r_8	1	3
r_9	3	1
r_{10}	1	4

(a) The relation R.



(b) The intersection of bitmap indices.

TID	$B^{[0,2)}$	$B^{[2,4)}$	$B^{[4,6)}$	$B^{[6,8)}$
r_1	1	0	0	0
r_2	0	0	1	0
r_3	0	0	1	0
r_4	0	1	0	0
r_5	0	1	0	0
r_6	0	0	0	1
r_7	1	0	0	0
r_8	0	1	0	0
r_9	0	1	0	0
r_{10}	0	0	1	0

(c) Bitmap index for $x + y$.

TID	$B^{[0,4)}$	$B^{[4,8)}$	$B^{[8,12)}$	$B^{[12,16)}$
r_1	1	0	0	0
r_2	0	0	1	0
r_3	0	0	0	1
r_4	0	0	1	0
r_5	0	1	0	0
r_6	0	0	0	1
r_7	1	0	0	0
r_8	0	0	1	0
r_9	0	1	0	0
r_{10}	0	0	0	1

(d) Bitmap index for $x + 3y$.

Figure 5.7: The intersection of bitmap indices on functions.

The bitmap index for x+3y consists of 4 bitmaps as well, corresponding to the intervals $[0, 4)$, $[4, 8)$, $[8, 12)$, and $[12, 16)$. The boundaries between these intervals, *i.e.*, the contour lines, are shown in Figure 5.7(b). The intersections of these intervals give the partition of the tuple space. For instance, the bucket corresponding to the intersection of the two shaded areas has constraints $\{2{\leq}x{+}y{<}4,\ 8{\leq}x{+}3y{<}12\}$. The upper-bound and lower-bound scores for this bucket are 6 and 3, respectively, based on linear programming. The bitmap for $2{\leq}x{+}y{<}4$ is 0001100110, and the bitmap for $8{\leq}x{+}3y{<}12$ is 0101000100. Therefore the bitmap for the shaded bucket is 0001100110 **and** 0101000100 = 0001000100 . That is, tuples r_4 and r_8 are in this bucket. ■

Heuristics for Choosing Index to Build

We discuss two index selection heuristics. The first heuristic, *random selection*, is simply to choose arbitrary functions to build index for. Clearly this strategy has the problem of exponential explosion- As the number of attributes involved increases, the hope of an arbitrary indexed function getting close to a future dynamic query is slim. This "curse of dimensionality" is well known in many other areas, such as multi-dimensional indexing.

To address the dimensionality problem, our second heuristic, *workload-based selection*, is to build bitmap indices for those functions that capture the query workload. By doing that, we achieve efficiency for the more frequent queries and sacrifice the less frequent ones. To be more specific, each linear ranking function can be viewed as a point in a multi-dimensional space. Given a set of previous queries, *i.e.*, a set of points in the space, we partition the space into buckets. [10] Associated with each bucket is a *virtual query*, located at the center of that bucket. We thus capture the queries in the bucket as a set of queries identical to the virtual query. This is based on the intuition that if the partition of the query space is fine-grained enough, the queries inside the same bucket are close enough to each other. After measuring the number of queries in each bucket, we choose to build bitmap indices on the virtual query functions of those buckets that contain a large number of queries.

The workload-based selection is effective only when there do exist frequent queries in the workload, *i.e.*, the workload is clustered. In other words, if the queries in the space have equal probability to be issued by users, then the strategy degrades to the above random selection heuristic. For (inverse) ranking queries, it is natural that the workload is clustered. *First*, in ranking a certain type of objects, there usually exists "common sense". *Second*, even though different people have different ranking criteria, similar users share common interests.

[10] The space and buckets of queries should not be confused with the space and buckets of tuples in Section 5.2.1.

Heuristics for Choosing Index to Intersect

With the bitmap indices built for the workload, given a new query, we select two indices that are closest to the query and intersect them. To be more specific, suppose the linear ranking function in the query is f: $w_1 \times a_1 + ... + w_n \times a_n$, where w_i is the weight and a_i is the attribute. Given an indexed function g: $w'_1 \times a'_1 + ... + w'_n \times a'_n$, the closeness of f and g is defined as their cosine similarity,

$$closeness(f, g) = \frac{\overrightarrow{v_f} \cdot \overrightarrow{v_g}}{\|\overrightarrow{v_f}\| \, \|\overrightarrow{v_g}\|}, \tag{5.3}$$

where $\overrightarrow{v_f}$=$<w_1, \ldots, w_n>$ and $\overrightarrow{v_g}$=$<w'_1, \ldots, w'_n>$.

Note that although this method is also based on multi-dimensional space, it does not suffer from the attribute mismatching problem for the approach utilizing multi-dimensional index (Section 5.3.1). The reason is that a function such as $w_1 \times a_1$ is a special case of functions involving more attributes such as $w_1 \times a_1 + w_2 \times a_2$. Therefore as long as $w_1 \times a_1$ appears frequently, the workload on dimensions (a_1, a_2) is able to capture it.

5.3.3 Dealing with Boolean Conditions

Up to this point, we always assume the nonexistence of Boolean conditions, *i.e.*, we assume the context relation $R_{\mathcal{B}}$ is simply one base table. This assumption was made only for the purpose of easy presentation. In fact, logical bit vector operations easily allow us to integrate the techniques in Section 5.3.2 with Boolean conditions. Before performing the intersections of bit vectors as shown in Figure 5.7, the vectors over the indexed function intervals must reflect the filtering effects of the Boolean conditions. If a tuple does not belong to the context relation $R_{\mathcal{B}}$, we must set its corresponding bits in the vectors to 0. The details are further explained below. Note that the techniques applied here are similar to those in Section 4.4.2, where bitmap index is used to incorporate clustering with ranking, together with Boolean conditions.

Selections: Suppose our query has a set of conjunctive range selection conditions $l_1 \leq c_1 \leq u_1, \ldots, l_j \leq c_j \leq u_j$. We first obtain a vector $vec_{\mathcal{B}}$, where the corresponding bits for the satisfying tuples in $R_{\mathcal{B}}$ are all set. Then given the bit vectors of the indexed functions to be intersected (such as the ones in Figure 5.7(c) and (d)), we intersect $vec_{\mathcal{B}}$ with each vector, to derive the filtered vectors. The are many works in the literature on answering Boolean queries using bitmap index, *i.e.*, getting $vec_{\mathcal{B}}$. We briefly discussed this issue in Section 4.4.2.

Joins: When join conditions exist in queries, we assume the tables have a snowflake-schema, consisting of one fact table and multiple dimension tables. There are multiple dimensions, each of which is described by a hierarchy, with one dimension table for each node on the hierarchy. The fact table is connected to the dimensions by foreign keys. The tables on each dimension are also connected by keys and foreign keys. As a special case of snowflake-schema, star-schema has only one table on every dimension, thus no hierarchy. Under such assumption, the same techniques for handling join queries in integrating clustering and ranking (Section 4.4.2) can be applied.

5.4 Experiments

The algorithms are implemented in C++. The bitmap index is based on [89], which builds multiple bitmap indices at different domain resolutions and compresses them using the WAH compression method [96]. The B-tree index intersection algorithm is built upon a publicly available B-tree implementation in *libgist*, an library that implements *GiST* [52].

We conducted experiments to compare the various implementation methods in Section 5.3, together with the straightforward exhaustive approach. Moreover, we investigated how the algorithms are affected by important factors under various configurations. The compared algorithms are: exhaustive method using sequential scan on single table (*Scan*), exhaustive method using sort-merge join (*SMJ*), B-tree intersection (*Btree*), intersecting bitmap index on attributes (*BAttr*), intersecting bitmap index on randomly selected functions (*BFuncRND*), and intersecting bitmap index on workload-based functions (*BFuncWKLD*).

5.4.1 Experimental Settings

Our experiments were conducted over synthetic tables. The schema of a table consists of a set of attributes, altogether with the total length of 100-byte. Some of the attributes are 4-byte floating number ranking attributes. The values of the ranking attributes are independently generated by various distributions, including uniform, Gaussian, and cosine distributions. Each query used in our experiments uses the weighted sum of the ranking attributes as the ranking function.

We also experiment with join queries in star-schema. The join condition is $A.j$=$B.j1$ **and** $B.j2$=$C.j$, where $A.j$ and $C.j$ are keys of A and C, respectively. $B.j1$ and $B.j2$ are the foreign keys in B referring them. A, B, and C have the same size. Among the tuples in A, about half of them do not join with any tuple in B. Each tuple in the remaining half in average joins with 2 tuples in B. The same statements apply to the join between C and B.

The ranking functions in the queries are weighted-sum functions. We experiment with various number of attributes involved in the functions. The workload is created by a data generator for clustering algorithms from [34]. Viewing each query ranking function as a point, *i.e.*, a vector of weights, in the query space, the workload is a set of clusters. The same data generator was used for the experiments in Section 4.6, where more details about the data generator can be found.

The experiments were conducted on a PC with 2.8GHz Intel Xeon SMP (dual hyperthreaded CPUs each with 1MB cache), 2GB RAM, and a RAID5 array of 3 146GB SCSI disks, running Linux 2.6.15.

5.4.2 Experimental Results

We evaluated the performances of various methods and studied how they are affected by several important configuration parameters, which are summarized in Table 5.1. For *BFuncRND* and *BFuncWKLD*, by default there are bitmap indices built for 200 functions. For *BFuncWKLD*, the functions are chosen based on a query workload containing 500 queries. For each index on one function, we use the BSI [84] mentioned in Section 4.3.3. To be more specific, the tuples' values on

parameter	meaning	values
t	# tuples	400K, 800K, 4M, 8M
a	# ranking attributes	2,3,4,5,6,7
q	rank of the query tuple (in percentage)	1%, 10%, 25%, 50%
i	# index built	100, 200, 300, 400
v	# vector per index	7, 8, 9, 10

Table 5.1: Configuration parameters for experiments on inverse ranking.

a function are partitioned into multiple ranges. The binary representation of these range numbers on this function is kept in v (Table 5.1) vectors, which can 2^v ranges.

Single-Table Queries:

To evaluate the performance of single-table queries, we conducted experiments under groups of configurations by the value combinations of the three relevant parameters, t, a, and q. In each group of experiments, we varied the value of one parameter and fixed the values of the rest two. We then run all the algorithms and studied how their performances are affected as the value of the varying parameter changed. The results on wall-clock execution time under six sample groups of experiments are shown in Figure 5.8, 5.9, and 5.10. From the figures, we can make the following observations:

First, *BFuncWKLD* is usually the best algorithm and it is several times more efficient than others. This validates the approach of using bitmap index built upon query workload.

