RANK-AWARE QUERY PROCESSING AND OPTIMIZATION

A Thesis

Submitted to the Faculty

of

Purdue University

by

Ihab F. Ilyas

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2004

To my parents, Aida and Francis, and to my wife, Mirette. Without your unconditional love, support and sacrifice, this thesis could not have been possible.

ACKNOWLEDGMENTS

I am totally indebted to my advisors, Walid Aref and Ahmed Elmagarmid for their enormous efforts and support throughout this journey. I will be always grateful for their invaluable guidance in research, and in life.

I learned from Walid how to be a researcher. He taught me how to identify research challenges, how to tackle them, and more importantly, how to write and publish my solutions. He significantly contributed to my passion for databases. Whether a research problem, or a critical career decision, Walid has been always there for me. I will never forget his generosity, thoughtfulness, and the endless hours he spent with me working and advising.

I will be always grateful for Ahmed's unlimited support. It was Ahmed who recruited me to Purdue, and it was him that put me on the beginning of the road of this research. Ahmed unique vision and research maturity added significant value to my thesis. Ahmed provided me with the freedom to work on whatever I liked to, and, on the same time the guidance that prevented me from diverting away from achieving my degree.

I would like to express my deepest gratitude to the rest of my advisory committee at Purdue, Jeffrey Scott Vitter and Sunil Prabhakar, for their help, support and invaluable comments and suggestions.

During two summers at IBM Almaden research center, I have worked with a wonderful group of people. I would like to thank Guy Lohman, Jun Rao, Volker Markl, Peter Haas and Paul Brown for their great mentorship and guidance. Through my work with this excellent group of researchers, I learned how to combine between the soundness and practicality of my research proposals, and how to conduct large scale system-oriented research.

My wife, Mirette, sacrificed her job, family and friends back home to support me in pursuing my Ph.D. at Purdue. Her love, dedication, patience and unconditional support made this thesis a reality. To her, I am forever indebted.

The beginning of my journey to the Ph.D. degree started as early as my first grade. Among all the people who helped and supported me, my parents Aida and Francis are the ones who suffered the most. Their love and endless sacrifices made me a person. They have never saved an effort to help me achieve my own goals in life. To them I dedicate all my previous and future achievements.

Special thanks goes to my friend Nagi Gebraeel. Nagi has been always there for me and in the toughest moments, his support kept me going.

Thinking of all these people who supported and helped me through life, I can only hope to be as generous and supportive to my kids, Andrew and Marina, and to my future students.

Ahead of all, I thank God for all the success I had in life and I keep praying to have his support and guidance in the rest of my career and life in general.

TABLE OF CONTENTS

Page

Page

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Ilyas, Ihab F. Ph.D., Purdue University, August, 2004. Rank-aware Query Processing and Optimization. Major Professors: Ahmed K. Elmagarmid and Walid G. Aref.

This dissertation focuses on supporting ranking in relational database systems through a rank-aware query processing and optimization framework. We introduce ranking algorithms and operators to be adopted by current relational query engines and we provide a cost-based query optimization technique that integrates the proposed operators in practical relational query processors. In particular, we introduce two rank-join algorithms. The first algorithm joins multiple ranked inputs on key attributes and is realized as the binary key rank-join query operator KRJN. The second rank-join algorithm is more general and joins multiple ranked inputs on general join conditions. The second algorithm is realized in two binary query operators; HRJN and HRJN*. Our rank-join algorithms make use of the individual orders of the input relations. The join results are ordered on a user-specified scoring function. The idea is to rank the join results progressively during the join operation. We address several practical issues and optimization heuristics to integrate the new join operators in practical query processors.

To make these operators practically useful, we introduce a rank-aware query optimization framework that fully integrates rank-join operators into relational query engines. The framework is based on extending System `R` dynamic programming algorithm in both enumeration and pruning. We define ranking as an interesting property that triggers the generation of rank-aware query plans. We introduce a probabilistic model for estimating the input cardinality, and hence the cost of a rank-join operator. To our knowledge, this work is the first effort in estimating the

needed input size for optimal rank aggregation algorithms. Costing ranking plans, although challenging, is key to the full integration of rank-join operators in real-world query processing engines.

We experimentally evaluate our rank-join operators and optimization framework by modifying the query optimizer of an open-source database management system. The experimental evaluation of our approach compares recent algorithms for joining ranked inputs and shows superior performance for our techniques. The experiments also show the validity of our framework and the accuracy of the proposed estimation model.

1 INTRODUCTION

This dissertation studies supporting ranked retrieval in database systems. In ranked retrieval, answers to user queries are reported (and computed) in an order that respects specific user criteria. Example of ranked retrieval support is producing the top k answers to a similarity query in multimedia databases.

In this chapter, we begin by highlighting the bigger picture of integrating information retrieval technology in database systems. Next, in Section 1.2, we motivate (using real examples) for the need to support ranked retrieval in database systems through efficient handling of ranking queries. In Section 1.4, we give formal definitions of two types of ranking queries: the top-k selection and the top-k join queries. We give an overview of our solution to support ranked retrieval in Section 1.5. We summarize the main contributions of the dissertation in Section 1.6. Section 1.7 gives an outline for the rest of the dissertation.

1.1 The Integration between Databases and Information Retrieval

Information retrieval (IR) is concerned with documents (objects) that are likely to be relevant to the user's need as expressed in his request [1]. A major development step in an IR system usually involves selecting a *ranking* function that determines the extent to which a document is relevant to a query [2]. On the other hand, database systems (DBMS) are based on *boolean logic* and traditionally supports a different kind of retrieval than those of IR systems. Most of the database systems now try to extend their applicability to support multimedia retrieval, digital libraries, keyword search in relational data and mark-up languages. Hence, the distinction between database systems and IR becomes less obvious. In many cases, an information system that combines the capabilities of both IR and database systems is the only efficient

solution. For many years, combining the advantages of both worlds, databases and information retrieval systems, has been the goal of many researchers. For example, the QUIQ engine [3] is a hybrid IR-DB system that combines the desirable features from IR and DB, and many commercial systems are adding capabilities for keyword search and handling XML documents [4]. Unfortunately, current database systems do not have a sophisticated notion of relevance and no efficient retrieval based on ranking.

One approach towards integrating databases and IR is to introduce IR-style queries as a challenging type of database queries. The new challenge requires several changes that vary from introducing new query language constructs to augmenting the query processing and optimization engines with new query operators. It may also introduce new indexing techniques and other data management challenges.

1.2 Ranking Queries

A ranking query (also known as top-k query) is an important type of queries that allows for supporting IR-style applications on top of database systems. Emerging applications that depend on ranking queries warrant efficient support of ranking queries in real-world database management systems. For example, in the context of the web, the main applications include building meta-search engines, combining ranking functions and selecting documents based on multiple criteria [5]. Efficient rank aggregation is the key to a useful search engine. In the context of multimedia and digital libraries, an important type of query is similarity matching. Users often specify multiple features to evaluate the similarity between the query media and the stored media. Each feature may produce a different ranking of the media objects similar to the query, hence the need to combine these rankings, usually, through joining and aggregating the individual feature rankings to produce a global ranking. Similar applications exist in the context of information retrieval and data mining.

Most of these applications have queries that involve joining multiple inputs, where users are usually interested in the top-k join results based on some score function. The answer to a *top-k join query* is an ordered set of join results according to some provided function that combines the orders on each input.

The following examples illustrate possible scenarios for top-k join queries and highlight their challenges to current relational database systems.

Example 1.2.1 *Consider a video database system where several visual features are extracted from each video object (frames or segments). Example features include color histograms, color layout, texture, and edge orientation. Features are stored in separate relations and are indexed using high-dimensional indexes that support similarity queries. Suppose that a user is interested in the top* 10 *video frames most similar to a given query image based on color. The top-k query is translated into a similarity query (a nearest-neighbor query) using the high-dimensional index on the color feature. Only the top* 10 *results are presented to the user.*

We call the previous query a *single-feature* or a *single-criterion* ranking. Answering single-criterion ranking query does not require any join. A database system that supports approximate matching ranks the tuples depending on how well they match the query according to some similarity measure.

Example 1.2.2 *In Example 1.2.1, suppose that the user is interested in the top* 10 *video frames most similar to the given query image based on color and texture combined. The user also provides a function for how to combine the ranking according to each feature in an overall ranking. For example, the global rank of a frame* $= 0.5 \times rank(color) + 0.5 \times rank(texture)$*, where* $rank(F)$ *of a frame (*v*) is the rank of* v *among all video frames with respect to the similarity of* v *to the query image, based on Feature* F*.*

We refer to the query in Example 1.2.2 as a *multi-criteria* ranking query, or simply a top-k join query. Unlike single-criterion ranking, in top-k join queries, the

database query processor combines (joins) the individual rankings into one global ranking by applying the provided rank-combining function.

Example 1.2.3 *Consider a user interested in finding a location (e.g., city) where the combined cost of buying a house and paying school tuition in that location is minimum. The user is interested in the top five least expensive places. Assume that there are two external sources (databases): Houses and Schools that can provide information on houses and schools, respectively. The Houses database can provide a ranked list of the cheapest houses and their locations. Similarly, the Schools database can provide a ranked list of the least expensive schools and their locations.*

A naïve way to answer the user query in Example 1.2.3 is to retrieve two lists: a list of the cheapest houses from *Houses* and a list of the cheapest schools from *Schools*. The user then "joins" the two lists based on location. A valid pair is a house and a school in the same location. For all the join results, the user computes the total cost of each pair, e.g., by adding the house price and the school tuition for five years. The five cheapest pairs constitute the final answer to the user query. Unfortunately, the user has to "guess" the size of the input lists that will produce five valid matches. If, after the join operation, there are fewer than five join results, the whole process needs to be restarted with larger input sizes.

In all previous examples, answering the top-k join query can be prohibitively expensive and requires complex *join* and *sorting* operations on large amounts of input data.

Efficient processing of ranking queries gained the attention of many researchers and database vendors. For example, strategies for answering top-k selection queries over relational databases have been introduced in [6] and were prototyped on top of Microsoft `SQL` Server. In [6], top-k selection queries are mapped to range queries with an adaptable range parameter to produce the top-k results. Other techniques that maintain materialized views or special indexes to enhance the response time of top-k queries are introduced in [7–9]. Although these techniques enhance the database

system performance in answering top-k queries, they are implemented either at the application level or outside the core query engine. Hence, processing and optimizing top-k queries lose the benefit of true integration with other basic database query types. We further discuss related work on efficient handling of top-k queries in Chapter 2.

1.3 Real-life Application

The research conducted in this dissertation was primarily motivated by the video database management system, VDMBS, developed at Purdue [10]. In VDBMS, hours of video data are stored inside the database. Visual (physical) features are extracted from the video objects on different levels (single video frame or a video segment) and stored in the database as well. Example features include color histograms, texture, edge orientation, and camera motion. Features are stored as multi-dimensional vector data, e.g, color histograms are stored as 32-dimensions vector data.

One important type of queries in VDBMS is similarity query by example; the user provides an image or a video segment and requests the top k video objects (frames or segments) that are most similar to the query object, based on one or multiple visual features. The scenario is very similar to the scenario described in Examples 1.2.1 and 1.2.2.

VDBMS is built using PREDATOR [11] and Shore [12], and we have provided support for single-feature top-k queries through the implementation of high-dimensional indexing techniques on the storage manager level. We implemented GiST [13] in shore as our indexing technique, built high-dimensional indexes on individual features, and implemented a nearest-neighbor scan operator on top of these indexes. Single-feature similarity query is implemented as an execution of the nearest-neighbor scan operator on this particular feature index. Figure 1.1 gives example single-feature similarity queries based on color histogram, texture and edge orien-

tation. The top 4 answers for each feature is reported using the high-dimensional index on this particular feature.

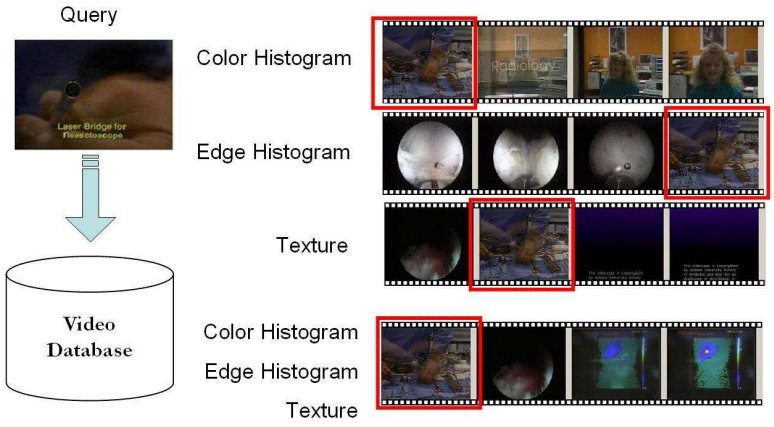


Figure 1.1. Example Single and Multi-feature Queries in VDBMS

To support multi-feature queries, we first experimented with the two basic techniques available in current database technology. In the first approach, we mapped a multi-feature top-k query to a `SQL` query that joins the output of multiple single-feature queries (based on the object id), and then sorts the join results based on a combined score (computed according to some combining function, e.g., a weighted sum). This approach, although simple, did not scale with respect to both the number of features and the database size. For example, a 4-feature top-k query took over 2 hours to complete. The main problem is that sorting is a blocking operation that requires full computation of the join process. Although the input to the join is sorted (ranked) on the individual features, this information cannot be exploited to the join operation in current database system implementation. Hence, sorting the join results becomes necessary to produce the top k answers. The second approach is to sequentially scan all the database objects, compute the score of each object according to each feature, and then combine the scores into a total score for each object. The second approach also suffers from scalability problem with respect to the database size and the number of considered features.

In this dissertation, we show how to modify the query engine of VDBMS to be rank-aware. The new processing paradigm achieves significant performance gain by cutting down the execution time by orders of magnitude as we show in our experiments.

1.4 Query Model

In this section, we formally define our top-k query model. As mentioned earlier in the examples, we distinguish between two types of top-k queries: top-k selection and top-k join queries.

Definition 1.4.1 *Top-k Selection Query: Consider one relation R. Each tuple in R has n attributes $A_1, \ldots, A_n$, and there are m scores, $s_1, \ldots, s_m$ defined on these attributes (e.g., $s_1 = A_1$ and $s_2 = A_2 + A_3$). The* top-k selection query *selects the top-k tuples (objects) from R with the largest combined score $s = f(s_1, \ldots, s_m)$, where f is some monotone scoring function.*

A possible SQL-like notation for expressing a top-k selection query is as follows:

```
SELECT some_attributes
FROM R
WHERE Selection_Condition
ORDER BY f(s_1, ..., s_m)
STOP AFTER k;
```

Definition 1.4.2 *Top-k Join Query: Consider a set of relations R_1 to R_m. Each tuple in R_i is associated with some score that gives it a rank within R_i. The* top-k join query *joins R_1 to R_m and produces the results ranked on a total score. The total score is computed according to some function, f, that combines individual scores. Note that the score attached with each relation can be the value of one attribute or a value computed using a predicate on a subset of its attributes.*

A possible SQL-like notation for expressing a top-k join query is as follows:

```
SELECT *
FROM R_1, R_2, ..., R_m
WHERE join_condition(R_1, R_2, ..., R_m)
ORDER BY f(R_1.score, R_2.score, ..., R_m.score)
STOP AFTER k;
```

The following query (Query Q1) is an example of a top-k join query that follows the template in Definition 1.4.2, expressed in SQL99.

Q1:

```
WITH RankedABC as (
  SELECT A.c1 as x ,B.c2 as y, rank() OVER
  (ORDER BY (0.3*A.c1+0.7*B.c2)) as rank
  FROM A,B,C
  WHERE A.c1 = B.c1 and B.c2 = C.c2)
SELECT x,y,rank
FROM RankedABC
WHERE rank <=5;
```

where `A, B` and `C` are three relations and `A.c1,B.c1,B.c2` and `C.c2` are attributes of these relations.

Top-k selection can be modeled as a special case of top-k join. One way of evaluating a top-k selection query as a top-k join query is to vertically partition R into m relations $R_1, \ldots, R_m$. Relation R_i has the attributes necessary to compute the score s_i. For example: $R_1 = (oid, A_1)$ and $R_2 = (oid, A_2, A_3)$. In this case, the join condition is an *equality* condition on *key attributes*. Despite this observation, top-k selection query needs to be handled separately due to the fact that many of the ranking queries are top-k selection. Hence, we need an efficient solution to handle this type of queries.

1.5 Our Solution to Integrate Ranking in a DBMS

Rank aggregation algorithms can be implemented in a database system using *table functions* or application-level programs. (i.e., outside the query engine). `SQL`table functions [14] are examples of this type of implementation. Since there is no straightforward method for pushing other query predicates into table functions [15], the performance of this query is severely limited and the approach does not give enough flexibility in optimizing the issued queries.

Our solution to support ranked retrieval in database systems is through encapsulating ranking algorithms in physical query operators that can be part of query execution plans. We call the new operators rank-join operators. The new operator encapsulates the rank-join algorithm in its *GetNext* operation; each call to *GetNext* reports the next top element from the ranked inputs. Integrating rank-join algorithms as physical join operators has the following advantages:

- The ranking query is under the optimizer's control; best ranking algorithms and strategies can be chosen based on the estimated cost and other execution environment variables.

- Ranking operators can be shuffled with other operators in a query evaluation plan for better performance (e.g., pushing down predicates and projections). This flexibility is not possible when ranking is implemented as a black box in a user-defined function.

- The ranking functionality is general enough and highly applicable that warrants efficient implementation as a core query operator. This native support of ranking greatly simplifies the development of many emerging applications, by pushing the ranking logic inside the database engine.

Using traditional join operators to answer a ranking query will result in the execution plan in Figure 1.2. The blocking sorting operator on top of the join

is unavoidable. If the inputs are large, the cost of this plan can be prohibitively expensive.

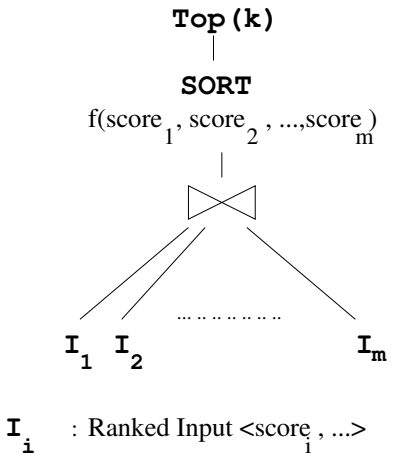


Figure 1.2. Existing Techniques for Ranking in a Query Plan

The biggest advantage of encapsulating a ranking algorithm in a physical query operator is that ranking can be adopted by practical query engines. The query optimizer will have the opportunity to optimize a ranking query by integrating the new operator in ordinary query execution plans. Figure 1.3 gives alternative execution plans to rank-join three ranked inputs.

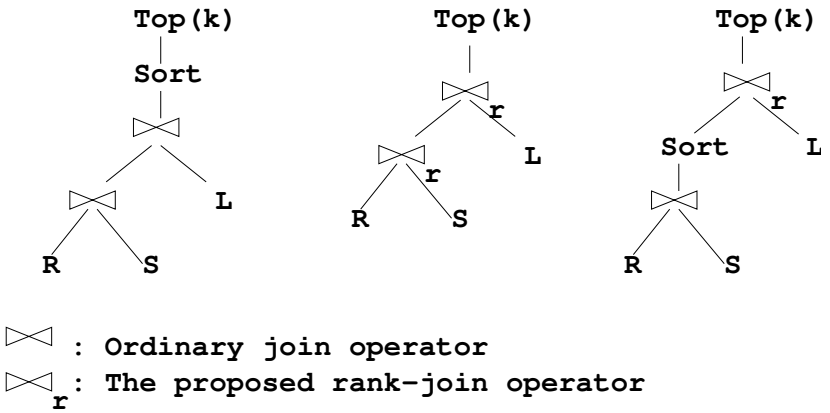


Figure 1.3. Alternative Execution Plans to Rank-join Three Ranked Inputs

1.6 Contributions

The main contributions of this dissertation are as follows:

- We show that supporting ranking as a basic database functionality enables many key applications on top of current database systems. We show the advantages of supporting ranking on the relational operator level inside the database engine in contrast to implementing ranking algorithms in the application level.

- We introduce two rank-join algorithms to produce ranked join results according to some user-defined function. The first algorithm extends an existing rank aggregation algorithm to allow for pipelining. The second algorithm is the most general algorithm so far to join ranked inputs under arbitrary join conditions. We provide complexity analysis and optimality proofs for the proposed algorithms.

- We introduce a family of physical binary join operators that implement the rank-join algorithms. The operators are pipelined and follow the same interface of other relational query operators. Hence, the new rank-join operators can be easily integrated within query execution plans. We address several optimization issues and practical aspects in integrating rank-join operators in pipelined query execution plans.

- The dissertation also introduces a cost-based optimization framework that integrates the proposed rank-join operators in practical query processors. The rank-aware query optimization framework is based on extending the System R dynamic programming algorithm in both enumeration and pruning. We define ranking as an interesting property that triggers the generation of rank-aware query plans. Unlike traditional join operators, optimizing for rank-join operators depends on estimating the input cardinality of these operators. We introduce a probabilistic model for estimating the input cardinality, and hence

the cost of a rank-join operator. To our knowledge, this work is the first effort in estimating the needed input size for optimal rank aggregation algorithms.

We experimentally evaluate the performance of the introduced operators and the accuracy of the cost estimation model. The results prove the validity of our approach and show the advantage of deploying our technique in a database system prototype. For example, in a multimedia retrieval application, using our rank-aware query processing framework achieved orders of magnitude enhancement in query response time.

1.7 Summary and Outline

Efficient handling of ranking queries in database systems enables many key applications in multimedia, web databases, information retrieval and other emerging applications. Rank-aware query processing and optimization aim at supporting ranking as a basic functionality on the query engine level. In this chapter, we introduced definitions for two variants of top-k queries. We briefly summarized our contributions in integrating ranking in database query processing through introducing rank-aware join algorithms and operators, and providing a cost-based optimization framework that generates rank-aware query execution plans.

The rest of this dissertation is organized as follows. Chapter 2 highlights some related work and necessary background, and gives a summary of some current ranking algorithms. Chapter 3 presents our solution for answering top-k selection queries. We introduce a new rank-join query operator named KRJN along with some optimization heuristics. Chapter 3 also gives an experimental evaluation of KRJN̈In Chapter 4, we present our solution for answering general top-k join queries. We provide the necessary correctness and optimality proofs and address several optimization heuristics of two new rank-join operators HRJN and HRJN*. We conclude Chapter 4 by a performance evaluation of the two rank-join operators and compare their performance with a recent algorithm to join multiple ranked inputs. Chapter 5 de-

scribes our rank-aware query optimization framework. The optimization framework includes a novel cost model to estimate the input cardinality of rank-join operators. We conclude in Chapter 6 by a summary and final remarks.

Parts of this dissertation have been published in conferences and journals; the new proposed rank-join query operators and their engineering details have been published in VLDB-2002 [16], VLDB-2003 [17], and the VLDB Journal [18]. The rank-aware query optimization framework is published in ACM SIGMOD-2004 [19].

2 BACKGROUND AND RELATED WORK

In this chapter, we present an overview of the state-of-the-art techniques for efficient evaluation of top-k queries. We discuss different alternatives to answer top-k selection queries and top-k join queries. One important technique is rank aggregation, which we build upon later in this dissertation. We give a theoretical background for rank aggregation methods and introduce a taxonomy of rank aggregation algorithms based on the generality of the join conditions among rankings and on the assumptions made on the available access methods. For the purpose of self containment, we give the necessary background information on practical relational join operators and cost-based query optimization. We rely on these concepts in describing our contributions later in this dissertation.

This chapter is organized as follows. Section 2.1 gives an overview of the different approaches proposed in the literature to efficiently answer top-k queries. A brief theoretical background on voting, rank aggregation and ranking distances is given in Section 2.2. In Section 2.3, we give an overview of rank aggregation algorithms recently proposed in the context of database applications. In Sections 2.4 and 2.5, we provide the necessary background information for the ripple join algorithms and cost-based query optimization, respectively. Section 2.6 summarizes the chapter.

2.1 Approaches for Evaluating Top-k Queries

In this section, we overview and compare the basic approaches of evaluating top-k selection and top-k join queries as defined in Section 1.4.

We can identify three basic approaches for efficient evaluation of top-k selection queries:

- *Filter/Restart* In this approach, a top-k selection query is mapped to a multi-dimensional range query [6, 20, 21]. The dimensions of the space are the score attributes. If the filtering is too selective and less than k results are produced, the query needs to be "restarted" to compute the rest of the results. Challenges include quantifying and minimizing the risk of restarts and optimizing top-k queries that follow the filter/restart model [22]

- *Rank Aggregation* The rank aggregation method maps a top-k selection query to aggregating multiple ranked lists. Several algorithms have been introduced in the literature [16, 23–28]. Later in this chapter, we present a new taxonomy that put these algorithms under one framework. The proposed taxonomy of rank aggregation algorithms gives better understanding of the difference and the applicability of these algorithms.

- *Ranking Indexes and Ranking Views* Using indexes and materialized views is another approach for enhancing the performance of frequent similar top-k selection queries. Example prototypes that follow this approach are PREFER [8] and the Onion technique [7].

Similarly, we can identify two major approaches for efficiently evaluating top-k join queries.

- *Rank Aggregation* New rank aggregation algorithms have been developed recently to cover general join condition among the input ranked lists [17,28,29]. We present these algorithms and show the common abstract idea behind these algorithms.

- *Ranking Indexes* Recently, the use of indexes was introduced to enhance the performance of top-k join queries [9]. We go through the details of this approach later in this chapter.

We summarize the differences among these approaches as follows:

- The filter/restart method is an easy-to-implement and read-to-go solution. It builds on the available support of range query processing. In principle, no need to change the core query engine implementation; the mapping is done on the application level. The main disadvantage of this approach is the difficulty of coming up with a "good" range parameter for the range query. A conservative range that guarantees no restarts can produce too many results that lead to inefficient execution. On the other hand, a more aggressive range choice with the probability of restarts can cause multiple executions of the same query to get the top-k answers. Although there have been studies to minimize the restarts [21, 22], the risk of restarts (for aggressive range choices) or generating too many false intermediate results (in conservative range choices) can compromise the efficiency of this approach. Moreover, most of the implantations of this approach assume the availability of distribution information (e.g., histograms) on the scores to estimate the range. Unfortunately, in many top-k applications, the scores are not available offline, rather, they are progressively computed during execution. Another disadvantage is the static nature of this approach; the filter/restart method assumes that k is known apriori, and the query needs to be reissued when more than k results are required, i.e., the approach is not incremental.

- The rank aggregation method can be implemented both at the application level (e.g., as a user-defined function) or at the query engine level (e.g., in terms of query operators). The main advantages of this technique are:

  - Most of the algorithms are incremental, i.e., k does not need to be know beforehand; the rank aggregation algorithm can produce the next top answer building on the current algorithm state (intermediate results).

  - Rank aggregation is deterministic, i.e., the output is exactly the top k answers. Hence, there is no need to restart the query as in the filter/restart method.

  - A notion of optimality can be defined that guarantees that rank aggregation algorithm do not retrieve more inputs than necessary. Hence, we can define some performance guarantees.

  In this chapter and in the rest of the dissertation we concentrate on the rank aggregation approach as an efficient way to evaluate top-k queries. In particular, we adopt the approach of implementing rank aggregation on the engine level in terms of query operators [16, 17, 19].

- Unlike the two previous methods (filter/restart and rank aggregation), the index and materialized view approach has a different objective. The main goal of this approach is to answer top-k queries as efficient as possible when similar top-k queries are issued with different scoring functions. In the two previous methods, the query needs to be reevaluated whenever the scoring function has changed. In the index and materialized view approach, the answers of top-k queries are materialized for different scoring functions to facilitate answering new top-k queries with different scoring functions. We view the indexing and materialized view approach as an orthogonal technique that can be used in tandem with our work in rank-aware query processing and optimization.

2.2 Voting and Rank Aggregation

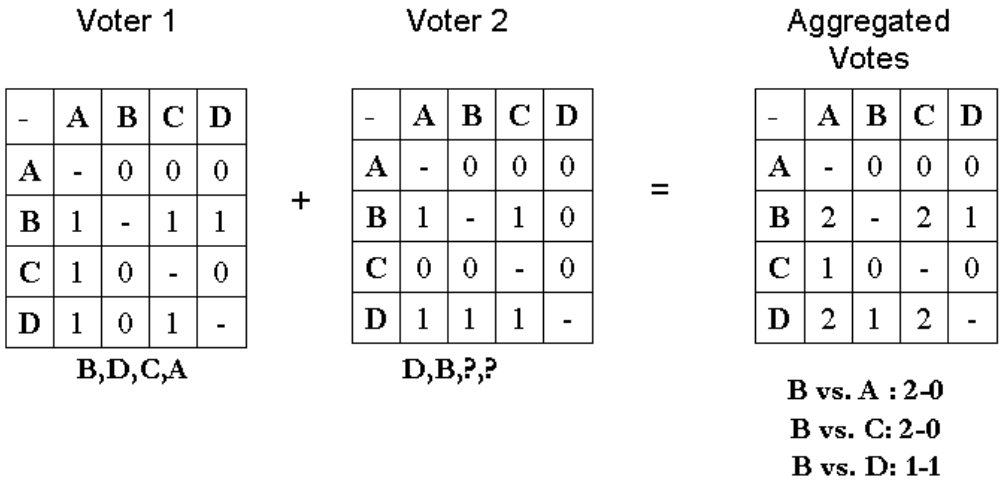
In this section, we give a more detailed overview of rank aggregation. We start by a historical and theoretical background on voting and rank aggregation methods, then we present a taxonomy of current rank aggregation algorithms.

2.2.1 Condorcet Voting

The problem of voting and rank aggregation goes back to at least two centuries when Marie Jean Antoine Nicolas Caritat, Marquis de Condorcet (1785) [30] introduced an efficient method for conducting elections based on ranked-pairs. The main

idea is that an election race is broken down into separate pairwise races between possible pairing of the candidates. Each ranked ballot is then interpreted as a vote in each of those one-on-one races. If candidate A is ranked above candidate B by a particular voter, that is interpreted as a vote for A over B. If one candidate beats each of the other candidates in their one-on-one races, that candidate wins. Otherwise, the result is ambiguous different procedures are used to resolve the ambiguity.

Figure 2.1 gives an example of a Condorcet voting process for two voters and four candidates(A, B, C and D). The vote of each voter is represented as a matrix that gives the preference of that voter in comparing pairs of candidates. Note that the matrix for Voter 2 did not specify a preference between candidates A and C, i.e., a total order among candidates is not required. As we will show later, this is a basic difference between the Condorcet method and other positional ranking methods that require total order of the candidates.



-	A	B	C	D
A	-	0	0	0
B	1	-	1	1
C	1	0	-	0
D	1	0	1	-

+

-	A	B	C	D
A	-	0	0	0
B	1	-	1	0
C	0	0	-	0
D	1	1	1	-

=

-	A	B	C	D
A	-	0	0	0
B	2	-	2	1
C	1	0	-	0
D	2	1	2	-

Figure 2.1. An Example of Condorcet Voting

Many of the existing aggregation methods do not ensure the election of the Condorcet winner, should one exist [5].

2.2.2 Optimal Rank Aggregation

Given a universe U, an ordered list (or simply, a list) τ with respect to U is a ranking of a subset S of U, i.e., $\tau = [x_1 > x_2 > \ldots > x_d]$, with each x_i in S, and $>$ is some ordering relation on S. Also, if i in U is present in τ, let $\tau(i)$ denote the position or rank of i. For a list τ, let $|\tau|$ denote the number of elements. By assigning a unique identifier to each element in U, we may assume without loss of generality that $U = 1, 2, ..., |U|$.

In the generic context of aggregating multiple lists that rank a set of objects, the notion of "better" depends on what distance measure we optimize. The distance measure in this case is our way to quantify the difference between two rankings with respect to S.

Two popular distance measures [31, 32] are: (1) The Spearman footrule distance is the sum, over all elements i in S, of the absolute difference between the rank (position) of i in the two lists. Formally, given two lists σ and τ, their Spearman footrule distance is given by $F(\sigma, \tau) = \sum_i |\sigma(i) - \tau(i)|$; and (2) The Kendall tau distance counts the number of pairwise disagreements between two lists; that is, the distance between two full lists σ and τ is $K(\sigma, \tau) = |\{(i, j) : i < j, \sigma(i) < \sigma(j) \wedge \tau(i) > \tau(j)\}|$.

Given lists $\tau_1, \ldots, \tau_k$, find a ranking σ such that σ is a full list with respect to the union of the elements of $\tau_1, \ldots, \tau_k$ and σ minimizes $K(\sigma, \tau_1, \ldots, \tau_k)$. The aggregation obtained by optimizing Kendall distance is called Kemeny optimal aggregation and in a precise sense, corresponds to the geometric median of the inputs.

Although Kemeny optimal rank aggregation chooses the Condorcet winner, if one exists, computing the Kemeny optimal aggregation is NP-Hard even when $k = 4$ [5]. It has been shown that Kendall distance can be approximated via the Spearman footrule distance [31]; for any two lists σ, τ, $K(\sigma, \tau) < F(\sigma, \tau) < 2K(\sigma, \tau)$. When the optimizing criterion is the footrule distance, the result aggregation is footrule optimal aggregation which can be computed in polynomial time. Although footrule optimal aggregation does not guarantee the choice of the Condorcet winner, it can

be a good approximation of the Kemeny optimal aggregation in the sense that: If σ is the Kemeny optimal aggregation of lists $\tau_1, \ldots, \tau_k$, and σ' optimizes the footrule aggregation, then: $K(\sigma', \tau_1, \ldots, \tau_k) < 2K(\sigma, \tau_1, \ldots, \tau_k)$

2.2.3 Borda's Method and Positional Ranking

Positional rank aggregation methods are perhaps the easiest to compute (linear time computation). Borda's method [33] is a positional method in that it assigns a score corresponding to the position in which a candidate appears within each voter's ranked list of preferences, and the candidates are sorted by the sum of their total score. However, it is proved that positional methods cannot satisfy the Condorcet criterion. Because of their simplicity, Borda's rank aggregation is a basic method for most of practical rank aggregation algorithms introduced in the literature.

2.3 Current Rank Aggregation Algorithms

Table 2.1 summarizes current rank aggregation algorithms and compares their basic properties. First, Algorithm FA [23] was introduced by Fagin as an efficient solution to the problem. Algorithms TA [24] (Fagin et al.), Multi-step [25] (Nepal et al.) and Quick-Combine [26] (Güntzer et al.) are equivalent and are an enhancement over the FA algorithm. these four algorithms depends on the availability of random access to the ranked inputs and hence are not pipelined; random access is not possible when the input arrives as output from another execution of the algorithm in the query pipeline. They can only be executed on the leaf level of the query evaluation plan. For example, realizing the FA algorithm in the IBM GARLIC middleware [34] was limited to a query operator that can exist only on the leaf level of the query evaluation plan.

Algorithms NRA [24] (Fagin et al.) and Stream-Combine [27] (Güntzer et al.) do not require random access to the ranked inputs. While Stream-Combine can be realized easily as a pipelined operator, the NRA algorithm is not pipelined since the

output does not have exact grades associated with the reported objects. Hence, the output of the NRA algorithm cannot serve as a valid input to another NRA execution. The aforementioned algorithms are designed for the top-k selection problem, i.e., they join (and aggregate) multiple ranked lists according to a "key" join condition.

The J^* algorithm [29] (Natsev et al.) also does not require random access to the input and can easily be realized as a pipelined query operator. J^* joins multiple ranked inputs according to a general join condition.

Recently, in [35], the authors introduced Algorithms Upper and Pick for evaluating top-k selection queries over web-accessible sources assuming that only random access is available for a subset of the sources. Similarly, Algorithm MPro by Chang and Hwang [28] addresses the expensive probing of some of the object scores in top-k selection queries. They assume a sorted access on one of the attributes while other scores are obtained through probing or executing some user-defined function on the remaining attributes. The authors in [28] introduced an extension to MPro to handle general join conditions.

Our algorithm, the Rank-join algorithm [17], is the most general rank-aggregation algorithm introduced so far. Rank-join doe snot require random access and hence pipelined (although it can exploit random access if available). Rank-join also handles general join conditions among input rankings.

In the following sections we give the details of two of these algorithms that will facilitate the description of our proposed techniques in the rest of this dissertation. In particular, we are interested in this class of rank-join algorithms that do not require random access to their input streams. Algorithm Stream-Combine is very similar to the NRA algorithm except for the fact that it requires a reported object to be seen, through sorted access, in all input streams. The NRA algorithm does not require this condition, and hence has a faster termination condition as we will discuss shortly. Hence, we choose to elaborate on the NRA algorithm and the J^* algorithm.

Table 2.1
Rank-join Algorithms

Algorithm	Required Random Access	Pipelined	Join Condition
FA	Yes	No	key
TA	Yes	No	key
Multi-step	Yes	No	key
Quick-Combine	Yes	No	key
Stream-Combine	No	Yes	key
NRA	No	No	key
J^*	No	Yes	General
MPro, Upper, Pick	Yes	No	General
Rank-join	No	Yes	General

The NRA Algorithm The NRA algorithm views the database as m input lists where each list consists of objects associated with grades and objects are sorted in descending order on these grades. Let t be a weighting function to compute the overall grade of an object by applying t to all the individual grades of this object. The NRA algorithm [24] (no-random-access) visits objects from the m input lists in parallel. At depth d (i.e., when the first d objects have been visited across all m streams), the *bottom* values $\underline{x}_1^{(d)}$, $\underline{x}_2^{(d)}$, ..., $\underline{x}_m^{(d)}$ are maintained as the grades last seen from each input list. For an object R with l discovered fields x_1, ..., x_l, $l \leq m$, we compute the worst grade as $W^{(d)}(R) = t(x_1, x_2, x_3, \ldots, x_l, 0, \ldots, 0)$ and the best grade as $B^{(d)}(R) = t(x_1, x_2, x_3, \ldots, x_l, \underline{x}_{l+1}, \ldots, \underline{x}_m)$. Objects encountered thus far are sorted in descending order according to their worst grade, where ties are broken using an object's best grade. If the top k objects are required, and we let $M_k^{(d)}$ be the k^{th} largest worst grade, then the algorithm halts when no object outside the top k objects encountered thus far has a best grade greater than $M_k^{(d)}$.

Using the NRA algorithm directly in the implementation of a pipelined query operator is complicated by two problems. First, the algorithm depends on a predefined value for k, the number of top results to be retrieved. What we need is an *incremental* version of the algorithm which produces the *next top* object when needed by the caller. The second problem is that the output from the algorithm does not have exact grades associated with the output objects. Instead, each object has a range from worst grade to best grade. This prevents pipelining the operator in the query plans, since the exact ranks (grades) will be available only from the source input streams.