Second, for single-table queries, *Scan* performed pretty well in comparison with various other methods, except *BFuncWKLD*. This observation verifies our analysis of the various methods in Section 5.3.1. They all have significant disadvantages. For instance, Figure 5.9 shows that, as the number of ranking attributes increases, the performance of intersecting B-trees degrades exponentially due to the fact that it has to fully traverse all the B-trees and intersect the tuple pointers. As another example, Figure 5.9(b) clearly illustrates the "curse of dimensionality" on using bitmap indices built upon randomly selected functions, as mentioned in Section 5.3.

Third, from Figure 5.10 we can see that the rank position is also important in determining

the efficiency. number of rank position. Especially, the smaller rank position, the more efficient *BFuncRND* and *BFuncWKLD* are. To obtain the rank of an object that is ranked at 1% (*e.g.*, the object ranked at 8192 when t=800K), *BFuncRND* outperforms *Scan*. However, as the rank position increases, it becomes worse than *Scan*. This figure clearly shows that inverse ranking queries are more challenging than top-k queries, in the sense that efficient approach for obtaining objects at small k may become even worse than the straightforward approach.

Join Queries:

The results of join queries are shown in Figure 5.11$-$ 5.13, in the same fashion as the results for single-table queries. Note that *SMJ* replaces *Scan* for joining, and *Btree* becomes inapplicable for join queries. From the figures, we can make the following observations:

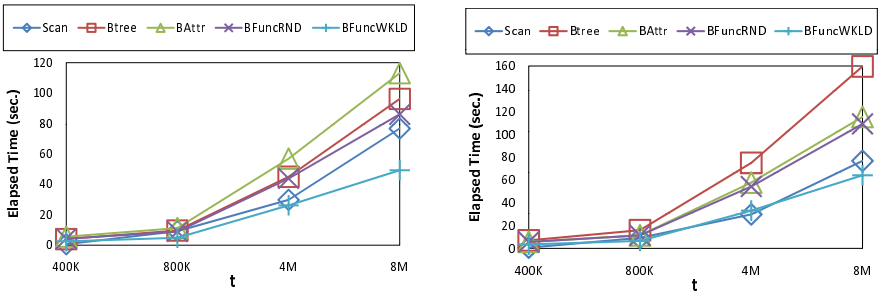
First, *BFuncWKLD* is still clearly the best algorithm, and its performance advantages over other algorithms are enlarged under join.

Second, different from the single-table scenario, the exhaustive approach *SMJ* now is often the worst method. This is due to the fact that a full join will scan large number of tuples and join large number of intermediate results. While other methods are able to zoom into candidate tuples, thus reduce the number of I/Os.

Third, *BFuncRND* is often the second best method under small number of ranking attributes. However, the "curse of dimensionality" is illustrated again by Figure 5.12.

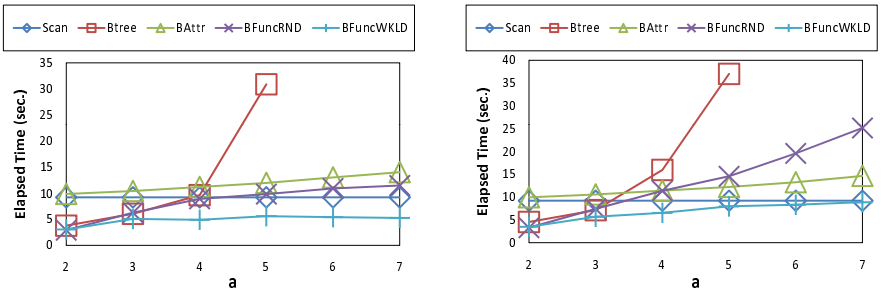
To further understand *BFuncRND* and *BFuncWKLD*, we analyze how the parameters affect their performances, as shown in Figure 5.14. As expected, as the number of built indices increases (Figure 5.14(a)), the algorithms are more efficient. However, 400 indices do not give us significant performance improvements over 100 indices. This indicates that the small number of indices are sufficient for the given workload. Under other workload, more indices may be required. In Figure 5.14(b), we see that increasing the number of vectors in fact makes the performance of *BFuncRND* worse. The reason is that randomly selected functions cannot match the query functions well, resulting in a large number of candidate buckets. As there are more vectors per index,

the partition has more buckets, therefore *BFuncRND* needs to intersect more bit vectors, compute the cardinalities for more candidate buckets, resulting in degraded performance. On the other hand, *BFuncWKLD* is not seriously affected by v, indicating that the workload-based functions can successfully capture the queries, resulting in small number of intersections and candidate buckets.



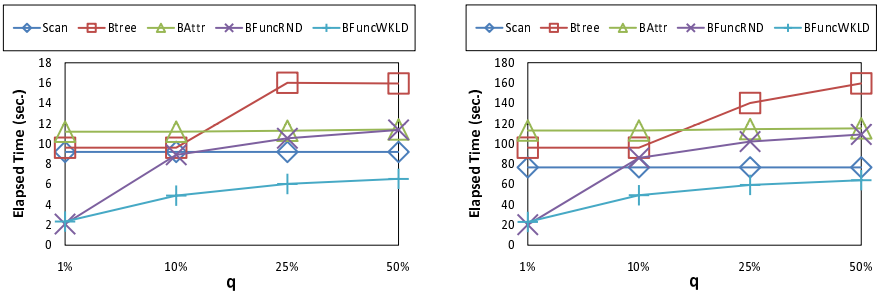
(a) a=4, q=10%. (b) a=4, q=50%.

Figure 5.8: Single table queries: Execution time varying by t.



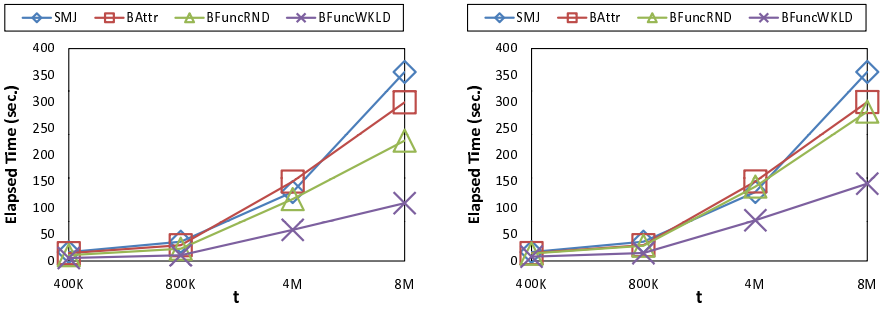
(a) t=800K, q=10%. (b) t=800K, q=50%.

Figure 5.9: Single table queries: Execution time varying by a.



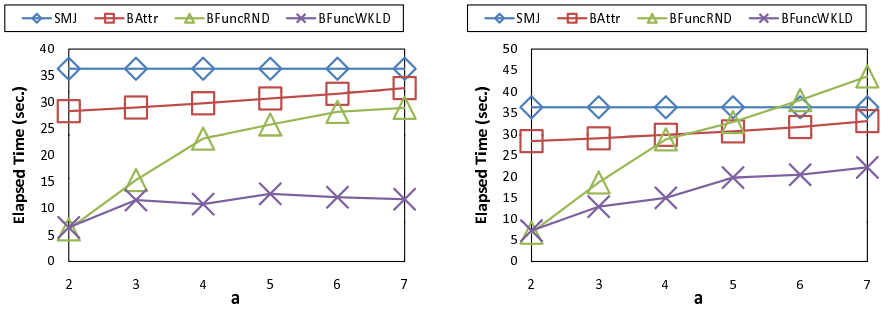
(a) t=800K, a=4%. (b) t=8M, a=4%.

Figure 5.10: Single table queries: Execution time varying by q.



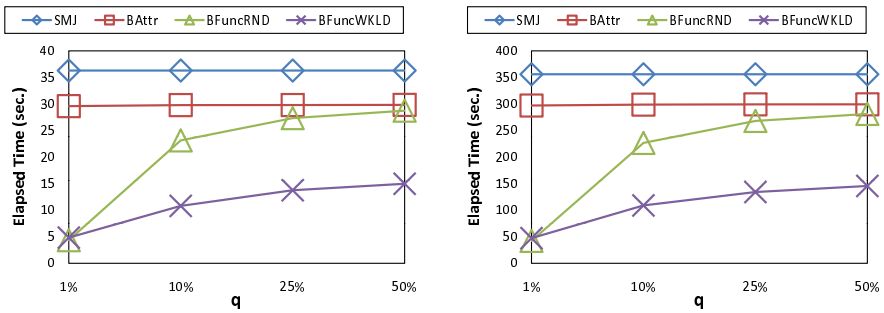
(a) a=4, q=10%. (b) a=4, q=50%.

Figure 5.11: Join queries: Execution time varying by t.



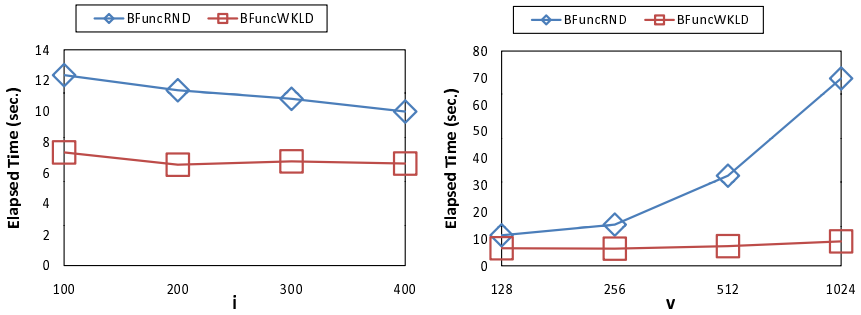
(a) t=800K, q=10%. (b) t=800K, q=50%.

Figure 5.12: Join queries: Execution time varying by a.



(a) t=800K, a=4%. (b) t=8M, a=4%.

Figure 5.13: Join queries: Execution time varying by q.



(a) t=800K, a=4, q=50%, varying i. (b) t=800K, a=4, q=50%, varying v.

Figure 5.14: Single table queries: Execution time varying by i and v.

Chapter 6

Initial Release of RankSQL

RankSQL is implemented inside the kernel of POSTGRESQL, an open-source DBMS. We have received requests for obtaining the source codes of our systems and we are releasing the initial version to the public. Our intention is to facilitate the research in the general area of supporting flexible retrieval functionalities in databases, to get support and help from the large community of open-source developers, and to look for opportunities of deploying the system in production environments.