The J^* Algorithm The J^* algorithm is introduced in [29] by providing the method *GetNext* that reports the next top join combination in each call. The algorithm is based on the A^* class of search algorithms. As in the A^* search algorithm, the cost of the path leading to the final answer is divided into two parts: the first part is the actual cost encountered thus far, and the second part is an estimate of the cost before reaching the final answer. The J^* algorithm works as follows. For each input stream, a variable is defined whose set of possible values are the tuples from the corresponding stream. The goal is to find a valid assignment for all the variables that maximizes the total score, which corresponds to finding the top valid join combination. The term *state* is defined as a set of variable assignments, and a *complete state* is a state that instantiates all the variables. Otherwise, the state is called a *partial state*. To find the next top join combination, the algorithm maintains a priority queue of partial and complete join results ordered on *upper bound* estimates of the final combination scores. At each step, the state on top of the queue is processed in an attempt to assign one of the unassigned variables by pulling the next tuple from the corresponding input stream. The algorithm terminates when a complete state (a valid join combination) appears on top of the priority queue. Table 2.2 gives a layout of the J* algorithm. The recursive call to *GetNext* in line 10, and the fact that the algorithm returns the answer along with the exact combined score, allows the algorithm to work well with join hierarchies.

Table 2.2
The J^* GetNext Operation

J^*: GetNext()

Given: a queue buffer Q

1. LOOP
2. IF Q is Empty
3. RETURN Null
4. *head* $= Q$.top.
5. IF *head* is a complete state
6. RETURN *head*
7. *head2* = a copy *head*.
8. X = an unassigned variables in *head2*.
9. IF no tuples available for this stream
10. tuple = stream.GetNext.
11. Assign tuple to X and compute score.
12. IF the assignment is valid
13. Push *head2* in Q.
14. Let X in *head* points to the next tuple in the corresponding stream.
15. Push *head* into the Q
16. END LOOP

2.4 Ripple Join

Since our new algorithm to handle top-k join queries with general join conditions depends on the idea of ripple join [36], we present a brief overview of ripple join. Ripple join is a family of join algorithms introduced in [36] in the context of online processing of aggregation queries in a relational DBMS. Traditional join algorithms are designed to minimize the time till completion. However, ripple joins are designed to minimize the time till an acceptably precise estimate of the query result is avail-

able. Ripple joins can be viewed as a generalization of nested-loops join and hash join. We briefly present the basic idea of ripple join below.

In the simplest version of a two-table ripple join, one previously-unseen random tuple is retrieved from each table (e.g., R and S) at each sampling step. These new tuples are joined with the previously-seen tuples and with each other. Thus the Cartesian product $R \times S$ is swept out as depicted in Figure 2.2.

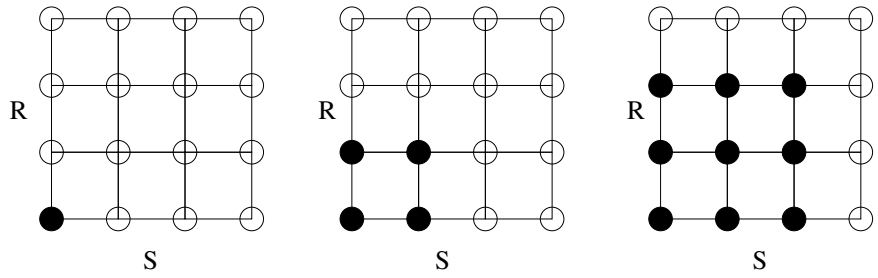


Figure 2.2. Three steps in Ripple Join

The *square* version of ripple join draws samples from R and S at the same rate. However, in order to provide the shortest possible confidence intervals, it is often necessary to sample one relation at a higher rate. This requirement leads to the general *rectangular* version of the ripple join where more samples are drawn from one relation than from the other. Variants of ripple join are: (1) *Block Ripple Join*, where the sample units are blocks of tuples of size b (In classic ripple join, $b = 1$), (2) *Hash Ripple Join*, where all the sampled tuples are kept in hash tables in memory. In this case, calculating the join condition of a new sampled tuple with previously sampled tuples is very fast (saving I/O). The second variant is exactly the *symmetric hash join* [37, 38] that allows a high degree of pipelining in parallel databases. When the hash tables grow in size and exceed memory size, the hash ripple join falls back to block ripple join.

2.5 Cost-based Query Optimization

The optimizer is the component in the query processing engine that transforms a parsed input query into an efficient query execution plan. The execution plan is then passed to the run-time engine for evaluation. The task of a query optimizer is to find the best execution plan for a given query. This task is usually accomplished by examining a large space of possible execution plans and comparing these plans according to their "estimated" execution cost. To estimate the cost of an execution plan, the optimizer adopts a cost model that takes several inputs, such as the estimated input size and the estimated selectivity of the individual operations, and estimates the total execution cost of the query. Most of the different generated plans come from different organizations of the join operations. In general, the larger the space of possible plan, the higher the chance that the optimizer will get a better execution plan.

2.5.1 Plan Enumeration Using Dynamic Programming

Since the join operation is implemented in most systems as a diadic (2-way) operator, the optimizer must generate plans that transform an n-way join into a sequence of 2-way joins using binary join operators. The two most important tasks of an optimizer are to find the optimal join sequence as well as the optimal join method for each binary join. Dynamic programming (*DP*) was first used for join enumeration in System R [39]. The essence of the *DP* approach is based on the assumption that the cost model satisfies the *principle of optimality*, i.e., the subplans of an optimal plan must be optimal themselves. Therefore, in order to obtain an optimal plan for a query joining n tables, it suffices to consider only the optimal plans for all pairs of non-overlapping m tables and $n - m$ tables, for $m = 1, 2, ..., n - 1$.

To avoid generating redundant plans, *DP* maintains a memory-resident structure (referred to as *MEMO*, following the terminology used in [40]) for holding non-pruned plans. Each MEMO entry corresponds to a subset of the tables (and applicable pred-

icates) in the query. The algorithm runs in a bottom-up fashion by first generating plans for single tables. Then it enumerates joins of two tables, then three tables, etc., until all n tables are joined. For each join it considers, the algorithm generates join plans and incorporates them into the plan list of the corresponding MEMO entry. Plans with larger table sets are built from plans with smaller table sets. The algorithm prunes a higher cost plan if there is a cheaper plan with the same or more general properties for the same MEMO entry. Finally, the cheapest plan joining n tables is returned.

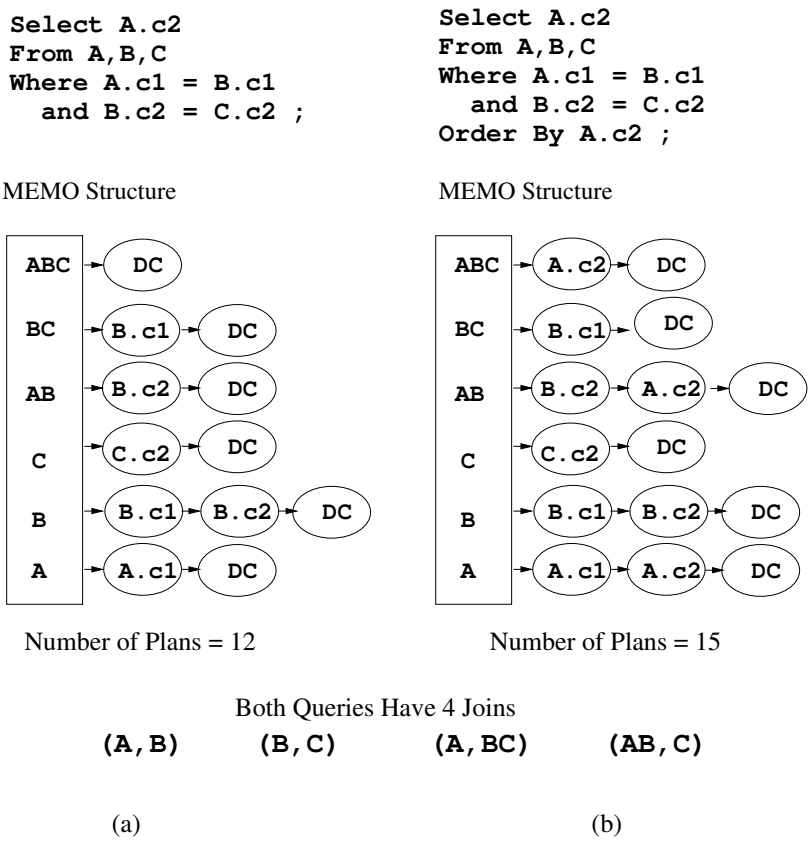


Figure 2.3. Number of Joins vs. Number of Plans

2.5.2 Plan Properties

Plan properties are extensions of the important concept of *interesting orders* [39] introduced in System R. Suppose that we have two plans generated for table R, one

produces results ordered on $R.a$ (call it P_1) and the other does not produce any ordering (call it P_2). Also suppose that P_1 is more expensive than P_2. Normally, P_1 should be pruned by P_2. However, if table R can later be joined with table S on attribute a, P_1 can actually make the sort-merge join between the two tables cheaper than P_2 since it doesn't have to sort R. To avoid pruning P_1, System R identifies orders of tuples that are potentially beneficial to subsequent operations for that query (hence the name interesting orders), and compares two plans only if they represent the same expression and have the same interesting order. In Figure 2.3(a), we show a 3-way join query and the plans kept in the corresponding MEMO structure. For each MEMO entry, a list of plans is stored, each carrying a different order property that is still interesting. We use DC to represent a "don't care" property value, which corresponds to "no order". The cheapest plan with a DC property value is also stored in each MEMO entry if this plan is cheaper than any other plan with interesting orders. Modifying the query to that in Figure 2.3(b), by adding an *orderby* clause, increases the number of interesting order properties that need to be kept in all MEMO entries containing A. By comparing Figure 2.3(a) with Figure 2.3(b), we can see that the number of generated join plans changes, even though the join graph is still the same. The idea of interesting orders was later generalized to multiple physical properties in [41, 42] and is used extensively in modern optimizers. Intuitively, a physical property is a characteristic of a plan that is not shared by all plans for the same logical expression (corresponding to a MEMO entry), but can impact the cost of subsequent operations.

2.6 Summary

In this chapter, we presented an overview of the various proposed approaches to efficiently evaluate top-k queries. All these techniques are proposed to be implemented on the application level. In summary, there are three main techniques: filter/restart, rank aggregation and maintaining indexes and materialized views. While

the first two techniques aim at efficient evaluation of a single top-k query, the indexes and materialized view approach aims at efficient evaluation of top-k queries by materializing the top-k answers according to several ranking criteria.

Optimal rank aggregation techniques are efficient and incremental approach to compute top-k queries. We gave a historical and a theoretical background for rank aggregation methods and we described in detail some of the practical, recently proposed, rank aggregation algorithms. Since we adopt supporting ranking on the query operator level, we gave an overview of ripple join, a class of physical join operators. We also described the basic technique of cost-based query optimization that integrates physical query operators in query execution plans.

3 TOP-K SELECTION ALGORITHMS AND OPERATORS

In this chapter, we introduce a new rank-join algorithm to answer top-k selection queries (top-k join on key attributes). We provide an efficient implementation of the algorithm in terms of a physical binary pipelined query operator that can be easily integrated in current query processors. The introduced algorithm is a modification of the NRA algorithm [24] to work on ranges of scores instead of exact scores. The modified algorithm is encapsulated in the rank-join operator, KRJN. We compare the performance of KRJN with the J^* algorithm [29].

The rest of this chapter is organized as follows. In Section 3.1 we introduce the physical pipelined query operator, KRJN. Section 3.2 presents an optimization heuristic to the basic KRJN algorithm to enhance its performance and scalability to long query pipelines. A comparison between the J^* and KRJN is described in Section 3.3. The two operators and several optimizations are evaluated through an empirical study in Section 3.4. Section 3.5 contains a summary of our findings, recommendations, and concluding remarks.

3.1 The KRJN Operator

The physical query operator, KRJN (Key Rank Join) [16], is a pipelined operator that implements a modified version of the NRA algorithm in [24].

We propose an incremental, pipelined version of the NRA algorithm that can be used to implement the *GetNext* operation of the KRJN operator. We present the modified algorithm in terms of the KRJN operations *Open*, *GetNext*, and *Close*. The internal state information needed by the operator consists of a *priority* queue which holds the objects encountered thus far. The objects are sorted on worst grade in descending order, and ties are broken using the best grade (and then arbitrarily for

ties on the best grades). In order to allow for pipelining, inputs to the algorithm may be source streams or output streams from other algorithm executions. Therefore, each object in the input streams is associated with a range of grades from worst grade w to best grade b (where $w = b$ for exact grades). At depth d (d is the number of objects retrieved from each input stream), the proposed algorithm maintains the bottom values $\underline{b}_1^{(d)}$ and $\underline{b}_2^{(d)}$. The worst grade of an object R is computed as $t(w_1, w_2)$, where t is the weighting function and w_i is either the worst grade of the object according to input i, or 0 if the object has not yet been encountered in input stream i. Similarly, the best grade of an object R is computed as $t(b_1, b_2)$, where b_i is the best grade of the object according to input stream i, or $\underline{b}_i^{(d)}$ if the object has not yet been encountered in input stream i.

In the *Open* operation, the operator initializes the internal state information and opens the left and right child iterators. The *Close* operation destroys the state information and closes the input iterators.

The algorithm for the *GetNext* operation is given in Table 3.1. *GetNext* is the core of the rank join operator.

The algorithm in Table 3.1 begins by checking the buffer (priority queue) to see if an object can be reported. An object can be reported if its worst grade is still greater than the best grades of all other objects. The maximum best grade for objects encountered thus far is obtained from the buffer. For objects not yet encountered, a threshold value can be used as an upper bound of the maximum possible best grade. The threshold is obtained by applying the weighting function to the best grades of the last encountered left and right objects. The maximum best grade in the buffer is maintained so that a scan of the whole buffer is not necessary for each call. To deal with grade ranges, the algorithm uses the best grades from the input streams to update the bottom values and to update the best grade of objects in the buffer.

We introduce a heuristic to the KRJN algorithm in Table 3.1 to reduce the unnecessary ranking overhead at the early stages of the pipeline. We refer to the problem of performing excessive ranking at the early stages of the query pipeline

as the *local ranking* problem. We elaborate on the *local ranking* problem and the heuristic to solve it in the next section.

3.2 Optimizing the KRJN Operator

The KRJN as given in Table 3.1 suffers from a computational overhead as the number of pipeline stages increases. To understand this problem we elaborate on how KRJN works in a pipeline of 3 inputs assuming three input streams, L_1, L_2 and L_3. When the top KRJN operator, OP_1, is called for the next top ranked object, several *GetNext* calls for the left and right children are invoked. According to the KRJN algorithm described in Table 3.1, at each step, OP_1 gets the next tuple from its left and right children. Hence, OP_2 will be required to deliver as many top ranked objects of L_2 and L_3 as the number of objects retrieved by L_1. These excessive calls to the ranking algorithm in OP_2 result in retrieving more objects from L_2 and L_3 than necessary and accordingly, result in larger queue sizes and more database accesses.

One solution is to unbalance the depth step in the operator children. We change the KRJN *GetNext* algorithm to reduce the local ranking overhead by changing the way it retrieve tuples from its children; for each p tuples accessed from the right child one tuple is accessed from the left child. The idea is to have less expensive *GetNext* calls to the left child, which is also an KRJN operator. Using different depths in the input streams does not violate the correctness of the algorithm [24], but will have a major effect on the performance. This optimization significantly enhances the performance of the KRJN operator as will be demonstrated in Section 3.4.2. Through the rest of the chapter we will call p the *balancing factor*. Choosing the right p is a design decision and depends on the data and the order of the input streams, but a good choice of p boosts the performance of KRJN.

For example, for a typical query with three ranked inputs, we compare between the total number of accessed tuples by the KRJN operator before and after applying the heuristic. Also, as a reference, we compare the KRJN operator with a direct

Table 3.1
The KRJN GetNext Operation

KRJN**:GetNext()**

```
1.  Threshold = 0;
2.  if (Q is not Empty)
3.      tuple = Q.Top;
4.      W= tuple.WorstGrade;
5.      B_max=Maximum Best Grade in Q-tuple;
6.      if (W ≥ Max(B_max, Threshold))
7.          RETURN tuple;
8.  LOOP
9.      leftTuple = Left.GetNext(depth);
10.     rightTuple = Right.GetNext(depth);
11.     leftBottom = leftTuple.BestGrade;
12.     rightBottom = rightTuple.BestGrade;
13.     Threshold = f(leftBottom,rightBottom);
14.     Check if tuple were seen before
15.     if(tuples exist in Q)
16.         Update Worst grade with exact grade and reinsert tuple;
17.     For each Object in the queue
18.         Update the BestGrade;
19.     tuple = Q.Top;
20.     W= tuple.WorstGrade;
21.     B_max=Maximum Best Grade in Q-tuple;
22.     if(W ≥ Max(B_max, Threshold))
23.         BREAK LOOP;
24. END LOOP;
25. Remove tuple from top of Q;
26. RETURN tuple;
```

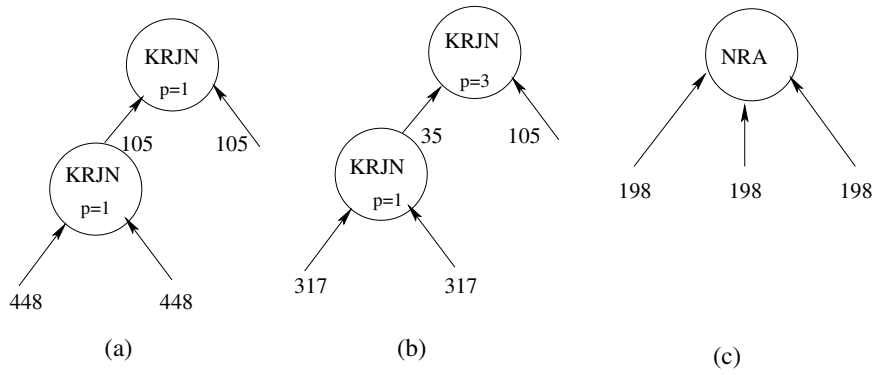


Figure 3.1. The Effect of Applying the Heuristic to Solve the Local Ranking Problem in KRJN

implementation of the NRA Algorithm, the NRA Operator. The NRA operator is a multi-way rank-join operator. Figure 3.1 shows the number of retrieved tuples for each case. In the plan in Figure 3.1 (a), p is set to 1 for both KRJN operators. According to a real data example execution of this query pipeline, the top KRJN operator retrieves 105 tuples from both children, hence the top 105 tuples are requested from the KRJN child operator, which has to retrieve 448 tuples from each of its children, for a total of 1001 tuples. In the plan in Figure 3.1 (b), p is set to 3 for the top KRJN operator. While retrieving the same answers, the total number of tuples retrieved is 739 tuples, which is much less than that of the KRJN before applying the heuristic since the top KRJN operator requested only 35 tuples from its left child. The plan in Figure 3.1 (c) shows the number of tuples retrieved by the NRA operator, which requires 198 tuples from each of its three children for a total of 594 tuples.

3.3 KRJN vs. J*

Both the J^* and the KRJN operators implement a joining algorithm that joins multiple ranked inputs. The two operators are binary and pipelined and can be integrated easily in query evaluation plans. On the other hand, the two operators were

designed for different problem settings. The KRJN operator requires the existence of a *key* as the join attribute. More precisely, the KRJN operator joins two streams of the same objects ordered differently. In contrast, the J^* operator can actually join different objects under arbitrary joining conditions, and hence can be used in a wider range of applications. Since it is more general, the J^* operator suffers from larger space requirements in the worst case, as we will show in the next sections. Both algorithms have proven to be *instance-optimal* with respect to database access cost, where instance optimality is a stronger optimality condition defined by Fagin et al. [24].

In this section we highlight important design differences between the KRJN operator and the J^* operator. Two important design aspects of an operator are the stopping criteria and the space requirements. The stopping condition has a direct effect on the number of database accesses made by the operator. The goal is to stop as soon as we have enough information to report the next top-ranked object. The space requirement is an important design parameter in a database engine, since the maximum space required by an operator is translated into the quantity of resources that must be allocated. We conduct a worst case analysis for both the KRJN and the J^* operators. For completeness, we also provide a best case analysis on the buffer size of both operators. The following comparisons are made for the problem of joining multiple sets of the same objects ordered differently in each ranked input. For arbitrary join conditions the J^* operator becomes the only choice.

3.3.1 Stopping Condition

The idea behind the rank-join algorithms is to stop as early as possible, without the need to actually sort all the streams or visit more objects than needed. The algorithm implemented by the KRJN operator achieves just that by introducing the *Worst Grade* and the *Best Grade* of an object. The idea is to stop when we are guaranteed that this object cannot have less overall score than any other object

in the database, even if not *all* streams have been seen so far. In contrast, the J^* operator requires that an object must be seen in every input stream before reporting it. This constraint can be seen from the algorithm in Table 2.2; only *complete* states on top of the queue can be reported. To illustrate the early stopping criteria of the KRJN we give the following example. Given two ranked inputs $L_1 = (R_1 : 10, R_2 : 5, R_3 : 4, R_4 : 3)$ and $L_2 = (R_2 : 5, R_3 : 4, R_4 : 3, R_1 : 1)$, where each object is attached to a different score in each list (the scores of R_1 are 10 in L_1 and 1 in L_2). Let W_i and B_i denote the worst grade and the best grade of object R_i, respectively. We use a simple monotone function $t(a, b) = a + b$ to calculate the overall score, and after two steps by the KRJN operator, the best grade and the worst grade of objects seen so far are as follows: $W_1 = 10, B_1 = 14$; $W_2 = 10, B_2 = 10$; and $W_3 = 4, B_3 = 9$. According to the stopping criteria of the KRJN operator, both R_1 and R_2 can be reported as the first top objects. Note that object R_1 has not been encountered yet in the input L_2 but it is guaranteed to have a worst grade that is larger than the best grade of any other object. For the same example using the J^* operator, R_1 cannot be reported as an output before accessing all objects in the input list L_2.

3.3.2 Space Complexity

Rank-join algorithms with no random access suffer from the problem of unbounded buffer requirements for tracking the best grade of the objects encountered so far. Thus, the operator may require bookkeeping tasks for a huge queue containing these objects before it can report the next object. When comparing the space requirements of the two operators, a worst case analysis is used to estimate the maximum size of the buffer that should be reserved in order to correctly report the output objects. Due to the ability of the J^* operator to handle general join conditions, it has to consider more join combinations in the maintained priority queue. Hence, the space required by the J^* operator is larger than that required by the KRJN, as shown in the following subsections and in the performance evaluation in Section 3.4.

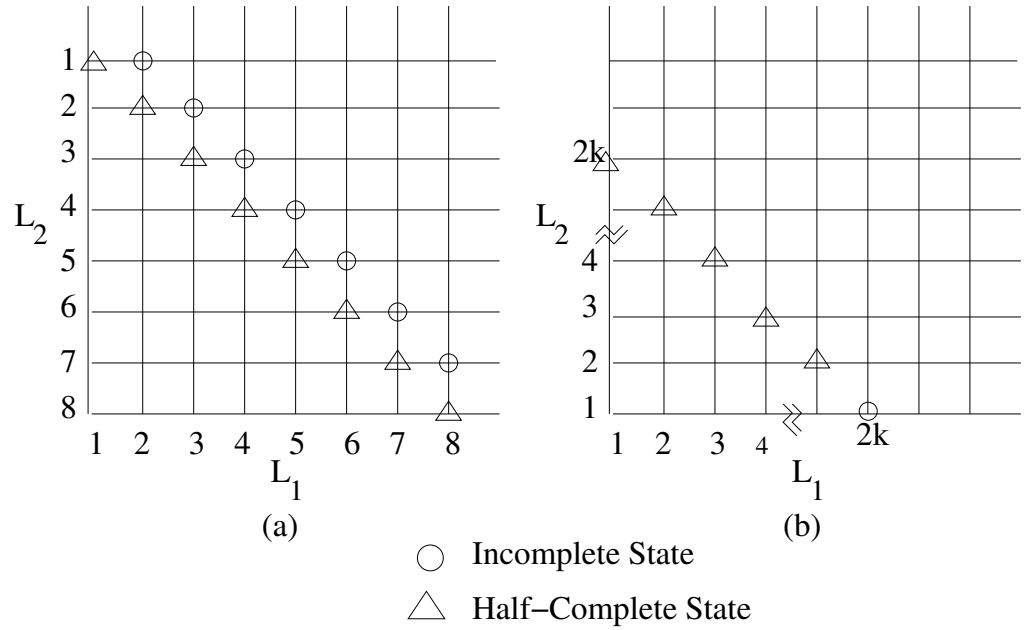


Figure 3.2. Space Complexity of the J^* Operator

Worst Case Analysis: In the worst case, a rank-join algorithm cannot report any object unless *all* objects from the input ranked lists have been seen. Let L_1 and L_2 be the two source rank lists for objects $\{R_1,\ R_2,\ \ldots,\ R_n\}$. For simplicity, let the grade of an object in a list be $n + 1 - rank$, let $L_1 = (R_1,\ R_2,\ R_3,\ \ldots\ ,\ R_n)$ and let $L_2 = (R_n,\ R_{n-1},\ R_{n-2},\ \ldots\ ,\ R_1)$. The grades of object R_1 in lists L_1 and L_2 are n and 1, respectively. Let the weighting function $t(a, b) = a + b$, i.e., a simple monotone function.

In the KRJN Operator, assume we have moved to depth d in the two lists, and that the objects encountered so far from lists L_1 and L_2 are $(R_1,\ R_2,\ \ldots\ ,\ R_d)$ and $(R_n,\ R_{n-1},\ \ldots\ ,\ R_{n-d+1})$, respectively. Our goal is to report the top-most object. The maximum worst grade value encountered so far is the worst grade of object R_1, computed as $W_1 = t(n, 0) = n$. Hence, R_1 is on top of the queue and we can report it only if the maximum best grade for all other objects is less than W_1. The maximum best grade for objects encountered so far (other than R_1) is that of object R_2, computed as $B_2 = t(n-1, n-d+1) = 2n-d$. According to the stopping criteria

of NRA, we can stop only when $2n - d < n$, and that occurs at depth $d = n$, i.e., we must move entirely through both lists with a buffer size of n objects.

In the J^* Operator, the J^* operator solves a more general problem than the KRJN, where it can handle arbitrary join conditions. To be able to compare both operators, we will consider only the case when the join condition is on a key attribute (for example, self-join). In order to see the space complexity of the operator in dealing with the input lists L_1 and L_2, refer to Figure 3.2. In the J^* algorithm with two input lists, a state can be either *complete*, incomplete with one unassigned variable (we will refer to this state as *half-complete*), or incomplete with two unassigned variables (we will refer to this state as *incomplete*). When processing a half-complete state, two states are produced. The first state is a complete state, which is inserted only if it is a valid join combination (when the two variables represent the same object in the case of self-join). The second state is another half-complete state and it is inserted in the queue. In Figure 3.2, triangles represent half-complete states, while circles represent incomplete states. Processing an incomplete state produces two states, a half-complete state and another incomplete state, and both of them are kept in the queue. In the example, when L_1 and L_2 are the two inputs, it is easy to see from Figure 3.2(a) that all valid half-complete states must be present in the queue before reporting any objects (all objects have the same global score). When processing these half-complete states, each state will produce a valid complete state that will be kept in the queue in addition to another half-complete state that is also inserted in the queue, yielding a buffer size of $2n - 1$ states. Given that each state holds two tuples, the total buffer size is $4n - 2$ tuples, which is larger than that of the KRJN operator.

Best Case Analysis: For the best case analysis, we compare the two operators when the two ranked inputs are identical. The KRJN does not need to keep any reported tuples therefore, the buffer size is always *zero*. For the J^* algorithm, the maximum buffer size is twice the size of the required results. To see that, we refer to the previous example with $L_1 = L_2 = (R_1, R_2, \ldots, R_d)$. Figure 3.2(b) shows

the type of the states that can be stored in the buffer at the time the object R_k can be reported. Because the J^* algorithm has to keep all possible combinations in the buffer, the buffer will have at least $2k$ states before reporting the k^{th} object.

3.4 Performance Evaluation

In this section, we experimentally evaluate KRJN through a comparison with the J^* algorithm [29]. The experiments are based on our research platform for a complete video database management system running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The research platform is based on PREDATOR [11], the object relational database system from Cornell University. Shore [12] is the underlying storage manager.

The used data is video visual features stored as high-dimensional *vectors* that must be indexed using a high-dimensional indexing scheme. To accommodate the high-dimensional indexing, we extended the indexing capabilities of Shore by adding the GiST general indexing framework [13]. We used the GiST implementation of the SR-tree [43] as the indexing technique. The nearest-neighbor search operator is implemented as an incremental NN search query on the SR-tree. The following experiments are conducted on the database table *Features*, which contains 100,000 records of features extracted from video frames. The feature fields include color histogram in YUV format (a vector of 32 dimensions), texture tamura (a vector of 16 dimensions) and texture edges (a vector of 9 dimensions). We use the query evaluation plan, given in Figure 3.3, to evaluate the proposed operators. The plan has m NN operators on m different visual features. $m-1$ rank-join binary operators are used, where the results of one operator are pipelined to the next operator in the pipeline.

We use the following query to evaluate the performance of the three rank-join operators:

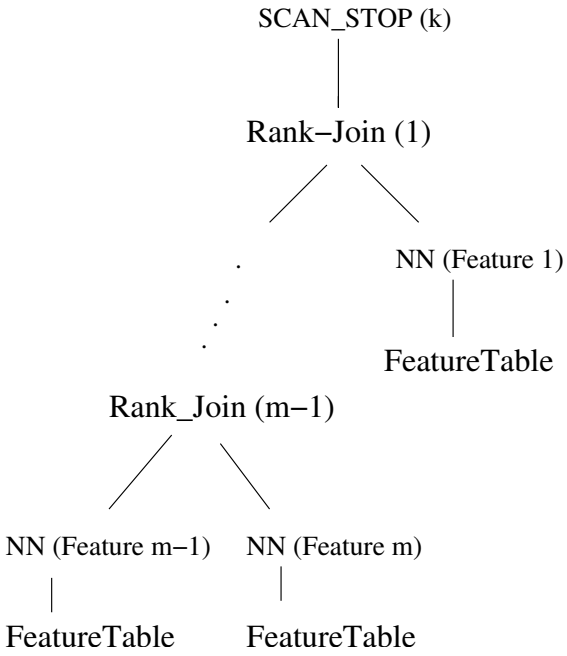


Figure 3.3. The Query Plan Used in the Experiments

Q: Retrieve the k most similar video shots to a given image based on m visual features.

where m varies from 2 to 6 features and k varies from 5 to 100. Note that the number of requested results, k is not an input to the rank-join operator. We limit the number of reported answers to k by applying the *Stop-After* query operator introduced by Carey and Kossmann [20, 21]; this is implemented in the prototype. The physical query operator *Scan-Stop* is a straightforward implementation of Stop-After and appears on top of the query plan given in Figure 3.3.

To evaluate the operators, the following performance measures are chosen:

1. The query running time to retrieve the top matching k output results.

2. The size of the buffer maintained by the operator.

3. The number of database accesses (in disk pages).

While the number of database accesses should give a good indication of the time complexity of the operator, the experiments show a significant CPU time complexity difference between the two operators that affect the total running time, especially for small numbers of inputs as shown by the following experiments. Another interesting set of experiments shows how ordering of the input streams in the pipeline affects the performance. This will have a significant impact on query optimization and the generation of query execution plans for queries involving joining multiple ranked inputs. In our experiments we study the effect of input streams ordering on both the KRJN and the J^* operators.

To compare the two pipelined operators, we implement the non-pipelined version of the NRA algorithm as a multi-way rank-join operator named *NRA*. Although most query optimizers are restricted to binary operators, the performance of the NRA gives useful insight when comparing the two pipelined operators, and gives a reference line for the best possible performance to get the required results.

3.4.1 The Effect of Input Ordering

In this experiment we study the effect of input stream ordering in the pipeline on the performance for both operators. Figure 3.4 gives the performance metrics of the KRJN operator for 6 possible orderings of the input streams in a query pipeline with $m = 4$. Figure 3.5 gives the same metrics for the J^* operator. The results show the sensitivity of the KRJN operator to the ordering of input. This sensitivity can be explained by the excessive local ranking in the query pipeline, and hence, choosing which pairs to rank together plays a major role in getting the final results. The operator J^* is less sensitive to input orderings due to the *guided* fewer invocations of local ranking in the query pipeline. In the experiments in the previous sections, we use the ordering O_1 for both operators. Ordering O_1 shows the best performance in the case of KRJN and the best execution time in the case of J^*.

3.4.2 The Effect of the Balancing Factor

The optimization proposed in Section 3.2 has a significant impact on the KRJN performance and its scalability to long queue pipelines. In Figures 3.7 (a) and (b) we compare the maximum queue size and the number of accessed pages of the KRJN operator for different values of p. The case in which $p = 1$ represents the unoptimized version of the KRJN. The experiment shows that, for small values of k, the performance enhances as p increases. For larger values of k, increasing p results in accessing more tuples from the right child than necessary and hence small values of p gives a better performance.

We measured the effect of choosing p on the scalability of the KRJN operator. Figures 3.8(a) and (b) give the maximum queue size and the number of accessed data pages for different values of m (the length of the pipeline). k is fixed to 20 output results. Also, we compare the performance of KRJN for different values of p. When p is variable, p can have a different value in each pipeline stage. For example, $p = 1$ in the first stage and $p = 2$ in the second stage, etc. The motivation behind having different values for p in different pipeline stages is that the cost of accessing the left child increases as we go up in the query pipeline. A good heuristic is to set p to depend on the pipeline stage. The figures show that this heuristic gives the best performance for $k = 20$. Setting p to 2 enhanced the performance significantly when compared against the unoptimized version when $p = 1$.

Choosing the right p is a design decision and depends on the data and the order of the input streams, but a good choice of p boosts the performance of KRJN.

3.4.3 Real World Data Comparison

Figure 3.6 gives performance comparisons between KRJN (with a balancing factor $p = 2$) and J^* using the best and worst input orderings. We set $m = 3$, where m is the number of input sources that give a pipeline of length $m - 1$. We compare the best/worst input ordering for KRJN obtained from the previous experiment

with the corresponding orderings of the J^* operator. For all orderings, KRJN is much faster than J^* because J^* has to evaluate many unnecessary join combinations (due to its generality). When comparing the space and I/O complexity, a good input ordering for KRJN achieves a close I/O and space overhead complexity to the optimal J^* algorithm.

3.4.4 The Effect of Pipelining

In this experiment, we evaluate how scalable the two pipelined operators are with respect to the length of the query pipeline m. By fixing $k = 20$, the operators KRJN and J^* are compared with respect to the three chosen performance metrics given in Section 3.4. Figure 3.8 compares the performance of the operators KRJN, NRA, and J^* as m increases from 2 to 6. Figure 3.8(a) shows that KRJN is an order of magnitude faster than J^* with respect to the overall running time. The running time of J^* increases drastically with the increase in m making it not scalable for long query pipelines. The total running time of KRJN is as good as that of NRA even for long query pipelines. Figures 3.8(b) and (c) show that both the KRJN and the J^* operators perform similarly with respect to maximum queue size and number of database accesses. The figures also show the effect of the pipelining on both operators as their performance starts to divert from that of NRA as m increases.

3.5 Summary

In this chapter, we introduced a new key-join query operator for efficient evaluation of top-k selection queries. we carried out an extensive performance study to evaluate our proposed operator against the J^* algorithm. Our proposed operator is an adaptation of the No-Random-Access algorithm. We focused on the use of these algorithms as binary pipelined query operators, which makes them practical for most database engines. Several experiments were conducted to illustrate the different per-

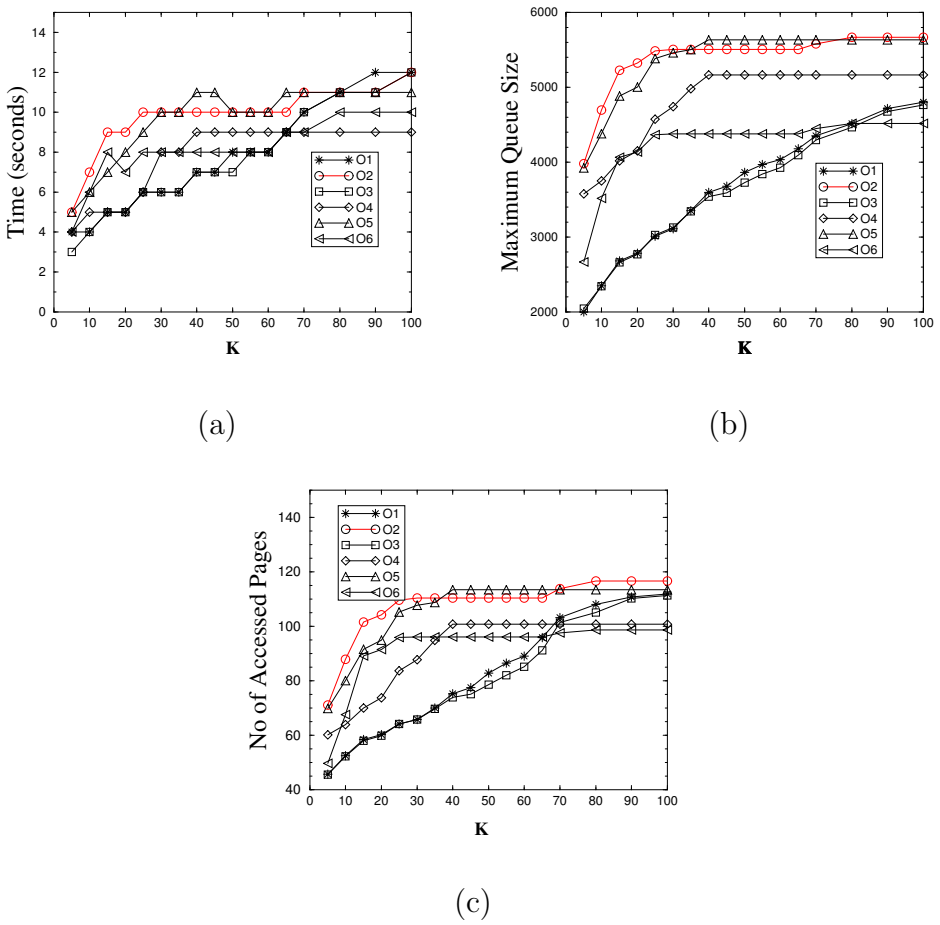


Figure 3.4. The Effect of Input Streams Ordering on KRJN

formance issues and trade-offs. The experiments were in the context of multimedia retrieval and were performed against a continuous media retrieval prototype.