The initial release of RankSQL has the implementation of the rank-relational algebra (Chapter 2), including several ranking query operators, a ranking query optimizer with a 2-dimensional enumerator and simple heuristic rules for costing the operators, a simple parser of ranking queries, a plan builder for manually constructing ranking query plans, and a visualizer for investigating the process of query optimization. The system also includes the implementation of the *rankagg* framework (Chapter 3) including the *rankagg* operator, the group- and rank-aware scan operator, the corresponding new join operator, and a query optimizer that utilizes the new operators for ranking aggregate queries.

We will continue the development and maintenance of RankSQL, including incorporating the system with the support of *ClusterRank* and *InverseRank* (which are currently implemented outside the kernel) and extending the system with new data retrieval functionalities through our research. Further releases of RankSQL will be made available to the public.

6.1 Introduction to POSTGRESQL

The history of POSTGRESQL starts with the INGRES project (from early 1970s to early 1980s) at the University of California, Berkeley, which was under the lead of Michael Stonebraker. After commercializing INGRES, he started the POSTGRES project (meant *Post Ingres*) around 1985. After the release of the first version in 1989 and several more releases, the POSTGRES project officially ended at the fourth version in 1993. The software was released under the BSD license, thus giving developers the access to the source codes and allowing them to further develop the system. In 1994, two Berkeley graduate students, Andrew Yu and Jolly Chen, added a SQL interpreter into POSTGRES and released the enhancement to public as POSTGRES95. The first open source version was released at August 1st, 1996, by a group of open source developers, including Marc Fournier, Bruce Momjian and Vadim Mikheev. The system was later renamed POSTGRESQL in 1997. Since then, there have been continuous releases of new versions and commercializing efforts. POSTGRESQL is one of the most popular open-course DBMSs, competing with other popular open-source ones such as MYSQL and the systems from commercial vendors.

One distinguishing advantage of POSTGRESQL is that the system embraces advanced features that are not available in some other systems. Many of the features were originated by the research community. Examples of advanced features include *GiST* (Generalized Search Trees [52]), bitmap index scans, user-defined objects including index on user-defined functions, support of expensive predicates [53] (now removed from the main release), and genetic query optimizer. Therefore POSTGRESQL has remained one of the most popular systems used in research projects.

The architecture of POSTGRESQL essentially follows a typical textbook description of a DBMS, as shown in Figure 6.1. The implementation of the components in the system is also fairly standard. When receiving a user query from the frontend, the parser in the backend query processor parses the query and comes up with an initial logical query plan. The optimizer generates a physical query plan, which is expected to be efficient, with the help of a System-R [15] style dynamic-programming based enumerator and cost models for operators, by utilizing system catalogs and

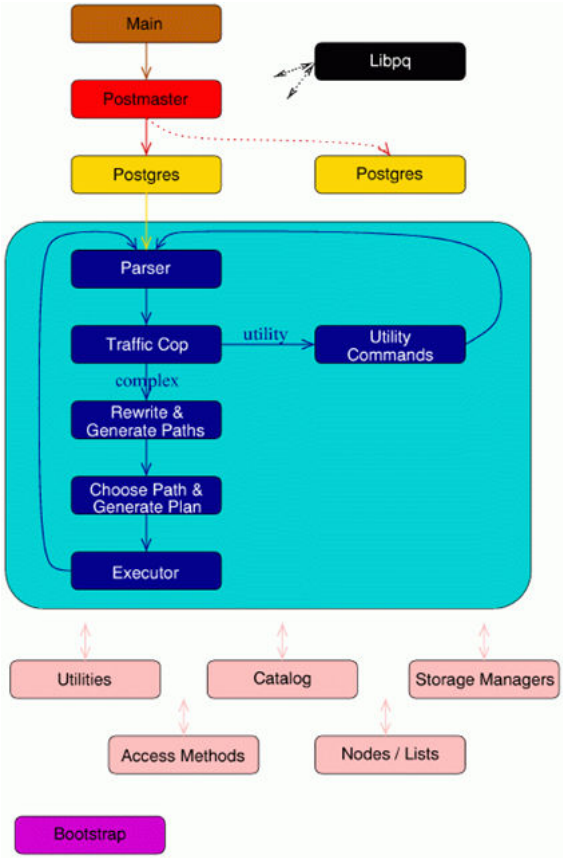


Figure 6.1: The architecture of POSTGRESQL [73].

statistics. The query plan is executed by the executor, producing the query results.

6.2 The Architecture of RankSQL

The RankSQL system is implemented inside POSTGRESQL 7.4.3. The code base of POSTGRESQL contains about $500,000$ lines of C programs, of which about 15% belongs to the query engine. RankSQL introduced about $20,000$ lines of new codes into the system. Figure 6.2 illustrates the architecture of RankSQL. The fundamental support of ranking imposes significant impacts on the whole system, including the underlying data model and algebra, the query parser for ranking queries and ranking aggregate queries, the newly invented query operators, and the optimization techniques.

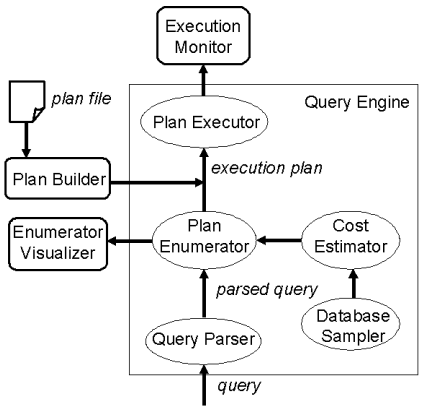


Figure 6.2: The architecture of RankSQL.

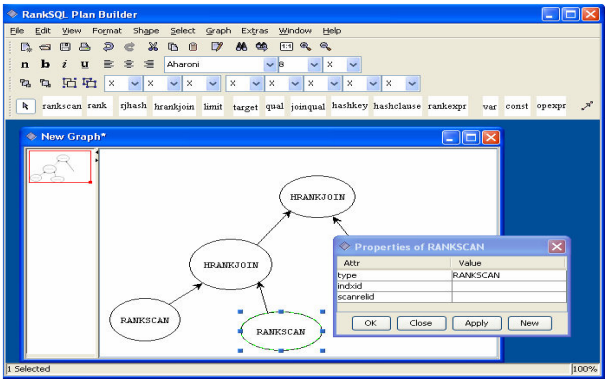


Figure 6.3: The Plan Builder.

In addition to the core query engine, we built a suite of useful tools for system builders to explore the process of query optimization and execution in our system. We also developed a Java GUI with JDBC connection to demonstrate the system and tools. We first introduce the tools in Section 6.2.1, and then describe a sample runtime scenario of RankSQL in Section 6.2.2.

6.2.1 The RankSQL Tools

The RankSQL tools include a *Plan Builder*, an *Enumerator Visualizer*, and an *Execution Monitor*. The tools are convenient for system builders to understand, to debug, and to improve various components of a query engine. Therefore, their usefulness goes beyond demonstrating and visualizing the new features introduced by RankSQL. We briefly introduce theses tools below.

The Plan Builder (Figure 6.3) directly constructs a physical execution plan (bypassing the

```
<SEQSCAN nParamExec="0" scanrelid="1">
  <targetlist>
    <TARGETENTRY>
      <RESDOM resno="1" restype="1043" restypmod="84"
              resname="name" ressortgroupref="0"
              resorigtbl="34596" resorigcol="1"
              resjunk="false"/>
      <VAR varno="1" varattno="1" vartype="1043"
           vartypmod="84" varlevelsup="0"
           varnoold="1" varoattno="1"/>
    </TARGETENTRY>
  </targetlist>
</SEQSCAN>
```

Figure 6.4: A sample plan file.

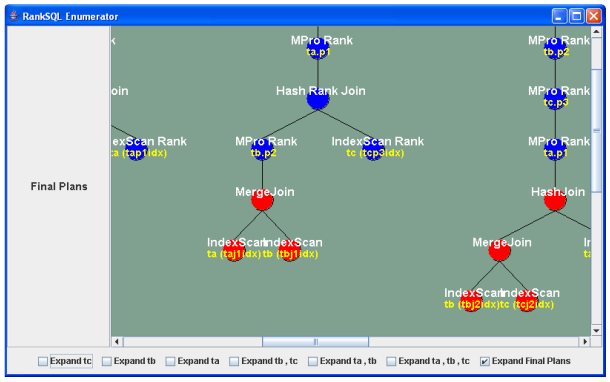


Figure 6.5: The Enumerator Visualizer.

query optimizer) from a plan file in XML format. For instance, Figure 6.4 shows a simple plan file that scans a table with ID 34596 in the catalog and returns tuples with the projected attribute *name*. The system builder provides a drag-and-drop GUI for editing the plan file visually. With this tool, system developers can experiment with plans that are not chosen by the query optimizer. In addition, it is useful in evaluating new query algorithms that are not yet incorporated in the optimizer's enumerator or cost model.

The Enumerator Visualizer (Figure 6.5) displays the information (*e.g.*, the estimated cost and the better plans with the same logical property) of every enumerated sub-plans including the pruned ones. The visualizer thus allows the system developers to compare these plans, identify the reason of pruning a plan, and figure out why and how the final execution plan is chosen.

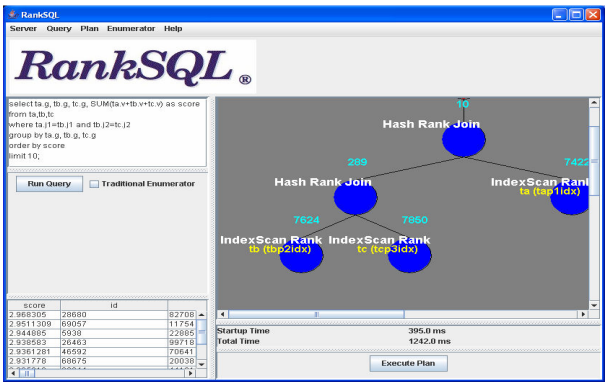


Figure 6.6: The RankSQL GUI with Execution Monitor.

The Execution Monitor (Figure 6.6) visualizes the execution of a physical plan, by showing the size of the intermediate results, the internal data structures for each operator, and the execution time. It also displays the cardinality information of every operators in a sub-plan.

6.2.2 An Example of Runtime Scenario

We describe one example scenario of using RankSQL, consisting of the following steps:

(1) The user connects to the database server from the menu option "Server" and types or loads a query (Figure 6.6).

(2) The user starts the enumerator visualizer, which displays the enumerated plans including the final execution plan (Figure 6.5).