Our study showed the importance of implementing rank-join algorithms as query operators. The performance of the KRJN operator is greatly enhanced through unbalancing the depth step of its inputs to reduce the overhead of local ranking in the earlier pipeline stages. As demonstrated, the optimized KRJN operator is superior over the J^* operator even for large number of ranked inputs. The optimized KRJN operator is an order of magnitude faster than the J^* operator, has less space requirements, and has a comparable number of disk accesses. The performance

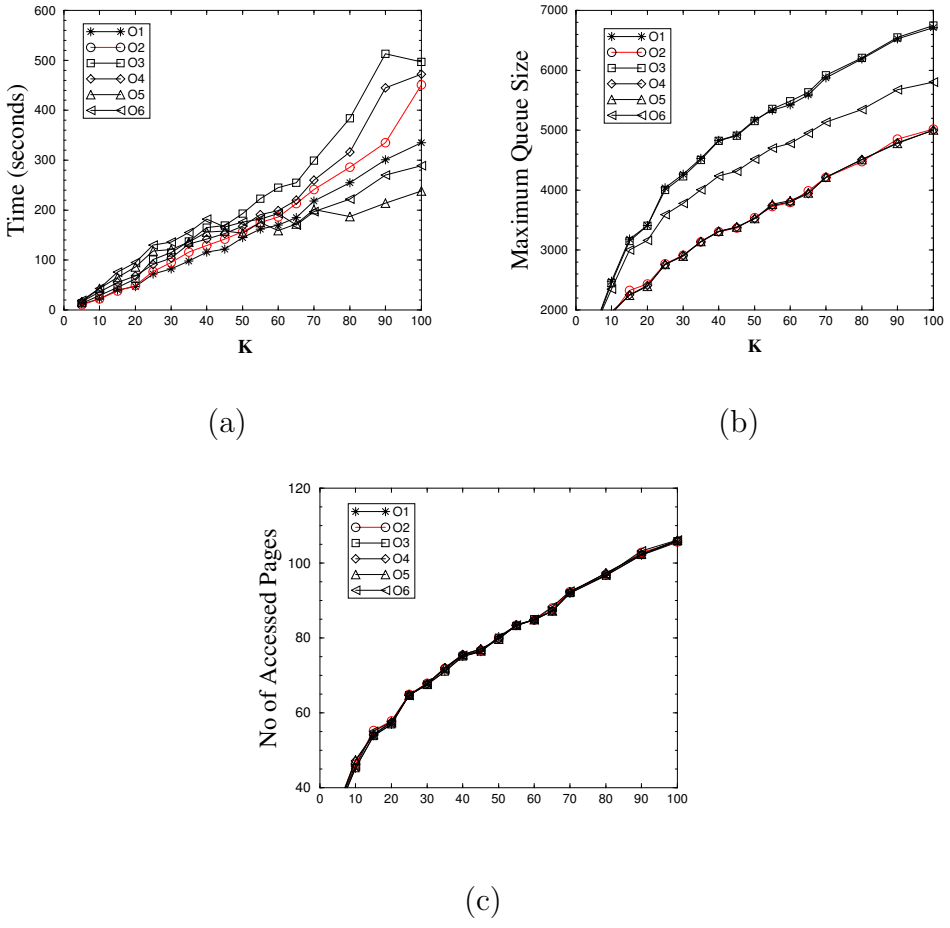
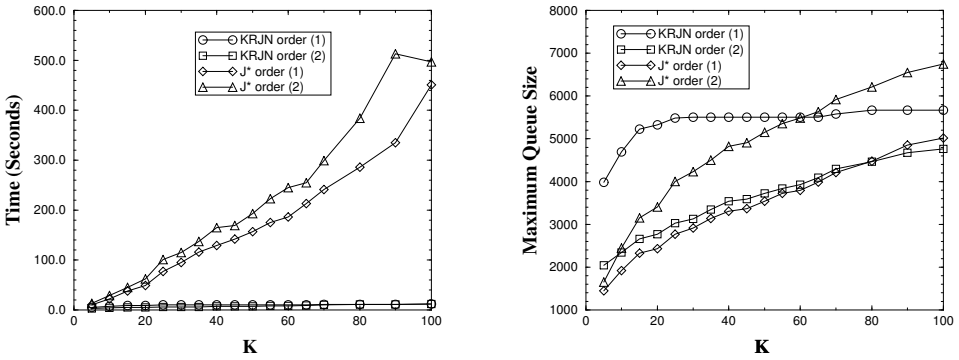
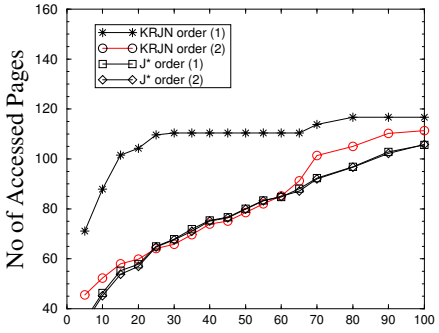


Figure 3.5. The Effect of Input Streams ordering on J^*

study also showed that the KRJN operator is more sensitive to the ordering of the ranked input streams than the J^* operator, which shows less sensitivity. This further motivates for the need to optimize rank-join queries as we will demonstrate in Chapter 5.

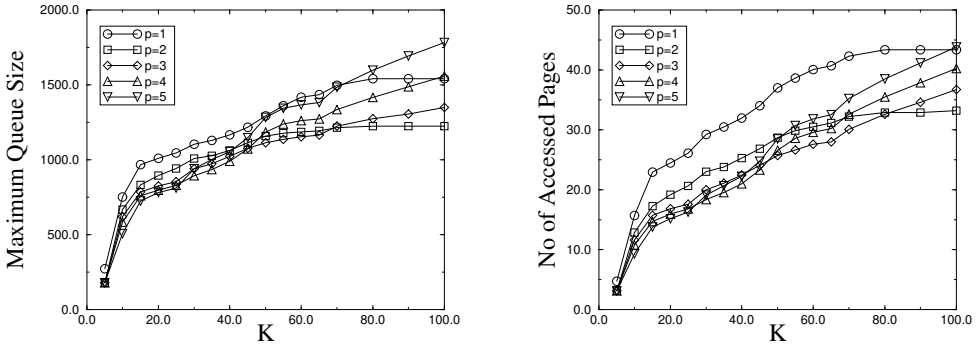


(a) (b)



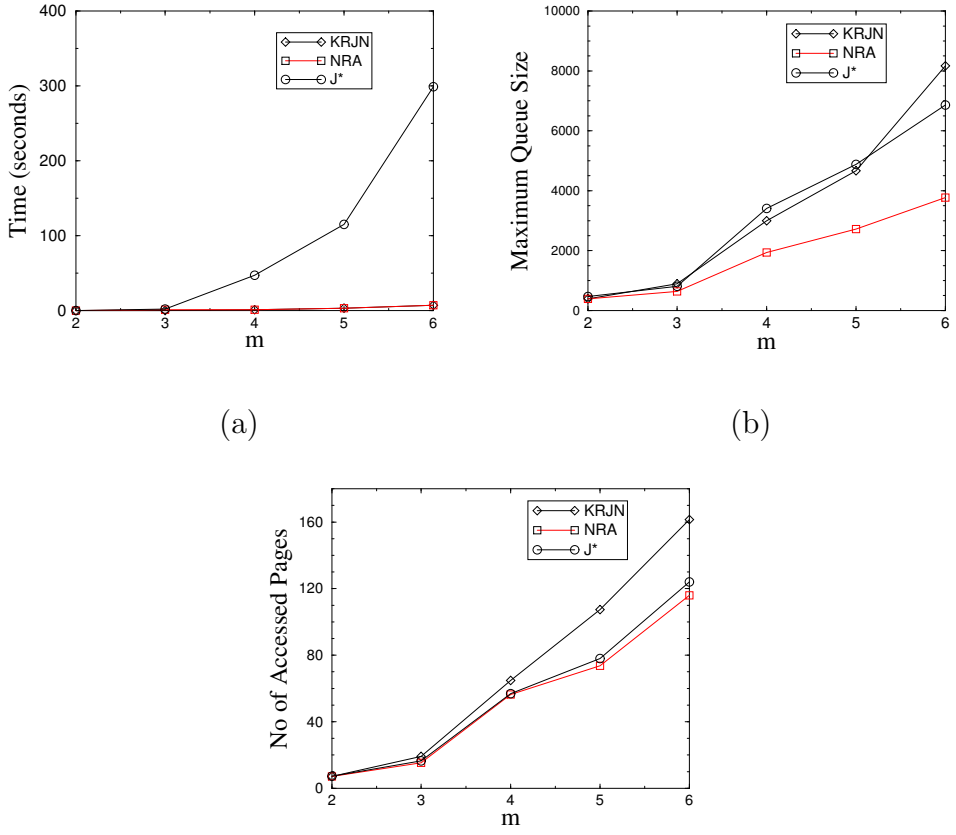
(c)

Figure 3.6. Comparing KRJN and J^* for $m = 3$



(a) (b)

Figure 3.7. The Effect of the Balancing Factor p for $m = 3$



(a) (b)

(c)

Figure 3.8. Scalability of the KRJN Operator

4 TOP-K JOIN ALGORITHMS AND OPERATORS

In this chapter, we introduce a new rank-join algorithm to answer general top-k join queries. We provide efficient implementations of the algorithm in terms of physical binary pipelined query operators that can be integrated easily in current query processors. The introduced rank-join algorithm is independent of the join strategy and is implemented in two physical rank-join operators, HRJN and HRJN*. The performance of these rank-join operators is compared against that of the J^* algorithm [29]. We start by defining the query model and present our approach to support evaluating this type of queries in relational query engines.

The remainder of this chapter is organized as follows. Section 4.1 describes the new rank-join algorithm along with its correctness and optimality proofs. We present two physical rank-join operators in Section 4.3. We introduce an efficient optimization heuristic for rank-join execution plans in Section 4.4. In Section 4.5, we generalize the rank-join algorithm to exploit any available random access to the input relations. Section 4.6 gives the experimental evaluation of the new rank-join operator and compares it with alternative techniques. We conclude in Section 4.7 with a summary and final remarks.

4.1 The New Rank-join Algorithm

Current implementations of the join operator do not make use of the fact that the inputs may be already ordered on their individual scores. Using these individual orderings, we can perform much better in evaluating the top-k join queries by eliminating the need to sort the join results on the combined score.

The join operation can be viewed as the process of spanning the space of Cartesian product of the input relations to get valid join combinations. An important

observation is that, only part of this space needs to be computed to evaluate *top-k join queries*, if we have the inputs ordered individually.

In this section we describe a new join algorithm, termed *rank-join*. The algorithm takes m ranked inputs, a join condition, a combining ranking function f and the number of desired ranked join results k. The algorithm reports the top k ranked join results in descending order of their combined score. The rank-join algorithm works as follows:

- Retrieve objects from the input relations in a descending order of their individual scores. For each new retrieved tuple:

  1. Generate new valid join combinations with all tuples seen so far from other relations, using some join strategy.
  2. For each resulting join combination, J, compute the score $J.score$ as $f(O_1.score, O_2.score, \ldots, O_m.score)$, where $O_i.score$ is the score of the object from the i^{th} input in this join combination.
  3. Let the object $O_i^{(d_i)}$ be the last object seen from input i, where d_i is number of objects retrieved from that input, $O_i^{(1)}$ be the first object retrieved from input i, and T be the maximum of the following m values:
     $f(O_1^{(d_1)}.score, O_2^{(1)}.score, \ldots, O_m^{(1)}.score)$,
     $f(O_1^{(1)}.score, O_2^{(d_2)}.score, \ldots, O_m^{(1)}.score)$,
     $\ldots$,
     $f(O_1^{(1)}.score, O_2^{(1)}.score, \ldots, O_m^{(d_m)}.score)$.
  4. let L_k be a list of the k join results with the maximum combined score seen so far and let $score_k$ be the lowest score in L_k, halt when $score_k \geq T$.

- Report the join results in L_k ordered on their combined scores.

The value T is an upper-bound of the scores of any join combination not seen so far. An object O_i^p, where $p > d_i$, not seen yet from input i, cannot contribute

to any join combination that has a combined score greater than or equal an upper-bound T_i, where $T_i = f(O_1^{(1)}.score, \ldots, O_i^{(d_i)}.score, \ldots, O_m^{(1)}.score)$. The value T is continuously updated with the score of the newly retrieved tuples.

Theorem 4.1.1 *Using a monotone combining function, the described rank-join algorithm correctly reports the top k join results ordered on their combined score.*

Proof: For simplicity, we prove the algorithm for two inputs l and r. The proof can be extended to cover the m inputs case. We assume that the algorithm access the same number of tuples at each step, i.e., $d_1 = d_2 = d$. The two assumptions do not affect the correctness of the original algorithm.

The proof is by contradiction. Assume that the algorithm halts after d sorted accesses to each input and reports a join combination $J_k = (O_l^{(i)}, O_r^{(j)})$, where $O_l^{(i)}$ is the i^{th} object from the left input and $O_r^{(j)}$ is the j^{th} object from the right input. Since the algorithm halts at depth d, we know that $J_k.score \geq T^{(d)}$, where $T^{(d)}$ is the maximum of $f(O_l^{(1)}.score, O_r^{(d)}.score)$ and $f(O_l^{(d)}.score, O_r^{(1)}.score)$. Now assume that there exists a join combination $J = (O_l^{(p)}, O_r^{(q)})$ not yet produced by the algorithm and $J.score > J_k.score$. That implies $J.score > T^{(d)}$, i.e.,

$$f(O_l^{(p)}.score, O_r^{(q)}.score) > f(O_l^{(1)}.score, O_r^{(d)}.score) \tag{4.1}$$

and

$$f(O_l^{(p)}.score, O_r^{(q)}.score) > f(O_l^{(d)}.score, O_r^{(1)}.score) \tag{4.2}$$

Since each input is ranked in descending order of object scores, then $O_l^{(p)}.score \leq O_l^{(1)}.score$. Therefore, $O_r^{(q)}.score$ must be greater than $O_r^{(d)}.score$. Otherwise, Inequality (4.1) will not hold because of the monotonicity of the function f. We conclude that $O_r^{(q)}$ must appear before $O_r^{(d)}$ in the right input, i.e.,

$$q < d \tag{4.3}$$

Using the same analogy, we have $O_r^{(q)}.score \leq O_r^{(1)}.score$. Therefore, $O_l^{(p)}.score$ must be greater than $O_l^{(d)}.score$. Otherwise, Inequality (4.2) will not hold because of the

monotonicity of the function f. We conclude that $O_l^{(p)}$ must appear before $O_l^{(d)}$ in the left input, i.e.,

$$p < d \tag{4.4}$$

From (4.3) and (4.4), if valid, the combination $J = (O_l^{(p)}, O_r^{(q)})$ must have been produced by the algorithm, which contradicts the original assumption. ∎

Theorem 4.1.2 *The buffer maintained by the rank-join algorithm to hold the ranked join results is bounded and has a size that is independent of the size of the inputs.*

Proof: Other than the space required to perform the join, the algorithm needs only to remember the top k join results independent of the size of the input. ∎

Following this abstract description of the rank-join algorithm, we show how to implement the algorithm in a binary pipelined join operator that can be integrated in commercial query engines. Theoretically, any current join implementation can be augmented to support the previously described algorithm. Practically, the join technique greatly affects the performance of the ranking process. We show the effect of the selection of the join strategy on the stopping criteria of the rank-join algorithm.

4.1.1 The Effect of Join Strategy

The order in which the points in the Cartesian space are checked as a valid join result has a great effect on the stopping criteria of the rank-join algorithm. Consider the two relations in Figure 4.1 to be joined with the join condition $L.A = R.A$. The join results are required to be ordered on the combined score of $L.B + R.B$.

Following the new rank-join algorithm, described in Section 4.1, a threshold value will be maintained as the maximum between the two values $f(L^{(1)}.B, R^{(d_2)}.B)$ and $f(L^{(d_1)}.B, R^{(1)}.B)$, where $L^{(d_1)}$ and $R^{(d_2)}$ are the last tuples accessed from L and R, respectively. Figure 4.2 shows two different strategies to produce join results.

Strategy (a) is a nested-loops evaluation while Strategy (b) is a symmetric join evaluation that tries to balance the access from both inputs. To check for possible

id	A	B
1	1	5
2	2	4
3	2	3
4	3	2

L

id	A	B
1	3	5
2	1	4
3	2	3
4	2	2

R

Figure 4.1. Two Example Relations

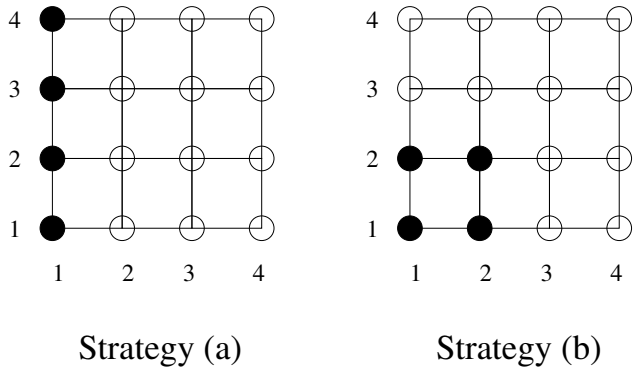


Figure 4.2. Two Possible Join Strategies

join combinations, Strategy (a) accesses four tuples from L and one tuple from R while Strategy (b) accesses two tuples from each relation. The rank-join algorithm at this stage computes a different threshold value T in both strategies. In Strategy (a), $T = max(5 + 2, 5 + 5) = 10$, while in Strategy (b) $T = max(5 + 4, 5 + 4) = 9$. At this stage, the only valid join combination is the tuple pair $[(1, 1, 5), (2, 1, 4)]$ with a combined score of 9. In Strategy (a), this join combination cannot be reported because of the threshold value of 10 while the join combination is reported as the top-ranked join result according to Strategy (b).

The previous discussion suggests using join strategies that reduce the threshold value as quickly as possible to be able to report top ranked join results early on. In

the next section, we present different implementations of the rank-join algorithm by choosing different join strategies.

4.2 Optimality of Algorithm Rank-join

In this section, we analyze the I/O cost of the proposed rank-join algorithm. The notion of *instance optimality* is defined by Fagin et al. [24]. Formally, instance optimality is defined as follows. Let $\mathcal{A}$ be a class of algorithms and let $\mathcal{D}$ be a class of databases. For an algorithm $A \in \mathcal{A}$ and a database $D \in \mathcal{D}$, let $cost(A, D)$ be the total number of I/O accesses incurred by applying A on D. An algorithm B is instance optimal over $\mathcal{A}$ and $\mathcal{D}$ if $B \in \mathcal{A}$ and for every $A \in \mathcal{A}$ and $D \in \mathcal{D}$ we have

$$cost(B, D) = O(cost(A, D))$$

Hence, there exist constants $c, c' > 0$ such that $cost(B, D) \leq c.cost(A, D) + c'$ for every choice of $A \in \mathcal{A}$ and $D \in \mathcal{D}$. The constant c is referred to as the *optimality ratio.*

Theorem 4.2.1 *Let $\mathcal{D}$ be the class of all databases consisting of m sorted relations (ranked lists) and let $\mathcal{A}$ be the class of all correct algorithms that produce the top k ranked join results from these lists. The rank-join algorithm is instance optimal over $\mathcal{A}$ and $\mathcal{D}$.*

Proof: We present the proof in the case of two lists L and R. The proof can be easily generalized to m lists by adjusting the optimality ratio. Refer to Figure 4.3 for illustration. Let l be the top element in L with a score s_l, and let r be the top element in R with a score s_r.