(3) The user selects some query plan, for which a plan tree is shown in the GUI to illustrate its structure (Figure 6.6).

(4) The user executes the plan tree, and the execution monitor illustrates the execution process (Figure 6.6). For each operator, it shows how many tuples are produced and how many are output to its upper operator. Summary information such as the execution time is shown.

(5) The user can also use the plan builder (Figure 6.3) to create or load a plan file, upon which the execution plan is directly constructed without the optimizer. The resulting plan is shown and executed as described above.

(6) A table of the query results is shown in the GUI (Figure 6.6).

6.3 Challenges and Implementation Techniques

In this section we discuss some of the challenges in our system implementation and our solutions in addressing these challenges.

The Query Executor:
Realizing the new algorithms designed for ranking queries and ranking aggregate queries requires us to introduce new query operators into the execution engine. Many of the new operators rely on the efficient maintenance of a common data structure of *ranking queue*. A ranking queue of an operator may become big when a lot of seen tuples must be buffered before they can be passed to the upper operator. Therefore we often need to swap in and out the tuples in ranking queues between the memory and the disk. The tuples close to the bottom of a ranking queue should stay in the disk as long as possible. Meanwhile we should avoid swapping out the tuples at the top of the queue since they are more likely to stay at the top than those that have already ranked low.

Another data structure that is frequently used by these operators is hash table, for example in hash-rank join operator, which hashes join input tuples by join attributes. The hash table for the rank- and group-aware join operator in *rankagg* is even more complex, since the operator needs to maintain two types of hash tables, one for hashing join input tuples by join attributes, and another one for hashing join result tuples by grouping attributes.

Similarly, the *rankagg* operator needs to hash input tuples based on grouping attributes. In computing aggregate values, one aggregator (containing the initial value, the function for computing intermediate values, and the function for computing the final aggregate) must be maintained for each group. For conventional aggregate queries, there is no need to maintain multiple aggregators simultaneously, as the intermediate tuples being aggregated are preprocessed by either sorting or hashing, thus the groups are processed one by one. However, the aggregated tuples in *rankagg* are incrementally obtained and most of the groups are never fully materialized, which excludes the option of preprocessing these input tuples. We thus must maintain multiple aggregators at the same time, by using hash tables on the grouping attributes.

The Query Optimizer:

In optimizing ranking queries, we implement the 2-dimensional enumeration approach to enumerate the plan space along the dimensions on two logical properties, the joined tables as well as the processed raking predicates. However, this new optimizer must not interfere with the optimization of non-ranking queries, should still allow the option of using materialize-then-sort approach, and should not slow down the enumeration process for non-ranking queries. These requirements are rather smoothly satisfied with the help of the clearly modularized structure and great extensibility of POSTGRESQL query optimizer. Intermediate query plans with the same logical properties are grouped under the same entry during enumeration. The entries were relatively easily extended to accommodate the new dimension on ranking predicates.

On the other hand, we also need the ability to bypass the optimizer under certain circumstances. Instead of resorting to the optimizer in producing a physical query plan, it is often necessary to allow the usage of any query operator or plan. Such scenarios occur when we need to explore a new algorithm (*e.g.*, the operators in the *rankagg* framework) even before the corresponding optimization techniques (*e.g.*, cost models) are developed. We achieve this goal through several ways. One is to use a dummy cost model which assigns arbitrarily low cost to a new operator that we need to enforce, thus the new operator is always selected by the optimizer. Another way is to implement configuration parameters that enable and disable both existing and new operators. Finally, we can entirely bypass the optimizer by manually constructing a physical query plan and translating the plan into its internal representation that is directly executable by the query executor. For instance, the aforementioned plan builder (Section 6.2.1) provides convenience for such purpose.

The Query Parser:

The power of our new query executor and optimizer can be utilized only if the query parser can appropriately parse ranking queries and identify the ranking predicates. We extend the POSTGRESQL parser to be able to analyze queries with simple ranking functions. For instance, a ranking function with additions and multiplications is split into a set of ranking predicates that are

the basic operands of these additions and multiplications. Such a parser is still quite primitive. It can be beneficial to view a sub-expression as a single predicate instead of multiple ones, in order to support more general form of ranking functions as well as achieve better efficiency.

The Internal Representation of Tuple:

The introduction of our new ranking algebra into the system changes the underlying data model of the query engine. The internal representation of a tuple is changed as an inherent upper-bound score is associated with every tuple in the system. This requires us to modify not only the existing definitions of data structures, but also the communication protocols among various components in the system. For instance, the database server (*postmaster* in POSTGRESQL) sends the query results to the database client (such as the interactive *psql* command in POSTGRESQL and the application client using JDBC), by following the known communication sequence and format shared by the two parties. The communication protocols must be changed to allow the delivery of ranking scores.

Chapter 7

Related Work

This chapter reviews the work related to the thesis. We organize the related work into four categories (ranking in databases, aggregate query processing, query results organization, and inverse ranking and quantile queries) and discuss their connections to our work.

7.1 Ranking (top-k) Queries in Databases

Ranking (top-k) query has gained great interests in the database field recently. In middleware settings, various algorithms are proposed for rank aggregation on a set of objects, by merging multiple ranked lists [31, 76, 43, 32], or scheduling random accesses efficiently [12, 17], with the goal of minimizing number of accesses to objects. The works [17, 12] explore the concept of upper-bound scores that inspires us to formalize our ranking principle for relational top-k queries. A similar sampling approach was applied in [17] to schedule predicates only, whereas we extend the approach to estimate the cost of general ranking query plans.

In DBMSs, there have been several proposals to support answering top-k queries at application level or outside the core of query engines [19, 18, 91, 41, 42, 55, 100], or for supporting special types of ranking queries [75, 56]. Recently, supporting top-k queries inside the relational query engine, in terms of physical query operators, has been proved to be an efficient approach that treats ranking as a basic database functionality [13, 56, 57, 58]. A *stop* operator is proposed in [13] to limit the cardinality of intermediate and query result, either conservatively by integrity constraints or aggressively with the risk of restarting the query plan. The order supported by the stop operator is from columns of relations in SQL queries. Aggregation of multiple ranking criteria was not

considered.

In [57] a new operator is devised for supporting rank join query, where rank join predicates co-exist with Boolean join predicates. Instead of conducting normal join algorithms on Boolean join predicates, the rank-join operator progressively produces the join results. In [58] the relational query optimizer is further extended to utilize the rank-join operator in generating efficient query plans. We complement their work and together provide a systematic support of relational ranking queries, as we use rank-join as one of the rank-aware operators and at the same time supply an algebraic foundation of such support. Our dimensional enumeration framework enumerates plans by two dual logical properties to handle both scheduling of rank operators and join order selection, while [58] extends the "interesting order" (physical property) concept to deal with join enumeration only. The "interesting order" was also extended to support optimizing queries with expensive Boolean predicates [23]. The concept of our dimensional enumeration is general and extensible for more dimensions, including scheduling such Boolean predicates, union, and intersection operators.

With respect to the approach of extending query algebra, [62] proposes an algebra for capturing the semantics of preference queries. In [83] an algebra is proposed for expressing complex queries over *Web relations* that are used to model Web repositories. The algebra extension focuses on capturing the semantics of application-specific ranking and order relationships over Web pages and hyperlinks, instead of enabling efficient query processing. In [21], the authors prospectively proposed several alternative extensions of relational algebra that aim at supporting flexible ranking and scoring, for integrating DB and IR for text- and data-rich applications. In [82], the authors proposed to add a native score management system to object-relational databases, for querying Web metadata. They extend SQL with score-management clauses and propose the sideway-value algebra (SVA) for the extended queries.

7.2 Aggregate Query Processing

None of the top-k query algorithms [31, 32, 12, 17, 13, 30, 19, 57, 58, 21, 67] support ranking aggregate queries. The work closest to ours is [70], where ranking aggregates are computed over a specified range on some dimensions other than the grouping dimensions, by storing pre-computed partial aggregate information in the data cube. Therefore it can only support pre-determined aggregate function and aggregated expression, lacking the ability to support ad-hoc ranking aggregate queries. However, it is complementary to our work as it can be used to obtain the group sizes when there are selection conditions over the dimensional attributes, as mentioned in Section 3.4.2.

Order optimization [88] was proposed in relational query optimizer to avoid sorting, to minimize the number of sorting columns, to push down sort across joins, or to combine multiple sorts into a single one. *Eager aggregation* [98, 22] and *lazy aggregation* [99] were proposed to optimize **group-by** processing by functional dependencies. [25, 77] proposed algorithms and framework for combining order and grouping optimization. [92] extended eager aggregation in OLAP environment by utilizing special access methods.

Our approach of processing ad-hoc ranking aggregate queries differs from these works as follows: (1) We optimize for the retrieval of top k groups while the notion of ranking is not considered in previous works as they focus on full sorting; (2) We handle ad-hoc ranking aggregate conditions while previous works sort by table columns; and (3) We improve performance by pruning based on upper-bounds while previous works save sorting and grouping costs mainly by using functional dependencies.

Efficient computation of data cubes [2, 104, 85] focuses on sharing the computation across different cuboids instead of how to process a single cuboid, which corresponds to a **group-by**. Answering aggregate queries using materialized views was addressed in [44, 90].

Semantically similar to top-k aggregate queries, *iceberg queries* [33] retrieve only the groups that qualify under a Boolean condition (expressed in the **having** clause). The techniques in [33] are confined to single-table queries (joins have to be materialized beforehand) and **sum** or **count**

instead of general aggregate functions. The notion of iceberg queries was extended to *iceberg cubes* in [11, 49]. Iceberg cuboids with the same dimensional attributes involve the computation of an iceberg query, or essentially a ranking aggregate query. These works focused on pruning cuboids in an iceberg cube, while how to efficiently compute a cuboid was not considered. Hence, we consider our work to be complementary in evaluating iceberg cubes.

Online aggregation [51, 47] supports *approximate* query answering by providing running aggregates with statistical confidence intervals. We extend its index striding technique to support the group-aware and rank-aware scan operator.

7.3 Clustering and Query Results Organization

Various clustering algorithms exploited summary of data during clustering, *e.g.*, [103, 35]. In particular, there are grid-based clustering and data mining algorithms such as STING [94] and WaveCluster [87]. They pre-compute and store the grids beforehand. Our summary-based clustering shares the same insight of clustering by the unit of bucket instead of individual tuple. However, we emphasize on the need in our target applications to construct the grid *on-the-fly* for coping with dynamic Boolean conditions, clustering attributes, and ranking attributes that are specified at query time, and to integrate filtering, clustering, and ranking altogether.

Note that the top-k processing techniques in the literature may not be applicable for processing *ClusterRank*. It is well known that top-k algorithms are optimized for small k. As k increases, their performances degrade quickly and become worse than straightforward materialize-then-sort approach. In our case, we must get the top k tuples within each cluster, and some of them may be globally ranked low. For example, the houses of one region in general may be more expensive and smaller than the ones in other regions, therefore even the top houses in this region are ranked quite low globally. That does not mean the houses in the region are bad choices. In fact, the reality may be the opposite since, say, that region is safe and has beautiful scenery. Using top-k algorithms under this situation will not be beneficial.

An idea of using candidate buckets for pruning was applied in answering top-k queries [19]. However, they rely on static pre-collected multi-dimensional histogram, while we utilize data summary that is dynamically constructed using bitmap index.

An automatic method for categorizing query results (instead of clustering) is proposed in [14]. They perform categorization as post-processing after Boolean query results are obtained, with the focus on minimizing users' navigation overhead. They do not consider integration with ranking either. The idea of categorizing database query results is also exemplified by the products from *Endeca*.

The idea of combining clustering and ranking has been proposed for organizing the results of Web search engines (*e.g.*, *vivisimo.com*), as well as for information retrieval systems [64]. Our work investigates the problem in the setting of DBMSs, which has intrinsically different data and query processing model and thus presents significant new challenges.

7.4 Inverse Ranking and Quantile Queries

[55] studies using materialized ranked results to answer ranking queries. [29] presents a more general approach of using views in answering top-k queries. It also uses linear programming during query processing. However, it focuses on small number of top k answers, and thus it is less important to build efficient access method (such as the index) for retrieving previous query results.

[101] studies quantile retrieval on multi-dimensional data. However, quantiles in that work are based on a single measure associated with multi-dimensional index structure, instead of ranking function. Therefore it does not incur the problem of mismatching between indexed attributes and ranking attributes. Moreover, it does not consider the integration with Boolean query conditions.

There were quite some works on computing quantiles in databases and stream data systems [3, 5, 37, 71, 6, 28, 27]. However, they are fundamentally different from the quantile queries in this work. *First*, they focus on the quantile of a set of data items, where the values of the items themselves give their ranks. To the contrary, we study general quantile queries, where database records

are ranked by ranking functions that involve multiple attributes or even multiple tables. *Second*, our quantile queries are in the context of general database queries, whereas they do not consider the integration with Boolean query conditions. *Finally*, from the perspective of application domains, the quantile queries in this thesis are for flexible data retrieval, exploration, and analysis. The previous works do not consider such applications. They use quantiles of data for query optimization, result size estimation, association rule mining, data cleaning, and data partitioning.

Chapter 8

Conclusion and Future Agenda

Towards enabling data retrieval, this thesis focuses on how to *fundamentally integrate ranking into databases* and how to *enable retrieval mechanisms beyond just ranking*. This section summarizes the contributions and discusses some open issues that warrant further research.

We introduced our RankSQL system for full support of ranking as a first-class operation in real-world database systems. As the foundation of our work, we present the key insight that ranking is another logical property of data, parallel to the "membership" property. Centering around this insight, we *first* introduced a novel and general framework for supporting ranking in relational query engines based on extending the relational algebra. The extended rank-relational algebra captures the ranking property with rank-relational model and introduces new and extended operators to fully express top-k queries. We also defined a set of algebraic laws for rewriting, hence optimizing, top-k query plans. *Second*, we presented a pipelined and incremental execution model of ranking query plans, by realizing the fundamental *ranking principle* in the extended algebra, thus enabling efficient processing of ranking queries. *Third*, based on the insight of the duality between ranking and membership properties, we introduced a generalized rank-aware optimization framework that defines ranking as an additional plan enumeration dimension beyond enumerating joins and generates the full space of rank-aware query evaluation plans. For practical purposes, we introduced heuristics that limit the generated space. Moreover, we introduced a novel technique for estimating the cardinality of top-k operations, hence, providing an effective plan pruning mechanism to get efficient ranking query plans. We presented the experimental results on our initial implementation of the RankSQL system.

Upon the algebraic foundation, we introduced a principled and systematic framework to sup-

port ad-hoc top-k (ranking) aggregate queries efficiently. We developed the Upper-Bound Principle that dictates the requirements of early pruning, and the Group-Ranking and Tuple-Ranking Principles that dictate the group-ordering and tuple-ordering requirements. The principles together realize optimal aggregate query processing. Guided by the principles, we proposed an execution framework for exploiting the principles. We addressed the challenges in applying the principles and implementing the new query operators. The experiment results validate our framework by showing significant performance improvement. To the best of our knowledge, this is the first piece of work that provides efficient support of ad-hoc top-k aggregates. Thus the techniques address a significant research challenge and can be very useful in many decision support applications.

Beyond ranking, we proposed to generalize **group-by** to enable fuzzy grouping (clustering in particular) of database query results, and to integrate grouping with ranking and further with Boolean filtering, for supporting structured data retrieval applications. We defined a new type of *ClusterRank* queries for this purpose. We designed a summary-based framework to meet the challenges in supporting such integration. We realized the framework by utilizing bitmap index to construct the summary on-the-fly, and to efficiently integrate Boolean filtering, clustering, and ranking altogether. Experimental study with our implementation shows that the framework achieves orders of magnitude better efficiency than the straightforward approach available in current databases, and at the same time it maintains high clustering quality. To the best of our knowledge, this work is the first to propose such generalization of fuzzy grouping and integration with ranking within DBMSs.

We also proposed a new type of inverse ranking queries, that obtain the ranks of given query objects among certain context objects. Such queries are useful in supporting data retrieval and exploration, thus can be important in many applications. We further developed a framework for processing these queries, and discussed several alternative methods within this general framework. Some of these methods utilize common data structures existing in current database systems and some others are based on the new data structures proposed. We analytically studied the cost model of these methods, and empirically compared them with each other as well, to understand the trade-off in applying these methods. We also compared these methods with the traditional straightfor-

ward method, and verified that our method is much more efficient. To the best of our knowledge, ours is the first work that studies inverse ranking queries. We believe that we have conducted a thorough investigation on the definition of the novel queries, the design of efficient processing methods for such queries, and the experimental evaluation of the methods.

Towards the ultimate goal of building next-generation data retrieval systems, many interesting and challenging problems remain open. *First*, we want to explore more retrieval mechanisms. *Second*, my thesis work focused on retrieval in traditional structured databases. However, as abundant data nowadays exist in a space of heterogeneous and interconnected sources, it is imperative to use auxiliary information from related sources for robust ranking of database records. *Finally*, to take full advantages of the processing techniques developed for retrieval queries, we must have a general and robust query optimizer for such queries.

Data Retrieval and Exploration Primitives beyond Ranking: For flexible, intuitive, and explorative querying of databases, we must look for novel retrieval mechanisms that go beyond ranking, such as the fuzzy grouping in Chapter 4 and the inverse ranking in Chapter 5. For such purpose, it is promising to investigate the primitive operations in information retrieval, data mining, and user interfaces. For instance, can we use association or correlation in relaxing restrictive query conditions? Can we automatically produce a navigation "map" that guides users to find information in databases? Supporting such alien concepts inside databases is surely challenging. We must integrate them with traditional Boolean constructs. We must invent efficient processing methods for them. Moreover, in supporting data exploration, we must look for innovative techniques of database user interfaces and data visualization.

Optimizing Retrieval Queries: While most existing works on ranking focus on query processing, there is a serious lack of general and accurate optimization techniques for ranking queries, and more generally for retrieval queries. For instance, in cost estimation, a key component in query optimizer, most existing works only deal with special cases under unrealistic assumptions such as uniform data distribution. We must tackle this challenge by inventing non-traditional database

statistics and analytical cost models that utilize the statistics. For example, results from order statistics may guide us in developing such statistics.

Context-Aware Ranking: Database tuples and queries are tightly connected to their "context" residing in diverse information sources outside the databases. Therefore instead of treating tuples in isolation, we must utilize the context for robust ranking. Such context includes *data context* (*e.g.*, textual annotations, data history and provenance) and *query context* (*e.g.*, personalized information, query history and workload). For instance, we can provide personalized retrieval by incorporating user profiles into ranking. There are many challenges in realizing context-aware ranking. We must combine context with tuples themselves in determining ranking; Context information from heterogeneous sources can be incomplete, imprecise, or even conflicting; Finally we must design efficient ways to coordinate query processing across multiple data sources.

Ranking by Object Relationships: In many non-traditional applications, we can utilize ranking criteria that are unavailable in conventional databases. For instance, object relationships (*e.g.*, co-recommendations, co-occurrences of text descriptions) are powerful for recommending top objects (*e.g.*, videos, music, and Blog articles) to users. While similar ideas were applied in collaborative filtering, current techniques have difficulties in coping with the dynamic nature of large systems. For example, as a video sharing Web site accepts new videos continuously, how to keep up with the scale and make *active* recommendations to each user individually?

Appendix

Proofs of Properties and Theorems

1 Proof of Property 2

Property 2 (Best-Possible Goal): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^{min} be its minimal tuple depth, *i.e.*, the number of tuples to retrieve from g before it can be pruned from further processing, or otherwise determined to be in the top-k groups. The H_g^{min} is the smallest number of tuples from g that makes the maximal possible score of g to be below θ, *i.e.*,

$$H_g^{min} = \min\{|\mathcal{I}_g| \overline{F}_{\mathcal{I}_g}[g] < \theta, \mathcal{I}_g \subseteq g\}, \tag{1}$$

or otherwise H_g^{min}=$|g|$ if such a depth does not exist. ■

Proof: By definition, H_g^{min} is the number of tuples retrieved from g when the query processing stops, *i.e.*, g is pruned from further processing or otherwise determined to be in the top-k groups.

If g belongs to the real top k groups, say $\mathcal{K}'$, g must be completely processed to obtain its exact aggregate. That is, H_g^{min}=$|g|$.

For any $g \notin \mathcal{K}'$, suppose a subset of tuples $\mathcal{I}_g \subseteq g$ have been accessed. We now prove the minimal tuple depth H_g^{min} is the smallest $|\mathcal{I}_g|$ that makes $\overline{F}_{\mathcal{I}_g}[g] < \theta$. On the one hand, H_g^{min} cannot be smaller than such *smallest* $|\mathcal{I}_g|$, otherwise $\overline{F}_{\mathcal{I}_g}[g] > \theta$. (Note that we assume no ties in scores.) Under such situation, no algorithm can exclude g from further processing since g still has a chance to achieve better than the lowest top-k group. On the other hand, H_g^{min} does not need to be larger than such $|\mathcal{I}_g|$, because we already have $\overline{F}_{\mathcal{I}_g}[g] < \theta$ when $\mathcal{I}_g$ tuples from g are retrieved.