Assume that the rank-join algorithm, when run on $D \in \mathcal{D}$, halts at depth d. Let $A \in \mathcal{A}$ be an arbitrary algorithm. We shall show that Algorithm A must get to depth d in at least one of the lists. It then follows that the rank-join algorithm is instance optimal with optimality ratio at most 2 (assuming two lists). Assume that

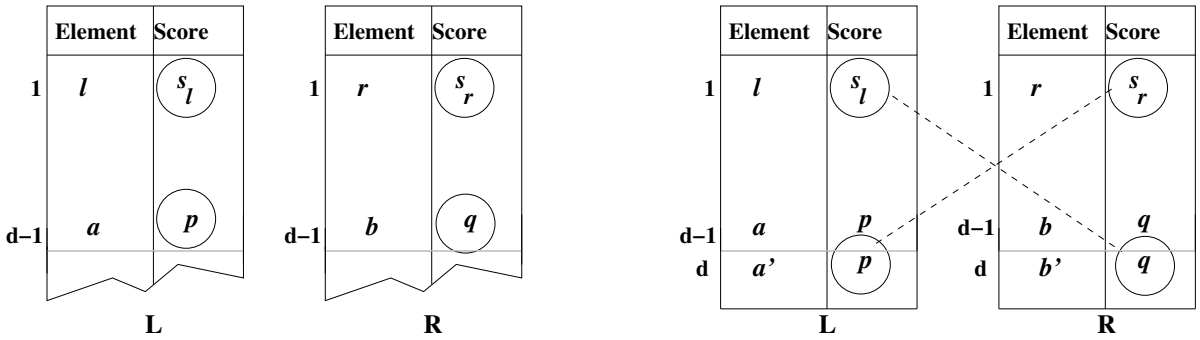


Figure 4.3. Instance Optimality of the Rank-join Algorithm

Algorithm A does not get to depth d in either list; we shall show that Algorithm A makes a mistake on some database.

Let $\mathcal{T}$ be the threshold value at depth $d-1$, where $\mathcal{T}$ is computed as $\mathcal{T} = MAX(f(s_l, q), f(s_r, p))$, where p is the score at depth $d-1$ in L, q is the score at depth $d-1$ in R, and f is the scoring function. Without loss of generality, assume that $f(s_l, q) \geq f(s_r, p)$, hence $\mathcal{T} = f(s_l, q)$.

Since rank-join did not halt at depth $d-1$, there are less than k joinable pairs (a, b) that have been seen by depth $d-1$ whose overall score is at least $\mathcal{T}$. We now construct a database D' on which Algorithm A errors.

Let D' have exactly d elements in each list (so that D' goes only to depth d). Let D' be identical to the original database, D, up to depth $d-1$ in both lists. Hence, Algorithm A performs exactly the same on both D and D'. At depth d, we put a new element a' with score p in the first list and a new element b' with score q in the second list such that b' joins with l (the top element in L). Hence, the join results (l, b') has score $f(s_l, q) = \mathcal{T}$.

Clearly, (l, b') is not on the output list of Algorithm A, since Algorithm A never sees b' before it stops. However, the output list of Algorithm A contains k joinable pairs with less than k having score $\geq \mathcal{T}$. So Algorithm A made a mistake on D'. ∎

Since database accesses are the dominant cost factor in querying large databases, the instance optimality of the rank-join algorithm plays an important role in optimizing top-k queries. Practically, instance optimality of the rank-join algorithm establishes a strong performance guarantee for the rank-join algorithm when compared with any other way to evaluate top-k queries.

4.3 New Physical Rank-join Operators

In this section, we present two alternatives to realize the new rank-join algorithm as a physical join operator. The main difference between the two alternatives is in the join strategy that is used in order to produce valid join combinations. Reusing the current join strategies (nested-loops join, merge join and hash join) results in a poor performance. Nested-loops join will have a high threshold value because we access all the tuples of the inner relation for only one tuple from the outer relation. Merge join requires sorting on the join columns (not the scores) of both inputs and hence cannot be used in the rank-join algorithm. Similarly, hash join destroys the order through the use of hashing when hash tables exceed memory size. The join strategies presented here depend on balancing the access of the underlying relations.

Since the join operation is implemented in most systems as a dyadic (2-way) operator, we describe the new operators as *binary* join operators. Following common query execution models, we describe the new physical join operators in terms of the three basic interface methods *Open*, *GetNext* and *Close*. The *Open* method initializes the operator and prepares its internal state, the *GetNext* method reports the next ranked join result upon each call, and the *Close* method terminates the operator and performs the necessary clean up.

In choosing the join strategy, the discussion in Section 4.1.1 suggests sweeping the Cartesian space in a way that reduces the threshold value. We depend on the idea of ripple join as our join strategy. Instead of randomly sampling tuples from the input relations, the tuples are retrieved in order to preserve ranking. One challenge

is to determine the rate at which tuples are retrieved from each relation. We present two variants of our rank-join algorithm. The two variants are based on adopting two ripple join variants: the hash ripple join and the block ripple join.

4.3.1 Hash Rank-join Operator (HRJN)

HRJNcan be viewed as a variant of the *symmetrical hash join* algorithm [37, 38] or the hash ripple join algorithm [36]. The *Open* method is given in Table 4.1. The HRJNoperator is initialized by specifying four parameters: the two inputs, the join condition, and the combining function. Any of the two inputs or both of them can be another HRJN operator [1]. The join condition is a general equality condition to evaluate valid join combinations. The combining function is a monotone function that computes a global score from the scores of each input. The *Open* method sets the state and creates the operator internal state which consists of three structures. The first two structures are two hash tables, i.e., one for each input. The hash tables hold input tuples seen so far and are used in order to compute the valid join results. The third structure is a priority queue that holds the valid join combinations ordered on their combined score. The *Open* method also calls the initialization methods of the inputs.

The *GetNext* method encapsulates the rank-join algorithm and is given in Table 4.2. The algorithm maintains a *threshold* value that gives an upper-bound of the score of all join combinations not yet seen. To compute the threshold, the algorithm remembers the two top scores and the two bottom scores (last scores seen) of its inputs. These are the variables L_{top}, R_{top}, L_{bottom} and R_{bottom}, respectively. L_{bottom} and R_{bottom} are continuously updated as we retrieve new tuples from the input relations. At any time during execution, the threshold upper-bound value (T) is computed as the maximum of $f(L_{top}, R_{bottom})$ and $f(L_{bottom}, R_{top})$.

[1] Because HRJN is symmetric, we can allow pipelined bushy query evaluation plans.

Table 4.1
The HRJN *Open* Operation

HRJN: **Open**(L,R,C,f)
Input L,R: Left and right ranked input
C: join condition.
f: monotone combining ranking function.
1. Allocate a priority queue Q;
2. Build two hash tables for L and R;
3. Set the join condition to C;
4. Set the combining function to f;
5. Threshold = 0;
6. L.Open();
7. R.Open();

The algorithm starts by checking if the priority queue holds any join results. If exists, the score of the top join result is checked against the computed threshold. A join result is reported as the next *GetNext* answer if the join result has a combined score greater than or equal the threshold value. Otherwise, the algorithm continues by reading tuples from the left and right inputs and performs a symmetric hash join to generate new join results. For each new join result, the combined score is obtained and the join result is inserted in the priority queue. In each step, the algorithm decides which input to poll. This gives the flexibility of optimizing the operator to get faster results depending on the joined data. A straight forward strategy is to switch between left and right input at each step.

4.3.2 Local Ranking in HRJN

Implementing the rank-join algorithm as a binary pipelined query operator raises several issues. We summarize the differences between HRJN and the logical rank-join algorithm as follows:

Table 4.2
The HRJN *GetNext* Operation

HRJN: **GetNext()**

```
1.  if (Q is not empty)
2.      tuple = Q.Top;
3.      if (tuple.score ≥ T)
4.          RETURN tuple;
5.  LOOP
6.      Determine next input to access, I; (Section 4.3.3)
7.      tuple= I.GetNext();
8.      if (I_firstTuple)
9.          I_top = tuple.score;
10.         I_firstTuple = false;
11.     I_bottom = tuple.score;
12.     T = MAX(f(L_top, R_bottom), f(L_bottom, R_top));
13.     insert tuple in I Hash table;
14.     probe the other hash table with tuple;
15.     For each valid join combination
16.         Compute the join result score using f;
17.         Insert the join result in Q;
18.     if (Q is not empty)
19.         tuple = Q.Top;
20.         if (tuple.score ≥ T)
21.             BREAK LOOP;
22. END LOOP;
23. Remove tuple from Q;
24. RETURN tuple;
```

- The total space required by HRJN is the sum of two hash tables and the priority queue. In a system that supports symmetrical hash join, the extra space required is only the size of the priority queue of join combinations. As shown in Section 4.1, in the proposed rank-join algorithm (with all inputs

processed together), the queue buffer is bounded by k, the maximum number of ranked join results that the user asks for. In this case, the priority queue will hold only the top-k join results. Unfortunately, in the implementation of the algorithm as a pipelined query operator, we can only bound the queue buffer of the top HRJN operator since we do not know in advance how many partial join results will be pulled from the lower-level operators. The effect of pipelining on the performance is addressed in the experiments in Section 4.6.

- Realizing the algorithm in a pipeline introduces a computational overhead as the number of pipeline stages increases. To illustrate this problem, we elaborate on how HRJN works in a pipeline of three input streams, say L_1, L_2 and L_3. When the top HRJN operator, OP_1, is called for the next top ranked join result, several *GetNext* calls from the left and right inputs are invoked. According to the HRJN algorithm, described in Table 4.2, at each step, OP_1 gets the next tuple from its left and right inputs. Hence, OP_2 will be required to deliver as many top partial join results of L_2 and L_3 as the number of objects retrieved by L_1. These excessive calls to the ranking algorithm in OP_2 result in retrieving more objects from L_2 and L_3 than necessary, and accordingly larger queue sizes and more database accesses. We call this problem the *Local Ranking* problem.

Solving The Local Ranking Problem: Another version of ripple join is the blocked ripple join [36]. At each step, the algorithm retrieves a new block of one relation, scans all the old tuples of the other relation, and joins each tuple in the new block with the corresponding tuples there. We utilize this idea to solve the local ranking problem by unbalancing the retrieval rate of the inputs. We issue less expensive *GetNext* calls to the input with more HRJN operators in its subtree of the query plan. For example, in a left-deep query execution plan, for each p tuples accessed from the right input, one tuple is accessed from the left input. The idea is to have less expensive *GetNext* calls to the left child, which is also an HRJN operator.

This strategy is analogous to the block ripple join algorithm, having the left child as an outer and the right child as inner with a block of size p. Using different depths in the input streams does not violate the correctness of the algorithm, but will have a major effect on the performance. This optimization significantly enhances the performance of the HRJN operator as will be demonstrated in Section 4.6. Through the rest of this chapter, we call p the *balancing factor*. Choosing the right value for p is a design decision and depends on the generated query plan, but a good choice of p boosts the performance of HRJN.

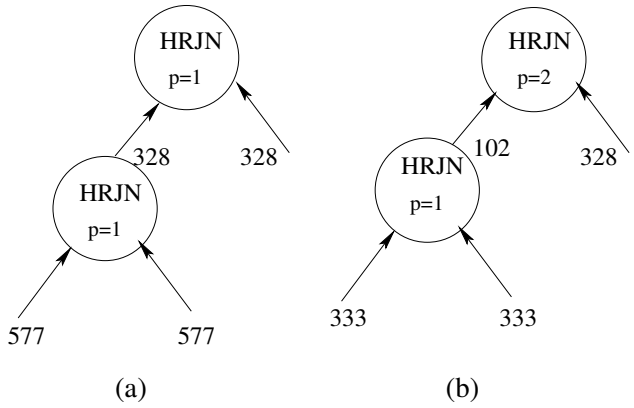


Figure 4.4. The Effect of Applying the Heuristic to Solve the Local Ranking Problem in HRJN

For example, in a typical query with three ranked inputs, we compare between the total number of accessed tuples by the HRJN operator before and after applying the heuristic. Figure 4.4 shows the number of retrieved tuples for each case. In the plan in Figure 4.4 (a), p is set to 1 for both HRJN operators. This query pipeline is applied on real data to retrieve the top 50 join results. The top HRJN operator retrieves 328 tuples from both inputs, hence the top 328 partial join results are requested from the HRJN child operator. The child HRJN operator has to retrieve 577 tuples from each of its inputs, for a total of 1482 tuples. In the plan in Figure 4.4 (b), p is set to 2 for the top HRJN operator. While retrieving the same answers, the total number of tuples retrieved is 994 tuples, which is much less

than that of the HRJN before applying the heuristic, since the top HRJN operator requested only 102 tuples from its left child.

4.3.3 HRJN*: Score-Guided Join Strategy

As discussed in Section 4.1.1, the way the algorithm schedules the next input to be polled can affect the operator response time significantly. One way is to switch between the two inputs at each step. However, this balanced strategy may not be the optimal. Consider the two relations L and R to be rank-joined. The scores from L are $100, 50, 25, 10 \ldots$ while the scores from R are $10, 9, 8, 5, \ldots$. After 6 steps using a balanced strategy (three tuples from each input) we will have the threshold of $max(108, 35) = 108$. On the other hand, favoring R by retrieving more tuples from R than L (four tuples from R and two tuples from L) will give a threshold of $max(105, 60) = 105$.

One heuristic is to try to narrow the gap between the two terms in computing the threshold value. Recall that the threshold is computed as the maximum between two virtual scores T_1 and T_2, where $T_1 = f(L_{top}, R_{bottom})$ and $T_2 = f(L_{bottom}, R_{top})$. f is the ranking function. If $T_1 > T_2$ more inputs should be retrieved from R to reduce the value of T_1 and hence the value of the threshold, leading to possible faster reporting of ranked join results.

This heuristic will cause the join strategy to adaptively switch between the hash join and nested-loops join strategies. Consider the previous example, since $T_1 > T_2$, more tuples will be retrieved from R till the end of that relation. In this case, L_{top} can be reduced to 50. In fact, because all the scores in L are significantly higher than R, the strategy will behave exactly like a nested-loops join. On the other extreme, if the scores from both relations are close, the strategy will behave as a symmetric hash join with equal retrieval rate. Between the two extremes, the strategy will gracefully switch between nested-loops join and hash join to reduce the threshold value as quickly as possible. Of course, this heuristic does not consider

the I/O and memory requirements that may prefer one strategy on the other. In the experimental evaluation of our approach, discussed in Section 4.6, we implement the new join strategy using the HRJN operator. We call the enhanced operator HRJN*. HRJN* shows better performance than those of other rank-join operators including the original HRJN.

4.4 Choosing the Best Join Order

As described in Section 4.3, the join operation is implemented in most systems as a dyadic (2-way) operator for flexibility and practical implementation reasons. Hence, rank-join operators, e.g., HRJN are implemented as *binary* join physical operators. To rank-join n ranked inputs, the inputs are organized in a pipelined query evaluation plan in the form of a binary tree. The evaluation plan determines the order at which we carry out the rank-join operations. In this section we highlight the effect of join order on the overall performance of top-k join queries. We propose an optimization heuristic to choose the best join order based on sampling.

Consider the following example to join three ranked inputs A, B, and C. Figure 4.5 gives the number of retrieved records from each input to report the top ten join results. Figure 4.5 also gives three different join orders (evaluation plans) for the rank-join operations among A, B, and C. The figure shows that the join order significantly affects the number of retrieved records from the inputs. For example, we save significant number of I/O accesses by changing the join order from Plan (b) to Plan (c). For Plan (b), the number of records retrieved from A, B, and C are 499, 499, and 217 records, respectively, with a total of 1215 records. For Plan (c), the number of records retrieved from A, B, and C are 324, 324, and 218 records, respectively, with a total of 866 records, which is 70% of the inputs required for Plan (b).

The main reason for the effect of the join order on the size of required input—and hence the performance of the rank-join operation—is the *correlation* or the

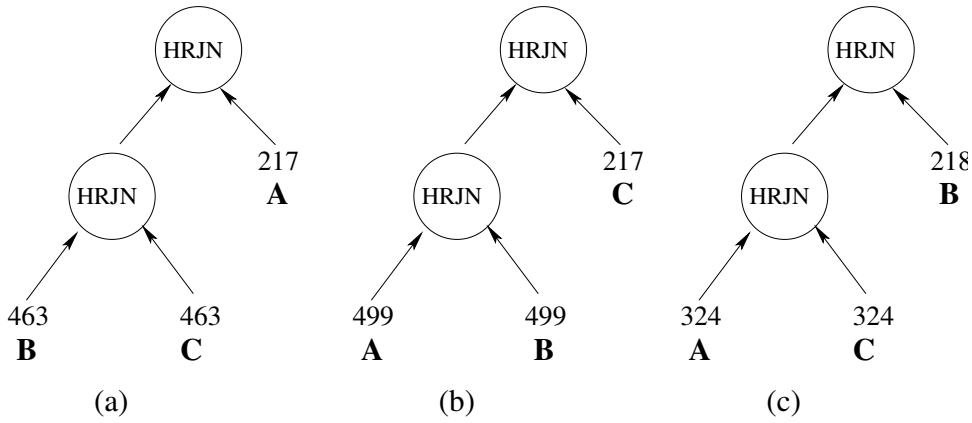
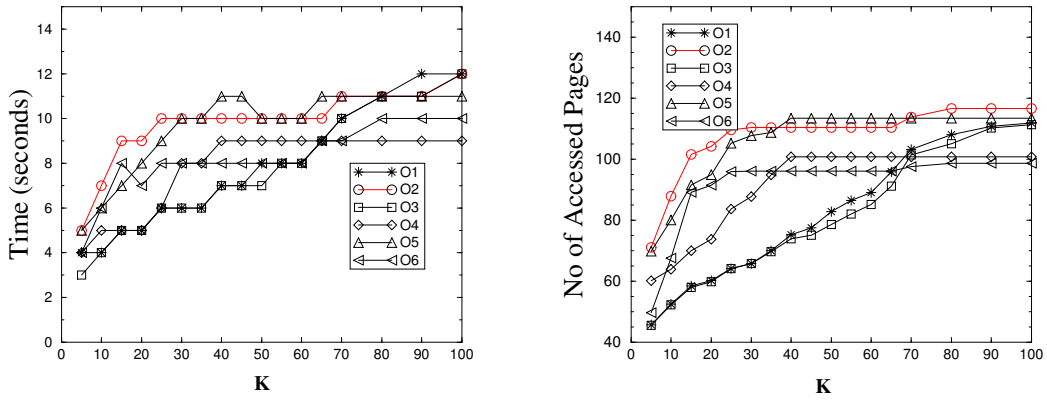


Figure 4.5. The Effect of the Join Order on the Size of Required Input

similarity among the input rankings. In Figure 4.5, the degree of similarity between the rankings of A and C is higher than that between the rankings of A and B. Hence, a rank-join between A and C is likely to require less number of records than a rank-join between A and B to produce the same number of ranked results. Like traditional query optimization, the main goal of optimizing rank-join queries is to choose the best join order. Unlike traditional optimization, the size of the inputs involved in the rank-join operation is not known apriori. Hence, it is hard to estimate the cost of a rank-join operation. Figure 4.6 gives actual total execution time and the number of I/O accesses to rank-join four ranked inputs with six different join orders. The figure shows the significant impact of the order on the overall performance of the rank-join operation.

An optimal query execution plan is the plan with the cheapest overall cost, where the cost includes various components, e.g., the I/O complexity, and the memory usage. Since the number of retrieved input records greatly affects the I/O and time complexities of the rank-join operation, we give a definition of an optimal rank-join order. For simplicity, the definition assumes that the I/O cost to retrieve a record is the same for all the inputs. This simplification can be easily relaxed by using a different weight for each input.



(a) (b)

Figure 4.6. The Effect of the Join Order on the Performance of Rank-join Operators

Definition 4.4.1 *Optimal rank-join order: A rank-join order is optimal if it requires the least number of input records to produce the same number of ranked join results.*

Computing the optimal rank-join order is hard and is even impossible in certain situations for the following reasons:

- Determining the number of input records needed to produce the top k join results requires a complete knowledge of the score distribution of each input and a well-defined notion of similarity measure among input rankings.
- There is no clear way of estimating the number of required join results, k, when pushed down in a plan pipeline. For example, in Plan (c) of Figure 4.5, while $k = 10$ for the top HRJN operator, $k = 111$ for the left child HRJN operator. The value of k in each rank-join operator depends on the final value of k (specified in the user query), the rank-join strategy, and the score distribution of the input.
- Assuming that there is a way to compute and use these statistics to estimate the input size, the input itself may not be available off-line. For example,

the input rankings may be computed as the output of single-feature similarity subqueries (refer to Examples 1.2.1 and 1.2.2).