Under such situation, g does not have a chance to get into the final top k groups. An algorithm that has obtained the final top k groups can safely exclude g from further processing. There H_g^{min} is the smallest possible tuple depth. ■

2 Proof of Property 3

Property 3 (Must-Have Information): For any group g, with a trivial upper-bound $\overline{F}_{\mathcal{I}_g}[g]= +\infty$ under every $\mathcal{I}_g$, H_g^{min}=$|g|$. ■

Proof: If the upper-bound of a g is always infinity, no algorithm can prune g early since g always has a chance to be one of the final top k groups. That is, all the tuples in g must be accessed, *i.e.*, H_g^{min}=$|g|$. ■

3 Proof of Property 4

Property 4 (Group-Ranking Principle): Let g_1 be any group in the current top-k ranked by maximal-possible scores $\overline{F}$ and g_2 be any group not in the current top-k. We have 1) g_1 must be further processed if g_1 is not fully evaluated, 2) it may not be necessary to further process g_2 even if g_2 is not fully evaluated, and 3) the current top-k groups are the query answers if they are all fully evaluated. ■

Proof: Suppose at any moment, the *current* top-k groups are $\mathcal{K}$, such that $\overline{F}_{\mathcal{I}_{g_1}}[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, $\forall g_1 \in \mathcal{K}$ and $\forall g_2 \notin \mathcal{K}$. Assume there is no tie in scores since ties can be broken by the "tie-breaker" mentioned in Section 3.1.

For any incomplete group $g_1 \in \mathcal{K}$ ($\mathcal{I}_{g_1} \subset g_1$), further processing g_1 is necessary, otherwise g_1 remains belonging to $\mathcal{K}$ (based on Eq. 3.3, $\overline{F}$ never increases as more tuples are obtained) and incomplete, resulting in that it is impossible to get the real top k groups and their exact aggregates. (g_1 may be one of the real top k groups since it remains belonging to $\mathcal{K}$; and the exact aggregate of g_1 can not be obtained since it is not fully evaluated).

Given any such $g_1 \in \mathcal{K}$ and $g_2 \notin \mathcal{K}$, since $\overline{F}_{\mathcal{I}_{g_1}}[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, whatever score g_2 can achieve, g_1 can possibly do better. Thus, further processing g_2 (when g_2 is incomplete) may not be necessary, since $\mathcal{K}$ may turn out to be the real top k groups, *i.e.*, $F[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, $\forall g_1 \in \mathcal{K}$. Further processing of g_2 can become necessary only if we further process some incomplete group $g_1 \in \mathcal{K}$ such that g_2 belongs to the updated $\mathcal{K}$.

When all the groups in $\mathcal{K}$ are complete, we can sufficiently declare $\mathcal{K}$ as the real top k groups, because $F[g_1] = \overline{F}_{\mathcal{I}_{g_1}}[g_1] > \overline{F}_{\mathcal{I}_{g_2}}[g_2]$, $\forall g_1 \in \mathcal{K}$ and $\forall g_2 \notin \mathcal{K}$. ■

4 Proof of Property 5

We present the following lemma before the proof.

Lemma 1: With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. At any moment, suppose $\mathcal{K}$ is the *current* top k groups ranked by maximal-possible scores $\overline{F}$, we have $\overline{F}[g] \geq \theta$, $\forall g \in \mathcal{K}$. ■

Proof: Suppose the set of the real top k groups is $\mathcal{K}'$. For any $g' \in \mathcal{K}'$, at any moment, $\overline{F}[g'] \geq F[g'] \geq \theta$, and thus there are at least k groups with maximal-possible scores over θ. Therefore for the current top k groups $\mathcal{K}$ at the moment, $\overline{F}[g] \geq \theta$, $\forall g \in \mathcal{K}$. ■

Property 5 (Tuple-Ranking Principle): With respect to a ranking aggregate F= $G(T)$, let the lowest top-k group score be θ. For any group g, let H_g^α be the tuple depth with respect to tuple order α: $t_1 \rightarrow \cdots \rightarrow t_n$, when the inter-group ordering follows Requirement 2.

- (**Order Independence**) The depth H_g^α depends on only α (the order of this group) and θ (the global threshold), and not on the order of other groups. Specifically, H_g^α is the smallest depth l of sequence α that makes the maximal possible score of g to be below θ, *i.e.*,

$$H_g^\alpha = \min_{l \in [1:n]} \{l | \overline{F}_{\{t_1,\ldots,t_l\}}[g] < \theta\}, \tag{2}$$

  or otherwise H_g^α=n if such a depth does not exist.

- (T**-based Ranking**) To find the optimal order α that results in the minimum H_g^α, *i.e.*, H_g^α=H_g^{min}, we need to only consider the class of orders T*-desc/asc* =

$$\{\alpha : t_1 \rightarrow \cdots \rightarrow t_n | \begin{array}{l} \text{either } T[t_i] \geq T[t_j] \forall j > i \text{ (from top);} \\ \text{or } \quad T[t_i] \leq T[t_j] \forall j > i \text{ (from bottom).} \end{array} \forall t_i\}. \tag{3}$$

■

Proof:

Order Independence: By definition, H_g^α is the number of tuples retrieved from g when the query processing stops. If Requirement 2 is followed, by the Stop condition, when we stop and conclude the real top k groups, say $\mathcal{K}'$, g is complete $\forall g \in \mathcal{K}'$ and therefore H_g^α=n.

For any $g \notin \mathcal{K}'$, if it has been accessed to some depth l, $t_1, \ldots, t_l$ are the accessed tuples since the order is α: $t_1 \rightarrow \cdots \rightarrow t_n$. We now prove the tuple depth H_g^α is the smallest l that makes $\overline{F}_{\{t_1,\ldots,t_l\}}[g] < \theta$. On the one hand, H_g^α cannot be smaller than such l, otherwise $\overline{F}_{\{t_1,\ldots,t'_l\}}[g] > \theta$ (say, H_g^α=$l' < l$). (Note that we assume no ties in scores.) Under such situation, we could not have concluded $\mathcal{K}'$ as the real top k groups if Requirement 2 is followed, since the Progressive condition would require to further process some incomplete group such as g. (Remember θ is the lowest $F[g]$ among groups in $\mathcal{K}'$.) On the other hand, H_g^α cannot be larger than such l, because we already have $\overline{F}_{\{t_1,\ldots,t_l\}}[g] < \theta$ when l tuples from g are retrieved. Under such situation, g cannot get into the current top k groups $\mathcal{K}$ anymore (Lemma 1). By the Progressive condition of Requirement 2, g cannot get any chance to be further processed.

T**-based Ranking:** We prove that for any *rest* order, there is always a better order in T*-desc/asc*.

As the complement of T*-desc/asc*, the space of all orders in the *rest* class is defined as

$\Omega_r = \{\alpha' | \alpha' : t_1 \rightarrow \cdots \rightarrow t_n, \exists_{x,j,k,j>x \wedge k>x} T[t_x] > T[t_j] \wedge T[t_x] < T[t_k]\}$.

That is, given any $\alpha' \in \Omega_r$, there exists at least one "next" tuple obtained at some step that is neither the highest nor the lowest score among the unseen tuples. (Without such "middle" tuple, α' becomes a T*-desc/asc* order.)

There can be multiple instances of such "middle" tuple t_x at various steps of the tuple sequence of α'. Without loss of generality, let us focus on the first instance of such t_x. Before that retrieval, suppose $\mathcal{I}_g$ is the set of retrieved tuples and the maximal-possible score for unseen tuples is $\overline{T}_{\mathcal{I}_g}$

(Eq. 3.3). After the retrieval, the maximal-possible score of unseen tuples is unchanged, *i.e.*, $\overline{T}_{\mathcal{I}_g \cup \{t_x\}}$=$\overline{T}_{\mathcal{I}_g}$. Therefore

$$\overline{F}_{\mathcal{I}_g \cup \{t_x\}}[g] = G\left(\begin{array}{lll} T_i = T[t_i] & \text{if } t_i \in \mathcal{I}_g \text{ (\textit{seen tuples})}; & \\ \{T_i|\ T_i = T[t_i] = T[t_x] & \text{if } t_i = t_x \text{ (\textit{just retrieved})}; & \forall t_i \in g\} \\ T_i = \overline{T}_{\mathcal{I}_g \cup \{t_x\}} = \overline{T}_{\mathcal{I}_g} & \text{otherwise (\textit{unseen tuples})}. & \end{array}\right). \tag{4}$$

However, if the one with the lowest score among unseen tuples, $t_\perp = argmin_{t \in g - \mathcal{I}_g} T[t]$, instead of t_x was retrieved, we have $\overline{F}_{\mathcal{I}_g \cup \{t_\perp\}}[g] < \overline{F}_{\mathcal{I}_g \cup \{t_x\}}[g]$ according to the following Eq. 5, which is the same as Eq. 4 except that $T[t_\perp] < T[t_x]$.

$$\overline{F}_{\mathcal{I}_g \cup \{t_\perp\}}[g] = G\left(\begin{array}{lll} T_i = T[t_i] & \text{if } t_i \in \mathcal{I}_g \text{ (\textit{seen tuples})}; & \\ \{T_i|\ T_i = T[t_i] = T[t_\perp] & \text{if } t_i = t_\perp \text{ (\textit{just retrieved})}; & \forall t_i \in g\} \\ T_i = \overline{T}_{\mathcal{I}_g \cup \{t_\perp\}} = \overline{T}_{\mathcal{I}_g} & \text{otherwise (\textit{unseen tuples})}. & \end{array}\right). \tag{5}$$

Now we show that we can convert any $\alpha_0 \in \Omega_r$ into a T-*desc/asc* order α_m by a series of transformations, $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \ldots \Rightarrow \alpha_m$, and $H_g^{\alpha_m} \leq H_g^{\alpha_0}$. In each transformation from α_i to α_{i+1}, we find the first instance of such "middle" tuple t_x in α_i, and swap the position of t_x and $t_\perp$, *i.e.*, $\alpha_i : \langle \ldots, t_x, \ldots, t_\perp, \ldots \rangle \Rightarrow \alpha_{i+1} : \langle \ldots, t_\perp, \ldots, t_x, \ldots \rangle$. According to Eq. 4 and Eq. 5, the value of $\overline{F}_{\mathcal{I}_g}[g]$ at each step of α_{i+1} is equal to or smaller than that of α_i, therefore $H_g^{\alpha_{i+1}} \leq H_g^{\alpha_i}$ based on Eq. 3.4, thus $H_g^{\alpha_m} \leq H_g^{\alpha_0}$. Moreover, for a group with n tuples, such transformation ends in at most $n-1$ steps, therefore we will reach a better hybrid order. The reason is as follows. Since t_x is the first instance of "middle" tuple in α_i, the retrieved tuple at each step before t_x is the one with either the highest or the lowest score among the unseen tuples. Suppose there are s tuples before t_x in the sequence of α_i. Then after the transformation from α_i to α_{i+1}, there are at least $s+1$ tuples before the first instance of "middle" tuple in α_{i+1}, and finally there is no such instance in α_m. ■

5 Proof of Theorem 1

Theorem 1 (Optimal Aggregate Processing): If query processing follows Requirements 2 and 3, the number of tuples processed across all groups, *i.e.*, $\Sigma_g H_g$, is the minimum possible for query answering, *i.e.*, $\Sigma_g H_g^{min}$. ■

Proof: Among all the tuple orders that follow Requirement 2, for any group g, according to Property 5, the number of processed tuples H_g is minimized when Requirement 3 is followed as well, *i.e.*, $H_g = H_g^{\alpha}$ where α is the optimal intra-group order for g.

Moreover, this H_g^{α} is indeed the smallest number of g's tuples to retrieve under any total tuple order, *i.e.*, $H_g^{\alpha} = H_g^{min}$, because any smaller number would result in $\overline{F}[g] \geq \theta$ so that we can neither get the exact aggregate of g (if g belongs to the real top k groups) nor conclude g does not belong to the real top k. In other words, the optimality of α is independent from whether Requirement 2 is followed or not because α is the optimal among all possible orders of retrieving g's tuples. In summary, the minimal total number of retrieved tuples is $\Sigma_g H_g^{min}$ and this minimum is achieved when both Requirement 2 and 3 are followed. ■

6 Proof of Property 6

Property 6: With respect to a relation T, a ranking function $\mathcal{F}(R)$, and k, suppose the top k tuples are T_k. The set of candidate buckets $\mathcal{B}'$ obtained by the algorithm in Figure 4.4 is both *correct*: $\mathcal{B}'$ contains all the top k tuples, *i.e.*, $T_k \subseteq T_{\mathcal{B}'}$; and *optimal*: $\mathcal{B}'$ is the smallest set of buckets that contain T_k, *i.e.*, $\forall \mathcal{B}''$ *s.t.* $\exists B_i \in \mathcal{B}'$ and $B_i \notin \mathcal{B}''$, there exists an instance of T *s.t.* $T_k \nsubseteq T_{\mathcal{B}''}$. ■

Proof: $\forall t \in T$, we use $B(t)$ to denote the bucket which t falls into in $\mathcal{G}(T, R)$. With respect to a bucket B_i, we use $T_{B_i}^{+}$ to denote the set of tuples that belong to the buckets whose lower-bound scores are higher than or equal to the upper-bound score of B_i, and $T_{B_i}^{-}$ to denote the remaining tuples ($T - T_{B_i}^{+}$), *i.e.*, $T_{B_i}^{+}$=$\{t | t \in T$ and $lower_{B(t)} \geq upper_{B_i}\}$ and $T_{B_i}^{-}$=$\{t | t \in T$ and $lower_{B(t)} < upper_{B_i}\}$.

Correctness: We prove the correctness by proving $\forall t \in T_k$, $B(t) \in \mathcal{B}'$, by contradiction. If $B(t) \notin \mathcal{B}'$, then $|T^+_{B(t)}| > k$ and $\forall t' \in T^+_{B(t)}$, $\mathcal{F}(R)[t'] \geq lower_{B(t')} \geq upper_{B(t)} > \mathcal{F}(R)[t]$ [1] (step 4-8 in Figure 4.4). This means there exists at least k tuples whose scores are higher than that of t, thus $t \notin T_k$, contradicting $t \in T_k$.

Optimality: Consider any $\mathcal{B}''$ *s.t.* $B_i \in \mathcal{B}'$ and $B_i \notin \mathcal{B}''$. Since $B_i \in \mathcal{B}'$, we have $|T^+_{B_i}| < k$ and $|T^-_{B_i}| > |T| - k$. Use $max(T^-_{B_i})$ to denote the maximal score among the tuples in $T^-_{B_i}$. We prove the optimality by proving there exists an instance of T *s.t.* $\exists t$, $t \in T_k$ and $B(t) = B_i$. One such instance is: $\forall t' \in T^-_{B_i}$, $\mathcal{F}(R)[t'] = lower_{B(t')}$, and $\exists t$, $B(t) = B_i$ and $\mathcal{F}(R)[t] = \frac{1}{2} \times (max(T^-_{B_i}) + upper_{B_i})$, thus $\mathcal{F}(R)[t] > \mathcal{F}(R)[t']$, $\forall t' \in T^-_{B_i}$. According to $|T^-_{B_i}| > |T| - k$, $t \in T_k$. Since $B(t) = B_i$ and $B_i \notin \mathcal{B}''$, therefore $T_k \nsubseteq T_{\mathcal{B}''}$. ■

[1] Note that $upper_{B(t)} > \mathcal{F}(R)[t]$ since the ranges are defined as left-end closed and right-end open, *cf.* Section 4.4.2.

References

[1] Foto N. Afrati and Rada Chirkova. Selecting and using views to compute aggregate queries (extended abstract). In *Proceedings of the 10th International Conference on Database Theory*, pages 383–397, 2005.

[2] Sameet Agarwal, Rakesh Agrawal, Prasad Deshpande, Ashish Gupta, Jeffrey F. Naughton, Raghu Ramakrishnan, and Sunita Sarawagi. On the computation of multidimensional aggregates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 506–521, 1996.

[3] Rakesh Agrawal and Arun Swami. A one-pass space-efficient algorithm for finding quantiles. In *Proceedings of the 7th International Conference on Management of Data, COMAD*, 1995.

[4] Sanjay Agrawal, Surajit Chaudhuri, Gautam Das, and Aristides Gionis. Automated ranking of database query results. In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.

[5] Khaled Alsabti, Sanjay Ranka, and Vineet Singh. A one-pass algorithm for accurately estimating quantiles for disk-resident data. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 346–355, 1997.

[6] Arvind Arasu and Gurmeet Singh Manku. Approximate counts and quantiles over sliding windows. In *Proceedings of the 23rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 286–296, 2004.

[7] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 407–418, 2004.

[8] Pavel Berkhin. Survey of clustering data mining techniques. Technical report, Accrue Software, San Jose, CA, 2002.

[9] D. Bertsimas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 2nd edition, 1995.

[10] D. Bertsimas and J. N. Tsitsiklis. *Introduction to Linear Optimization*. Athena Scientific, Belmont, Massachusetts, 1997.

[11] Kevin Beyer and Raghu Ramakrishnan. Bottom-up computation of sparse and iceberg cube. *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 359–370, 1999.

[12] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 369–380, 2002.

[13] Michael J. Carey and Donald Kossmann. On saying "enough already!" in SQL. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 219–230, 1997.

[14] Kaushik Chakrabarti, Surajit Chaudhuri, and Seung-won Hwang. Automatic categorization of query results. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 755–766, 2004.

[15] Donald D. Chamberlin, Morton M. Astrahan, Mike W. Blasgen, Jim Gray, W. Frank King III, Bruce G. Lindsay, Raymond A. Lorie, James W. Mehl, Thomas G. Price, Gianfranco R. Putzolu, Patricia G. Selinger, Mario Schkolnick, Donald R. Slutz, Irving L. Traiger, Bradford W. Wade, and Robert A. Yost. A history and evaluation of system R. *Communications of the ACM (CACM)*, 24(10):632–646, 1981.

[16] Chee Yong Chan and Yannis E. Ioannidis. An efficient bitmap encoding scheme for selection queries. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, 1999.

[17] Kevin Chen-Chuan Chang and Seung-won Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data*, pages 346–357, 2002.

[18] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R. Smith. The onion technique: indexing for linear optimization queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pages 391–402, 2000.

[19] Surajit Chaudhuri and Luis Gravano. Evaluating top-*k* selection queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 397–410, 1999.

[20] Surajit Chaudhuri, Rajeev Motwani, and Vivek Narasayya. On random sampling over joins. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 263–274, 1999.

[21] Surajit Chaudhuri, Raghu Ramakrishnan, and Gerhard Weikum. Integrating DB and IR technologies: What is the sound of one hand clapping? In *Proceedings of the 2nd Biennial Conference on Innovative Data Systems Research*, pages 1–12, 2005.

[22] Surajit Chaudhuri and Kyuseok Shim. Including Group-By in query optimization. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 354–366, 1994.

[23] Surajit Chaudhuri and Kyuseok Shim. Optimization of queries with user-defined predicates. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 87–98, 1996.

[24] Danette Chimenti, Ruben Gamboa, and Ravi Krishnamurthy. Towards on open architecture for $\mathcal{LDL}$. In *Proceedings of the 15th International Conference on Very Large Data Bases*, pages 195–203, 1989.

[25] Jens Claussen, Alfons Kemper, Donald Kossmann, and Christian Wiesner. Exploiting early sorting and early partitioning for decision support query processing. *VLDB J.*, 9(3), 2000.

[26] Sara Cohen, Werner Nutt, and Alexander Serebrenik. Rewriting aggregate queries using views. In *Proceedings of the 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 155–166, 1999.

[27] Graham Cormode, Minos N. Garofalakis, S. Muthukrishnan, and Rajeev Rastogi. Holistic aggregates in a networked world: Distributed tracking of approximate quantiles. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 25–36, 2005.

[28] Graham Cormode, Flip Korn, S. Muthukrishnan, and Divesh Srivastava. Effective computation of biased quantiles over data streams. In *Proceedings of the 21st International Conference on Data Engineering*, pages 20–31, 2005.

[29] Gautam Das, Dimitrios Gunopulos, Nick Koudas, and Dimitris Tsirogiannis. Answering top-k queries using views. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 451–462, 2006.

[30] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 411–422, 1999.

[31] Ronald Fagin. Combining fuzzy information from multiple systems. In *Proceedings of the 15th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 216–226, 1996.

[32] Ronald Fagin, Amnon Lote, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 102–113, 2001.

[33] Min Fang, Narayanan Shivakumar, Hector Garcia-Molina, Rajeev Motwani, and Jeffrey D. Ullman. Computing iceberg queries efficiently. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 299–310, San Francisco, CA, USA, 1998.