In the following, we propose a simple yet efficient heuristic to choose a "good" rank-join order based on sampling.

4.4.1 Rank-join Order Heuristic

The main idea of the proposed heuristic is to push the rank-join of *similar* rankings as early as possible in the query evaluation plan. We describe the intuition behind this heuristic as follows:

- Since the number of required results from each rank-join operator increases as we go down in the query pipeline, we aim at making early rank-join operations (deep in the query plan) as fast as possible.

- The best case scenario for the rank-join algorithm occurs when joining identical ranked inputs. We can easily show that the algorithm performance deteriorates, i.e., requires more input before termination, as the similarity between the input rankings decreases.

The proposed technique depends on two main steps: first, obtaining a *ranked* sample of size S from each input, and second, having a well-defined notion for the similarity between two rankings. The first step depends on the type of the input rankings. In general, input rankings can be in one of the following two forms:

- *Available off-line as regular database relations*: in this case the ranked sample is the top S records from the inputs and is available statically without the need to run the whole (or part) of the top-k query.

- *Dynamically computed input*: examples of this category is the output of single-feature similarity queries, or through pulling ranked results from an external source (e.g., a website). In this case we need to run *warm-up* subqueries on the

inputs. A warm-up subquery is a single-feature top-S query on each individual input.

In both cases, the ranked samples are ranked "lists" of the top S objects from each individual ranking.

We define a similarity measure between two rankings based on the *footrule distance* [31,32] between those two rankings. The footrule distance between two ranked lists L and R *over the same set of objects* is defined as $F(L,R) = \sum_i |L(i) - R(i)|$, where $L(i)$ and $R(i)$ is the *rank* of Object i in L and R, respectively. For two input rankings (possibly on different sets of objects), with a join condition to join objects from the first input with objects from the second input, we generalize the distance metric $F(L,R)$ to $F(L,R) = \sum_{i,j} |L(i) - R(j)|$, where (i,j) is a valid join result that joins Object i from L with Object j from R.

Using the ranked sample and the definition of the distance metric, $F(L,R)$, we layout the rank-join order technique in Table 4.3. The technique is an adaptation of Kruskal's minimum spanning tree algorithm to build the final rank-join evaluation plan.

The algorithm in Table 4.3 starts by building a graph structure that represents the similarity measure among all inputs. An edge in the graph connects two vertices, each representing an input ranking list, where a join condition exists between these two inputs. The edge is labeled by the value of the distance metric $F(L,R)$, described earlier. The final rank-join evaluation plan, say P, is built bottom-up by choosing the next most similar pair of inputs from the graph. If none of the two chosen inputs exists in the current plan, a new subplan, P' is formed by providing these two inputs as the inputs to a rank-join operator. P' is joined to the current evaluation plan, P, through building a new rank-join root operator; the inputs of the new root are the current evaluation plan, P, and the new formed subplan, P'. If only one input from the pair of chosen inputs does not exist in P, this input is joined to the plan using a rank-join operator. Note that because of the symmetry of rank-join operators, we do not distinguish between the left and right child while building P.

Table 4.3
The Rank-join Order Algorithm

Rank-join Order($L_1, L_2, \ldots, L_m$, JCs)

Input $L_1, \ldots, L_m$: m input ranked lists of size S

JCs: a set of join conditions between each pair of inputs.

Output P: a query plan to rank-join the inputs

1. Compute $F(L_i, L_j)$ for each pair of inputs L_i and L_j
2. Define a graph $G = (V, E)$ as follows:
3. Each input ranked list represents one vertex in V
4. Edge (L_i, L_j) exists if there exists a join condition in JCs between L_i and L_j
5. Each edge (L_i, L_j) is labeled with $F(L_i, L_j)$
6. Let $T = \{\}$ be a set of graph vertices
7. LOOP while E is not empty
8. Choose the edge $e = (L_i, L_j)$ with the least value of $F(L_i, L_j)$.
9. Remove e from E
10. if $L_i \in T$ and $L_j \in T$, then ignore e
11. else if $L_i \notin T$ and $L_j \notin T$, then:
12. Form a subplan P' that rank-joins L_i and L_j
13. if $P = NULL$, then $P = P'$
14. else let P'' be a subplan that rank-joins P and P'; and set $P = P''$
15. $T = T \cup \{L_i, L_j\}$
16. else if $L_i \notin T$, (same for L_j) then:
17. let P'' be a subplan that rank-joins P andL_i; and set $P = P''$
18. $T = T \cup \{L_i\}$
19. END LOOP

Figure 4.7 gives a real example of applying the rank-join order algorithm in Table 4.3 on 4 inputs. The sample size $S = 100$ records from each input. The join condition is an equi-join on the object id from each list. The labels on the graph edges represent the ranking distance as described earlier in this section. First, the algorithm chooses the two inputs B and D, since they have the least distance value, 7174. The current evaluation plan $P = P1$ is a rank-join operator that joins B and

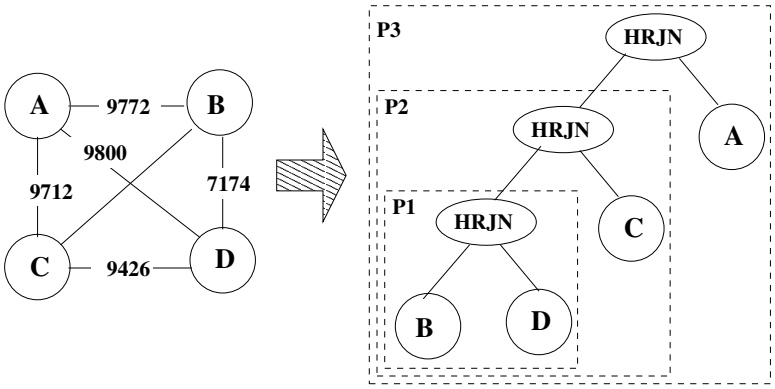


Figure 4.7. Example Execution of the Rank-join Order Algorithm

D. Since the pair of inputs C and D has the next smallest distance value, C is joined to $P1$ to form the evaluation plan $P2$. Finally, the next most similar two inputs are A and C. Hence, A is joined to $P2$ to form the final evaluation plan $P3$. Other edges in the graph are ignored since we consumed all the inputs.

4.5 Generalizing Rank-Join to Exploit Random Access Capabilities

The new rank-join algorithm and query operators assume only sorted access to the input. Random access to some of these inputs is possible when indexes exist. Making use of these indexes may give better performance depending on the type of the index and the selectivity of the join operation. We would like to give the optimizer the freedom to choose whether to use indexes given the necessary cost parameters.

In this section, we generalize the rank-join algorithm to make use of the random access capabilities of the input relations. The main advantage to using random access is to further reduce the upper-bound of the score of unseen join combinations, and hence being able to report the top-k join results earlier. For simplicity, we present the algorithm by generalizing the HRJN operator to exploit the indexes available on the join columns of the ranked input relations. Consider two relations L and

R, where both L and R support sorted access to their tuples. Depending on index existence, we have two possible cases. The first case is when we have an index on only one of the two inputs, e.g., R. Upon receiving a tuple from L, the tuple is first inserted in L's hash table and is used to probe the R index. This version can be viewed as a hybrid between hash join and index nested-loops join. The second case is when we have an index on each of the two inputs. Upon receiving a tuple from $L(R)$, the tuple is used to probe the index of $R(L)$. In this case, there is no need to build hash tables.

On-the-fly Duplicate Elimination: The generalization, as presented, may cause duplicate join results to be reported. We eliminate the duplicates on-the-fly by checking the combined score of the join result against the upper-bound of the scores of join results not yet produced. Consider the two relations L and R with an index on the join column of R. A new tuple from L, with score L_{bottom}, is used to probe R's index and generate all valid combinations. A new tuple from R, with score R_{bottom}, is used to probe the L's hash table of all *seen* tuples from L. A key observation is that any join result, not yet produced, cannot have a combined score greater than $U = f(L_{bottom}, R_{bottom})$. Notice that L_{bottom} is an upper-bound of all the scores from L not yet seen. All join combinations with scores greater than U were previously generated by probing R's index. Hence, A duplicate tuple can be detected and eliminated on-the-fly if it has a combined score greater than U. A similar argument holds for the case when both L and R have indexes on the join columns. One special case is when the two new tuples from L and R can join. In this case, only one of them is used to probe the other relation.

Faster Termination: Although index probing looks similar to hash probing in the original HRJN algorithm in Table 4.2, it has a significant effect on the threshold values. The reason is that since the index contains all the tuples from the indexed relation (e.g., L), the tuple that probes the index from the other relation (e.g., R) cannot contribute to more join combinations. Consequently, the top value of

Relation R should be decreased to the score of the next tuple. For example, for the two ranked relations L and R in Figure 4.1, assume that relation R has an index on the join column to be exploited by the algorithm. In the first step of the algorithm, the first tuple from L is retrieved:$(1, 1, 5)$. We use this tuple to probe the index of R, the resulting join combination is $[(1, 1, 5), (2, 1, 4)]$. Since the tuple from L cannot contribute to other join combinations, we reduce the value L_{top} to be that of the next tuple $(2, 2, 4)$, i.e., 4. In this case we always have $L_{top} = L_{bottom}$ which may reduce the threshold value $T = max(L_{top} + R_{bottom}, L_{bottom} + R_{top})$. Note that if no indexes exists, the algorithm behaves exactly like the original HRJN algorithm.

4.6 Performance Evaluation

In this section, we experimentally evaluate the proposed rank-join operators through a comparison with the J^* algorithm [29]. The experiments are based on our research platform for a complete video database management system running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The research platform is based on PREDATOR [11], the object relational database system from Cornell University. Shore [12] is the underlying storage manager.

We use a set of synthetic tables that have the schema *(Id, JC, Score, Other Attributes)*. Each table is accessed through a sorted access plan and tuples are retrieved in a descending order of the *Score* attribute. *JC* is the join column (not a key) having D distinct values.

We compare the two rank-join operators, HRJN and HRJN* introduced in Section 4.3, with another rank-join operator based on the J^* algorithm. We use a simple ranking query that joins four tables on the non-key attribute JC and retrieves the join results ordered on a simple function. The function combines individual scores which in this case a weighted sum of the scores (w_i is the weight associated with input

i). Only the top k results are retrieved by the query. The following is a SQL-like form of the query:

```
Q:  SELECT T1.id, T2.id, T3.id, T4.id
    FROM T1, T2, T3, T4
    WHERE T1.JC=T2.JC and
          T2.JC=T3.JC and
          T3.JC=T4.JC
    ORDER BY w1*T1.Score + w2*T2.Score +
             w3*T3.Score + w4*T4.Score
    STOP AFTER k;
```

One pipelined execution plan for the query **Q** is the left-deep plan, *Plan A*, given in Figure 4.8. We limit the number of reported answers to k by applying the *Stop-After* query operator [20, 21]. The operator is implemented in the prototype as a physical query operator *Scan-Stop*, a straightforward implementation of Stop-After and appears on top of the query plan. *Scan-Stop* does not perform any ordering on its input.

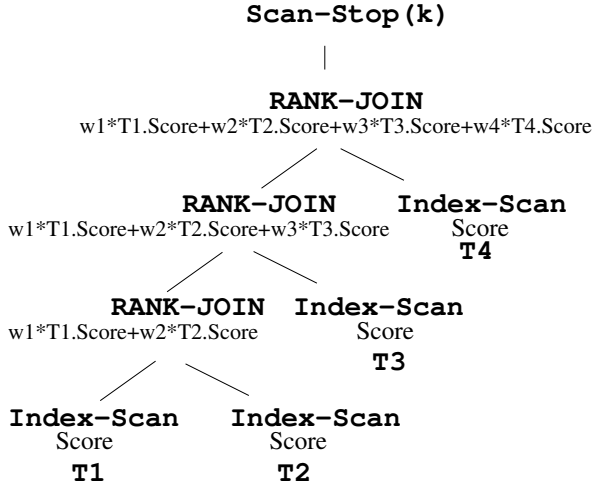


Figure 4.8. *Plan A*: A Left-deep Execution Plan for **Q**

A Pipelined Bushy Tree: *Plan A* is a typical pipelined execution plan in current query optimizers. *Plan B* is a bushy execution plan given in Figure 4.9. Note that bushy plans are not pipelined in current query processors because of the current join implementations. Because rank-join is a symmetric operation, a bushy execution plan can also be pipelined. The optimizer chooses between these plans depending on the associated cost estimates.

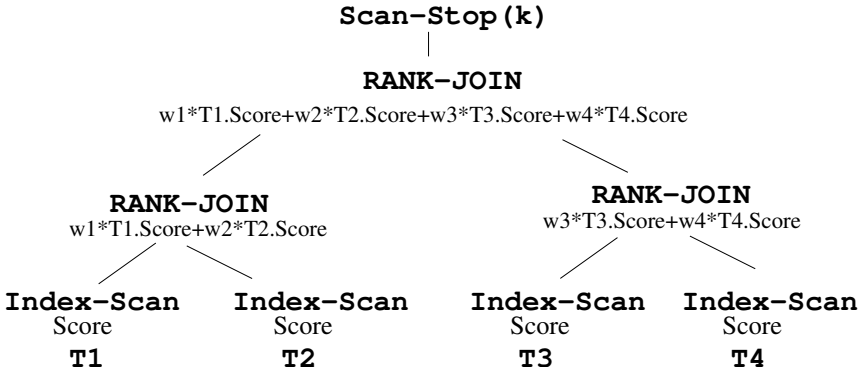


Figure 4.9. *Plan B*: A Bushy Execution Plan for **Q**

Plan B does not suffer from the local ranking problem, described in Section 4.3.2, because each operator has almost the same cost for accessing both of its inputs (same number of plan levels). However, having large variance of the score values between inputs, retrieving more inputs from one side may result in a faster termination. This is a typical case where the operator HRJN* can perform better, because HRJN* uses input scores to guide the rate at which it retrieves tuples from each input. In the following experiments, we use *Plan A* as the execution plan for **Q**. Using *Plan B* gave similar performance results.

Changing the number of required answers: In this experiment, we vary the number of required answers, k, from 5 to 100 while fixing the join selectivity to 0.2%. Figure 4.10 (a) compares the total time to evaluate the query. HRJN and HRJN* show a faster execution by an order of magnitude for large values of k. The high CPU complexity of the J^* algorithm is because it retrieves one join combination

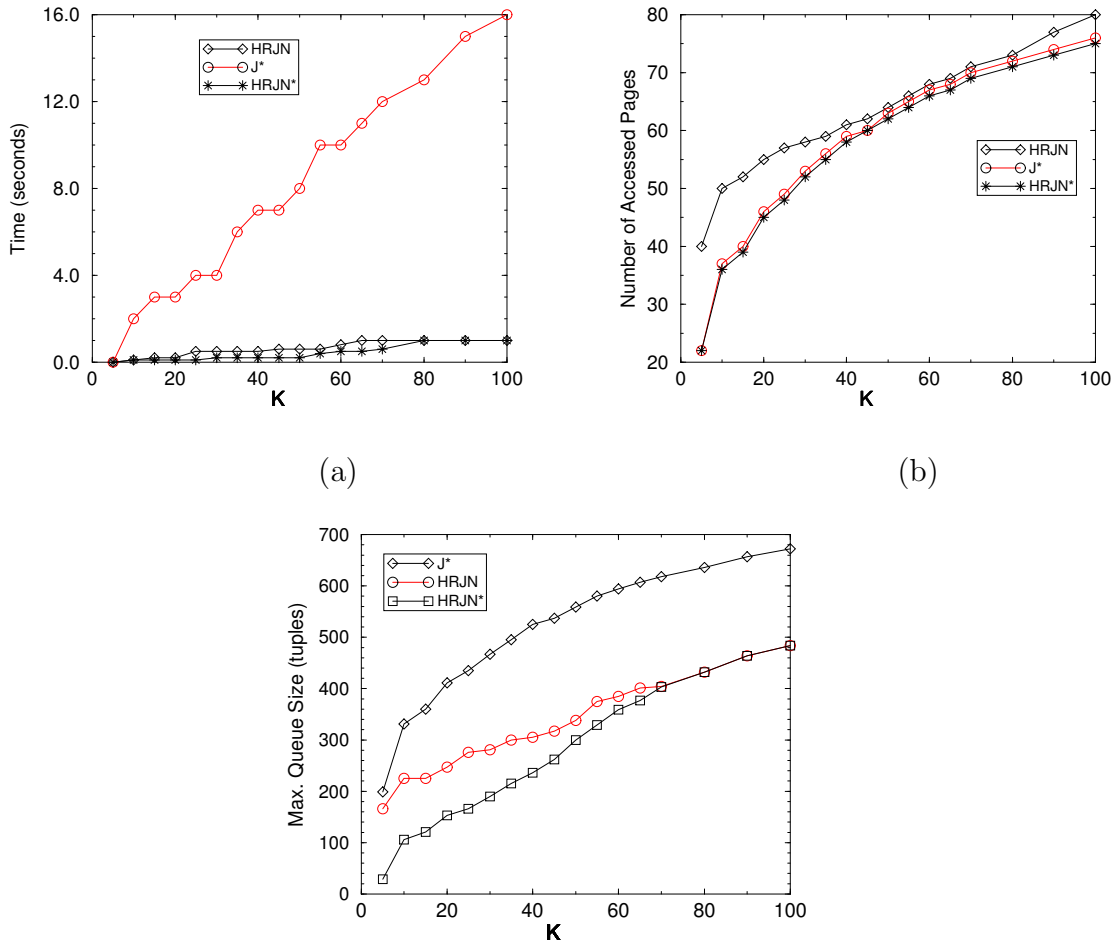


Figure 4.10. Comparing HRJN, J^* and HRJN* for $m = 4$ and Selectivity $= 0.2\%$

in each step. In each step, J^* tries to determine the next optimal point to visit in the Cartesian space. Since both HRJN and HRJN* use symmetric hash join to produce valid join combinations, more join combinations are ranked at each step. Figure 4.10 (b) compares the number of accessed disk pages. The three algorithms have a comparable performance in terms of the number of pages retrieved. J^* and HRJN* achieve better performance because retrieving a new tuple is guided by the score of the inputs, which makes both algorithms retrieve only the tuples that

makes significant decrease in the threshold value and hence less I/O. Figure 4.10 (c) compares the number of maintained buffer space. HRJN and HRJN* have low space overhead because they use the buffer only for ranking the join combinations, while J^* maintains all the retrieved tuples in its buffer. Had we also included the space of the hash tables, J^* will have a lower overall space requirement. In most practical systems the hash space is already reserved for hash join operations. Hence, the space overhead is only the buffer needed for ranking.

Changing the join selectivity: In this experiment, we fix the value of k to 50 and vary the join selectivity from 0.12% to 2%. Figure 4.11 (a) compares the total time to report 50 ranked results, while Figures 4.11 (b) and 4.11 (c) compare the number of accessed disk pages and the extra space overhead, respectively. For all selectivity values, HRJN* shows the best performance. J^* has a better performance than HRJN for high selectivity values while HRJN performs better for low selectivity values. The reason is that HRJN* combines the advantages of J^* and HRJN. While HRJN* uses a score-guided strategy to navigate in the Cartesian space for a faster termination (similar to J^*), it also uses the power of producing fast join results by using the symmetric hash join technique (similar to HRJN).

The effect of pipelining: In this experiment, we evaluate the scalability of the rank-join operators. We vary the number of join inputs, m, from 3 to 6 and fix $k = 50$ and the join selectivity to 0.2%. Figure 4.12 (a) gives the effect of pipelining on the total query time. HRJN and HRJN* show much better scalability than that of J^* by orders of magnitude. The CPU complexity of J^* increases significantly as m increases. On the other hand, J^* and HRJN* show better performance in terms of the number of accessed pages compare to HRJN (Figure 4.12 (b)), because of the score-guided strategy they are using. HRJN* is the most scalable in terms of the space overhead as shown in Figure 4.12 (c).