[34] F. Farnstrom, J. Lewis, and C. Elkan. Scalability for clustering algorithms revisited. 2(1):51–57, August 2000.

[35] Venkatesh Ganti, Johannes Gehrke, and Raghu Ramakrishnan. CACTUS - clustering categorical data using summaries. In *KDD*, pages 73–83, 1999.

[36] Martin Gavrilov, Dragomir Anguelov, Piotr Indyk, and Rajeev Motwani. Mining the stock market (extended abstract): which measure is best? In *SIGKDD*, pages 487–496, 2000.

[37] A. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. How to summarize the universe: Dynamic maintenance of quantiles. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 454–465, 2002.

[38] Goetz Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[39] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the 9th International Conference on Data Engineering*, pages 209–218, 1993.

[40] Jim Gray, Surajit Chaudhuri, Adam Bosworth, Andrew Layman, Don Reichart, Murali Venkatrao, Frank Pellow, and Hamid Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *J. Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[41] Sudipto Guha, Dimitrios Gunopulos, Nick Koudas, Divesh Srivastava, and Michail Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 778–789, 2003.

[42] Sudipto Guha, Nick Koudas, Amit Marathe, and Divesh Srivastava. Merging the results of approximate match operations. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 636–647, 2004.

[43] U. Güntzer, W. Balke, and W. Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the 26th International Conference on Very Large Data Bases*, pages 419–428, 2000.

[44] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate query processing in data warehousing environments. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 358–369, 1995.

[45] Himanshu Gupta, Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Index selection for OLAP. In *Proceedings of the 13th International Conference on Data Engineering*, pages 208–219, 1997.

[46] Antonin Guttman. R-trees: a dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[47] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, pages 287–298, 1999.

[48] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann, New York, 2000.

[49] Jiawei Han, Jian Pei, Guozhu Dong, and Ke Wang. Efficient computation of iceberg cubes with complex measures. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 1–12, 2001.

[50] Venky Harinarayan, Anand Rajaraman, and Jeffrey D. Ullman. Implementing data cubes efficiently. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 205–216, 1996.

[51] Joseph M. Hellerstein, Peter J. Haas, and Helen J. Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 171–182, 1997.

[52] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 562–573, 1995.

[53] Joseph M. Hellerstein and Michael Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 267–276, 1993.

[54] Wei Hong and Michael Stonebraker. Optimization of parallel query execution plans in XPRS. In *ICPDIS*, pages 218–225, 1991.

[55] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. 2001.

[56] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 950–961, 2002.

[57] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join quereis in relational databases. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 754–765, 2003.

[58] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey S. Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, pages 203–214, 2004.

[59] Yannis E. Ioannidis. The history of histograms (abridged). In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 19–30, 2003.

[60] A. K. Jain, M. N. Murty, and P. J. Flynn. Data clustering: a review. *ACM Computing Surveys*, 31(3):264–323, 1999.

[61] Kittisak Kerdprasop, Nittaya Kerdprasop, and Pairote Sattayatham. Weighted k-means for density-biased clustering. In *DaWaK*, pages 488–497, 2005.

[62] Werner Kießling. Foundations of preferences in database systems. In *Proceedings of the 28th International Conference on Very Large Data Bases*, pages 311–322, 2002.

[63] Bjornar Larsen and Chinatsu Aone. Fast and effective text mining using linear-time document clustering. In *SIGKDD*, pages 16–22, 1999.

[64] A. Leuski and J. Allan. Improving interactive retrieval by combining ranked lists and clustering. In *RIAO*, pages 665–681, 2000.

[65] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Inverse ranking queries. *In submission*.

[66] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. Supporting ad-hoc ranking aggregates. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, pages 61–72, 2006.

[67] Chengkai Li, Kevin Chen-Chuan Chang, Ihab F. Ilyas, and Sumin Song. RankSQL: Query algebra and optimization for relational top-k queries. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, pages 131–142, 2005.

[68] Chengkai Li, Mohamed A. Soliman, Kevin Chen-Chuan Chang, and Ihab F. Ilyas. RankSQL: Supporting ranking queries in relational database management systems. In *Proceedings of the 31st International Conference on Very Large Data Bases (demonstration description)*, pages 1342–1345, 2005.

[69] Chengkai Li, Min Wang, Lipyeow Lim, Haixun Wang, and Kevin Chen-Chuan Chang. Supporting ranking and clustering as generalized Order-By and Group-By. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 127–138, 2007.

[70] Hua-Gang Li, Hailing Yu, Divyakant Agrawal, , and Amr El Abbadi. Ranking aggregates. Technical report, UCSB, July 2004.

[71] Xuemin Lin, Hongjun Lu, Jian Xu, and Jeffrey Xu Yu. Continuously maintaining quantile summaries of the most recent n elements over a data stream. In *Proceedings of the 20th International Conference on Data Engineering*, pages 362–374, 2004.

[72] C. Mohan, Donald J. Haderle, Yun Wang, and Josephine M. Cheng. Single table access using multiple indexes: Optimization, execution, and concurrency control techniques. In *Proceedings of the 2nd International Conference on Extending Database Technology*, pages 29–43, March 1990.

[73] Bruce Momjian. PostgreSQL internals through pictures. Software Research Associates, December 2001.

[74] Fujiki Morii. A generalized k-means algorithm with semi-supervised weight coefficients. In *ICPR*, pages 198–201, 2006.

[75] Apostol Natsev, Yuan-Chi Chang, John R. Smith, Chung-Sheng Li, and Jeffrey Scott Vitter. Supporting incremental join queries on ranked inputs. In *Proceedings of the 27th International Conference on Very Large Data Bases*, pages 281–290, 2001.

[76] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the 15th International Conference on Data Engineering*, pages 22–29, 1999.

[77] Thomas Neumann and Guido Moerkotte. A combined framework for grouping and order optimization. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 960–971, 2004.

[78] Jürg Nievergelt, Hans Hinterberger, and Kenneth C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[79] Patrick E. O'Neil. Model 204 architecture and performance. In *Proceedings of the 2nd International Workshop on High Performance Transaction Systems*, pages 40–59, 1987.

[80] Patrick E. O'Neil and Goetz Graefe. Multi-table joins through bitmapped join indices. *SIGMOD Record*, 24(3):8–11, 1995.

[81] Patrick E. O'Neil and Dallan Quass. Improved query performance with variant indexes. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 38–49, 1997.

[82] Gültekin Özsoyoğlu, Ismail Sengör Altingövde, Abdullah Al-Hamdani, Selma Ayşe Özel, Özgür Ulusoy, and Zehra Meral özsoyoğlu. Querying web metadata: Native score management and text support in databases. *ACM Transactions on Database Systems*, 29(4):581–634, 2004.

[83] Sriram Raghavan and Hector Garcia-Molina. Complex queries over web repositories. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 33–44, 2003.

[84] Denis Rinfret, Patrick O'Neil, and Elizabeth O'Neil. Bit-sliced index arithmetic. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 47–57, 2001.

[85] Kenneth A. Ross and Divesh Srivastava. Fast computation of sparse datacubes. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 116–125, 1997.

[86] P. Griffiths Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, pages 23–34, 1979.

[87] Gholamhosein Sheikholeslami, Surojit Chatterjee, and Aidong Zhang. WaveCluster: A multi-resolution clustering approach for very large spatial databases. In *Proceedings of the 24th International Conference on Very Large Data Bases*, pages 428–439, 1998.

[88] David E. Simmen, Eugene J. Shekita, and Timothy Malkemus. Fundamental techniques for order optimization. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 57–67, 1996.

[89] Rishi Rakesh Sinha, Soumyadeb Mitra, and Marianne Winslett. Bitmap indexes for large scientific data sets: A case study. In *IPDPS*, 2006.

[90] Divesh Srivastava, Shaul Dar, H. V. Jagadish, and Alon Y. Levy. Answering queries with aggregation using views. In *Proceedings of the 22nd International Conference on Very Large Data Bases*, pages 318–329, 1996.

[91] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *Proceedings of the 19th International Conference on Data Engineering*, pages 391–402, 2003.

[92] Aris Tsois and Timos K. Sellis. The generalized pre-grouping transformation: Aggregate-query optimization in the presence of dependencies. In *Proceedings of the 29th International Conference on Very Large Data Bases*, pages 644–655, 2003.

[93] Gerasimos Malakasiotis Vaclav Lin, Vasilis Vassalos. MiniCount: Efficient rewriting of COUNT-queries using views. In *Proceedings of the 22nd International Conference on Data Engineering*, 2006.

[94] Wei Wang, Jiong Yang, and Richard R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 186–195, 1997.

[95] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *ICPDIS*, pages 68–77, 1991.

[96] Kesheng Wu, Ekow Otoo, and Arie Shoshani. Optimizing bitmap indices with efficient compression. *ACM TODS*, 31(1):1–38, 2006.

[97] Ming-Chuan Wu and Alejandro P. Buchmann. Encoded bitmap indexing for data warehouses. In *Proceedings of the 14th International Conference on Data Engineering*, pages 220–230, 1998.

[98] Weipeng P. Yan and Per-Åke Larson. Performing group-by before join. In *Proceedings of the 10th International Conference on Data Engineering*, pages 89–100, 1994.

[99] Weipeng P. Yan and Per-Åke Larson. Eager aggregation and lazy aggregation. In *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 345–357, 1995.

[100] Ke Yi, Hai Yu, Jun Yang, Gangqiang Xia, and Yuguo Chen. Efficient maintenance of materialized top-k views. In *Proceedings of the 19th International Conference on Data Engineering*, pages 189–200, 2003.

[101] Man Lung Yiu, Nikos Mamoulis, and Yufei Tao. Efficient quantile retrieval on multi-dimensional data. In *Proceedings of the 10th International Conference on Extending Database Technology*, pages 167–185, 2006.

[102] Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle. Introduction to OLAP function. *Change proposal. ANS-NCTS H2-99-14 (April)*, 1999.

[103] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. BIRCH: An efficient data clustering method for very large databases. In *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data*, pages 103–114, 1996.

[104] Yihong Zhao, Prasad Deshpande, and Jeffrey F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 159–170, 1997.

Author's Biography

Chengkai Li was born in Nanchang, the capital city of Jiangxi Province in China. He graduated from Nanjing University with a B.S. degree and an M.Eng. degree in Computer Science, in 1997 and 2000, respectively. Following the completion of his Ph.D. at the University of Illinois at Urbana-Champaign, Chengkai Li will begin work as an assistant professor in the Department of Computer Science and Engineering, at the University of Texas at Arlington.