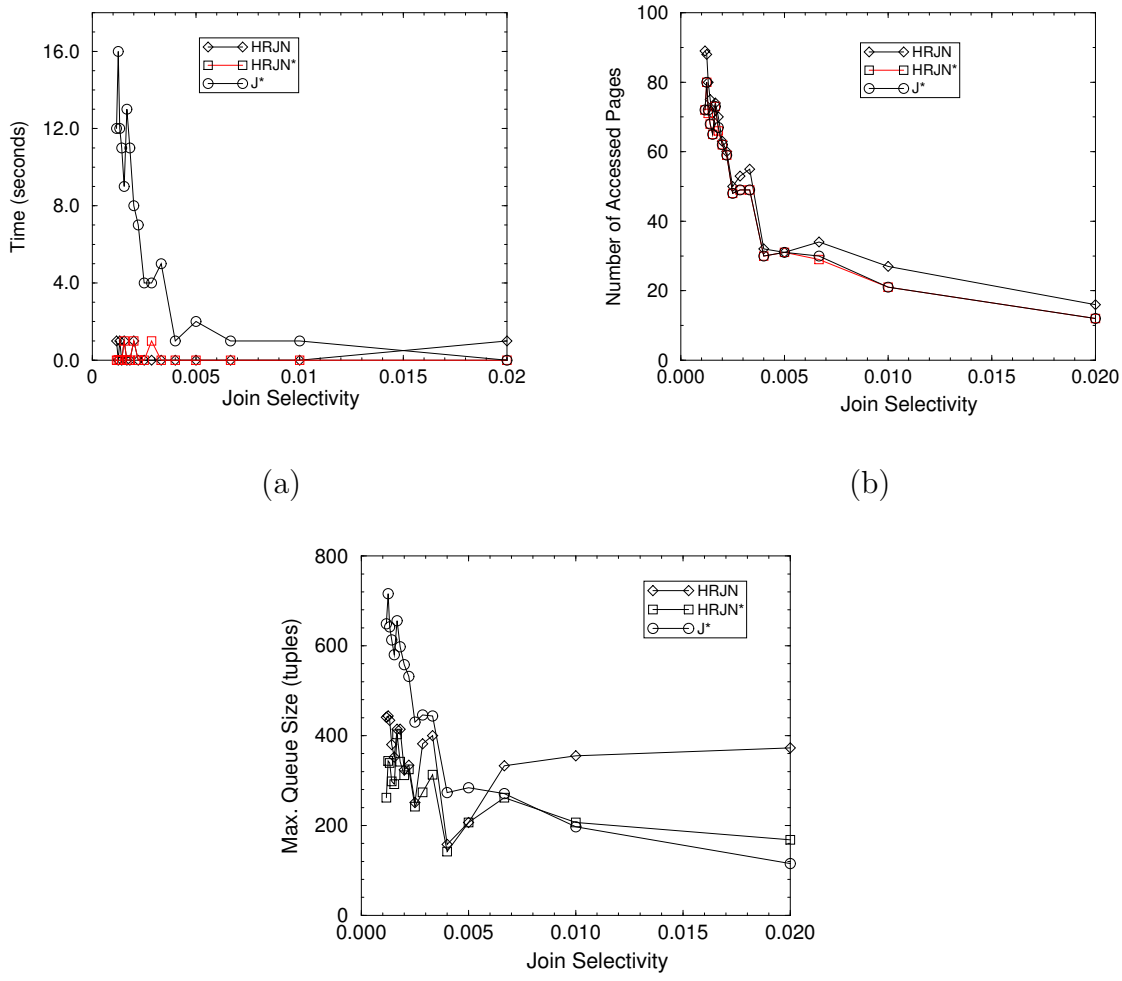


Figure 4.11. The Effect of Selectivity on HRJN, J^* and HRJN* for $m = 4$ and $K = 50$

4.7 Summary

In this chapter, we addressed supporting top-k join queries in practical relational query processors. We introduced a new rank-join algorithm that is independent of the join strategy, along with its correctness proof. The proposed rank-join algorithm makes use of the ranking on the input relations to produce ranked join results on a

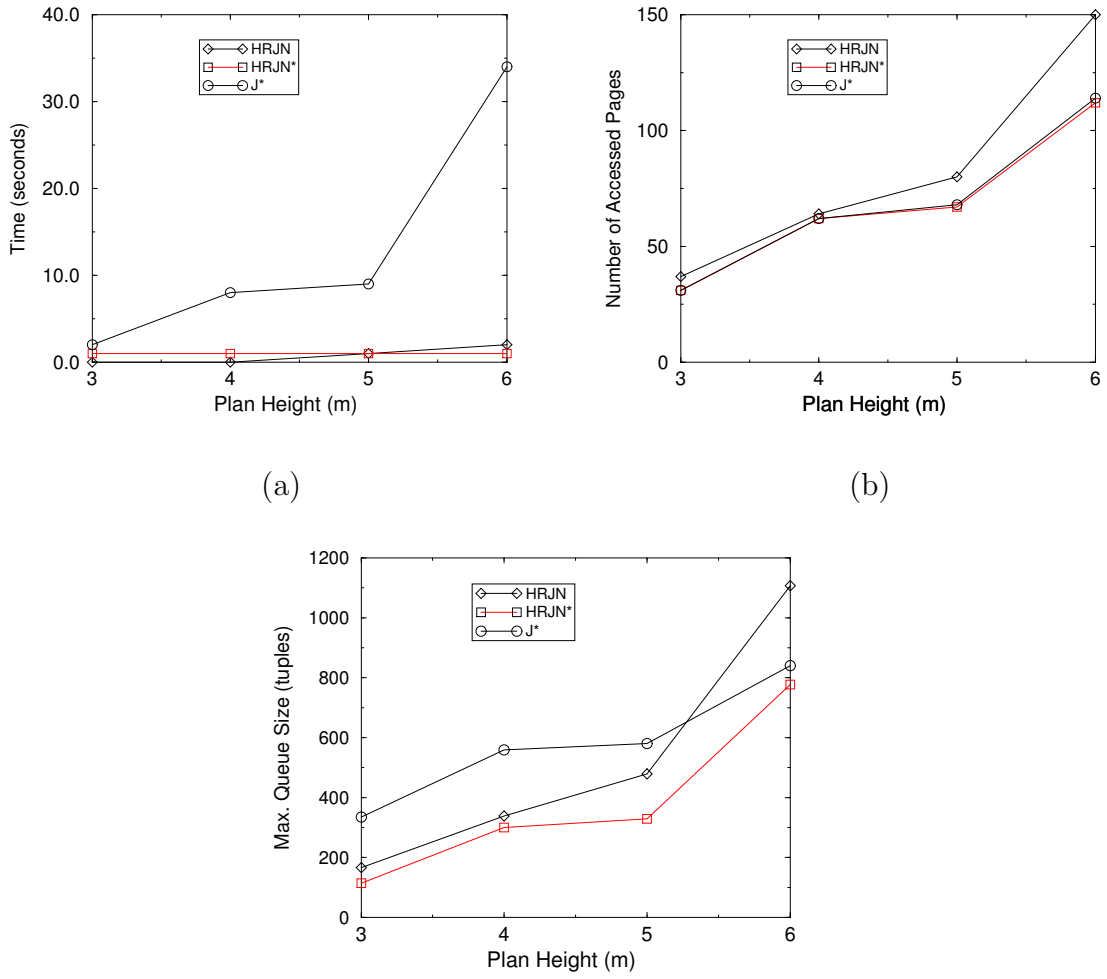


Figure 4.12. The Effect of Pipelining on HRJN, J^* and HRJN* for Selectivity 0.2% and $K = 50$

combined score. The ranking is performed progressively during the join and hence, there is no need for a blocking sort operation after join. We analyzed the I/O performance of the proposed rank-join algorithm and proved its optimality in terms of the number of accessed input tuples. We presented a physical query operator to implement rank-join based on ripple join; the hash rank join (HRJN). We proposed a new join strategy that is guided by the input score values. We applied the new

strategy on the original HRJN algorithm and called the new operator HRJN*. We studied the effect of rank-join order on the performance of rank-join query evaluation pipeline. We introduced an efficient rank-join order heuristic to help choose a near-optimal join order. We addressed exploiting available indexes on the join columns. We proposed a general rank-join algorithm that utilizes these indexes for faster termination of the ranking process. We experimentally evaluated the proposed join operators and compared their performance with a recent algorithm to join ranked inputs. We conducted several experiments varying the number of required answers, the join selectivity, and the number of inputs in the pipeline. The experiments proved the concept and showed a significant performance enhancement, especially for low values of join selectivity.

5 RANK-AWARE QUERY OPTIMIZATION

In this chapter, we introduce a rank-aware query optimization framework that fully integrates rank-join operators into relational query engines. The framework is based on extending the System R dynamic programming algorithm in both enumeration and pruning. We define ranking as an interesting property that triggers the generation of rank-aware query plans. Unlike traditional join operators, optimizing for rank-join operators depends on estimating the input cardinality of these operators. We introduce a probabilistic model for estimating the input cardinality, and hence the cost of a rank-join operator. Costing ranking plans, although challenging, is key to the full integration of rank-join operators in real-world query processing engines. The experiments show the validity of our framework and the accuracy of the proposed estimation model.

The rest of this chapter is organized as follows. Section 5.1 motivates the need for integrating ranking in relational query optimization and highlights the main challenges. We show how to extend traditional query optimization to be rank-aware in Section 5.2. Moreover, in Section 5.2, we show how to treat ranking as an interesting physical property and its impact on plan enumeration. In Section 5.3, we introduce a novel probabilistic model for estimating the input size (depth) of rank-join operators and hence estimating the cost and space complexity of these operators. In Section 5.4, we experimentally verify the proposed estimation model and show the accuracy of estimating the input size and the maximum buffer size needed by rank-join operators. We discuss related work in Section 5.5 and conclude in Section 5.6 by a summary and final remarks.

5.1 The Need for Rank-aware Query Optimization

For the new rank operators, described earlier in Chapters 3 and 4, to be practically useful they must be integrated in real-world query optimizers. Top-k queries often involve other query operations such as join, selection and grouping. A key challenge is how to choose a query execution plan that uses the new rank-join operators most efficiently.

An observation that motivates the need for integrating rank-join operators in query optimizers, is that a rank-join operator may not always be the best way to produce the required ranked results. In fact, depending on many parameters (for example, the join selectivity, the available access paths and the memory size) a traditional join-then-sort plan may be a better way to produce the ranked results.

Figure 5.1 gives the estimated I/O cost of two plans: a *sort plan* and a *rank-join plan*, for various values of the join selectivity. The sort plan is a traditional plan that joins two inputs and sorts the results on the given scoring function, while the rank-join plan uses a rank-join operator that progressively produces the join results ranked on the scoring function. The figure shows that for low values of the join selectivity, the traditional sort-plan is cheaper than the rank-join plan. On the other hand, for higher selectivity values, the rank-join plan is cheaper.

The previous example highlights the need to optimize top-k queries by integrating rank-join operators in query optimization. This approach, although appealing and intuitive, is hindered by the following challenges:

- How to generate plans that make use of rank-join operators ? What will be the plan property that triggers the generation of such plans ?
- How to estimate the cost of a rank-join query operator ? What will be the value of k when pushed all the way down in the query pipeline ? What is the effect of other operators in the plan on the cost estimation ?

Another way to phrase the first set of questions is how to make the query optimizer "aware" of the newly available ranking operators and their unique properties.

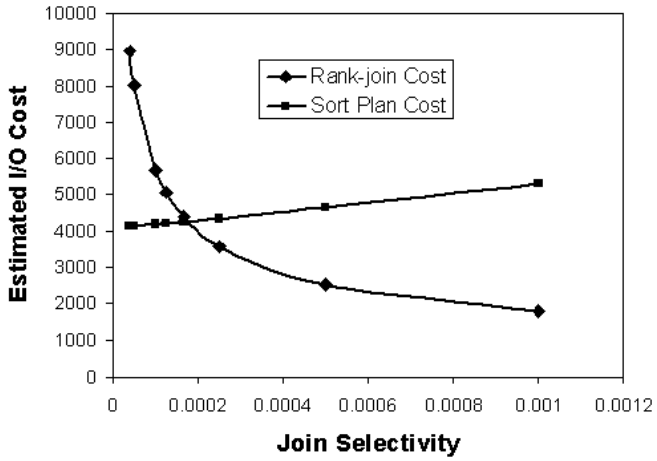


Figure 5.1. Estimated I/O Cost for Two Ranking Plans

Throwing these operators as yet another join implementation would not work without defining new physical properties that guarantee the best use of these operators.

Unlike traditional query operators, it is hard to estimate the cost of rank-join operators because of their "early out" feature; whenever the top k results are reported, the execution stops without consuming all the inputs. The "early out" feature poses many challenges in costing rank-join operators.

In this chapter, we show how to generate the rank-join plan as an alternative execution plan to answer top-k queries. We also show how we came up with the cost estimation of the rank-join plan used in Figure 5.1, for effective query optimization.

5.2 Rank-aware Optimization

In this section, we describe how to extend the traditional query optimization–one that uses dynamic programming a la [39]–to handle the new rank-join operators. Integrating the new rank-join operators in the query optimizer includes two major tasks:(1) enlarging the space of possible plans to include those plans that use rank-join operators as a possible join alternative, and (2) providing a costing mechanism

for the new operators to help the optimizer prune expensive plans in favor of more general cheaper plans.

In this section, we elaborate on the first task while in the following section we provide an efficient costing mechanism for rank-join operators. Enlarging the plan space is achieved by extending the enumeration algorithm to produce new execution plans. The extension must conform to the enumeration mechanism of other traditional plans. In this work, we choose the *DP* enumeration technique, described in Section 2.5. The *DP* enumeration is one of the most important and widely used enumeration techniques in commercial database systems. Current systems use different flavors of the original *DP* algorithm that involve heuristics to limit the enumeration space and can vary in the way the algorithm is applied (e.g., bottom-up versus top-down). We stick to the bottom-up *DP* as originally described in [39]. Our approach is equally applicable to other enumeration algorithms.

5.2.1 Ranking as an Interesting Property

As described in Section 2.5, interesting orders are those orders that can be beneficial to later operations. Practically, interesting orders are collected from: (1) columns in equality predicates in the join condition, as orders on these columns make upcoming sort-merge operations much cheaper by avoiding the sort, (2) columns in the *groupby* clause to avoid sorting in implementing sort-based grouping, and (3) columns in the *orderby* clause since they must be enforced on the final answers. Current optimizers usually enforce interesting orders in an *eager* fashion. In the eager policy, the optimizer generates plans that produce the interesting order even if they do not exist naturally (e.g., through the existence of an index).

In the following example, we describe a top-k query using current `SQL` constructs by specifying the ranking function in the *orderby* clause.

Q2:

```
WITH RankedABC as (
  SELECT A.c1 as x ,B.c1 as y, C.c1 as z, rank() OVER
  (ORDER BY (0.3*A.c1+0.3*B.c1+0.3*C.c1)) as rank
  FROM A,B,C
  WHERE A.c2 = B.c1 and B.c2 = C.c2)
SELECT x,y,z,rank
FROM RankedABC
WHERE rank <=5;
```

where `A, B` and `C` are three relations and `A.c1, A.c2, B.c1, B.c2, C.c1` and `C.c2` are attributes of these relations. Following the concept of *interesting orders*, the optimizer considers orders on `A.c2, B.c1, B.c2` and `C.c2` as interesting orders (because of the join) and *eagerly* enforces the existence of plans that access `A, B` and `C` ordered on `A.c2, B.c1, B.c2` and `C.c2`, respectively. This enforcement can be done by *gluing* a sort operator on top of the table scan or by using an available index that produces the required order. Currently, orders on `A.c1` or `C.c1` are "not interesting" since they are not beneficial to other operations such as a sort-merge join or a sort. The reason being that a sort on the expression `(0.3*A.c1+0.3*B.c1+0.3*C.c1)` cannot benefit from ordering the input on `A.c1` or `C.c2` individually.

Having the new rank-aware physical join operators, orderings on the individual scores (for each input relation) become *interesting* in themselves. In the previous example, an ordering on `A.c1` is interesting because it can serve as input to a rank-join operator. Hence, we extend the notion of interesting orders to include those attributes that appear in the ranking function.

Definition 5.2.1 *An Interesting Order Expression is ordering the intermediate results on an* expression *of database columns that can be beneficial to later query operations.*

In the previous example, we can identify some interesting order expressions according to the previous definition. We summarize these orders in Table 5.1. Like an

Table 5.1
Interesting Order Expressions in Query **Q2**

Interesting Order Expressions	Reason
`A.c1`	Rank-join
`A.c2`	Join
`B.c1`	Join and Rank-join
`B.c2`	Join
`C.c1`	Rank-join
`C.c2`	Join
`0.3*A.c1+0.3*B.c1`	Rank-join
`0.3*B.c2+0.3*C.c2`	Rank-join
`0.3*A.c1+0.3*C.c2`	Rank-join
`0.3*A.c1+0.3*B.c2+0.3*C.c2`	Orderby

ordinary interesting order, an interesting order expression *retires* when it is used by some operation and is no longer useful for later operations. In the previous example, an order on `A.c1` is no longer useful after a rank-join between table `A` and `B`.

5.2.2 Extending the Enumeration Space

In this section, we show how to extend the enumeration space to generate rank-aware query execution plans. Rank-aware plans will integrate the rank-join operators, described in Chapters 3 and 4, into general execution plans. The idea is to devise a set of rules that generate rank-aware join choices at each step of the *DP* enumeration algorithm. For example, on the table access level, since interesting orders now contain ranking score attributes, the optimizer will enforce the generation of table and index access paths that satisfy these orders. In enumerating plans at higher levels (join plans), these ordered access paths will make it feasible to use rank-join operators as join *choices*.

For a query with n input relations, T_1 to T_n, assume there exists a ranking function $f(s_1, s_2, \ldots, s_n)$, where s_i is score expression on relation T_i. For two sets of input relations, L and R, we extend the space of plans that join L and R to include rank-join plans by adapting the following:

- *Join Eligibility* L and R are rank-join-eligible if all the following apply:

  1. There is a join condition that relates at least one input relation in L to an input relation in R.
  2. f can be expressed as $f(f_1(S_L), f_2(S_R), f_3(S_O))$, where f_1, f_2 and f_3 are three scoring functions, S_L are the score expressions on the relations in L, S_R are the score expressions on the relations in R, and S_O are the score expressions on the rest of the input relations.
  3. There is at least one plan that accesses L and/or R ordered on S_L and/or S_R, respectively.

- *Join Choices* As described in previous chapters, rank-join can have several implementations as physical join operators. For each rank-join between L and R, plans can be generated for each join implementation. For example, an HRJN plan is generated if there exist plans that access both L and R sorted on S_L and S_R, respectively.
- *Join Order* For symmetric rank-join operators (e.g., HRJN), there is no distinction between outer and inner relations. For the nested-loops implementation, a different plan can be generated by switching the inner and the outer relations. L (R) can serve as inner to an nested-loops operator if there exists a plan that accesses L (R) sorted on S_L (S_R).

For example, for Query **Q2** in Section 5.2.1, new plans are generated by enforcing the interesting order expressions listed in Table 5.1 and using all join choices available including the rank-join operators. As in traditional *DP* enumeration, generated plans are pruned according to their cost and properties. For each class of properties, the cheapest plan is kept. Figure 5.2 gives the MEMO structure of the retained

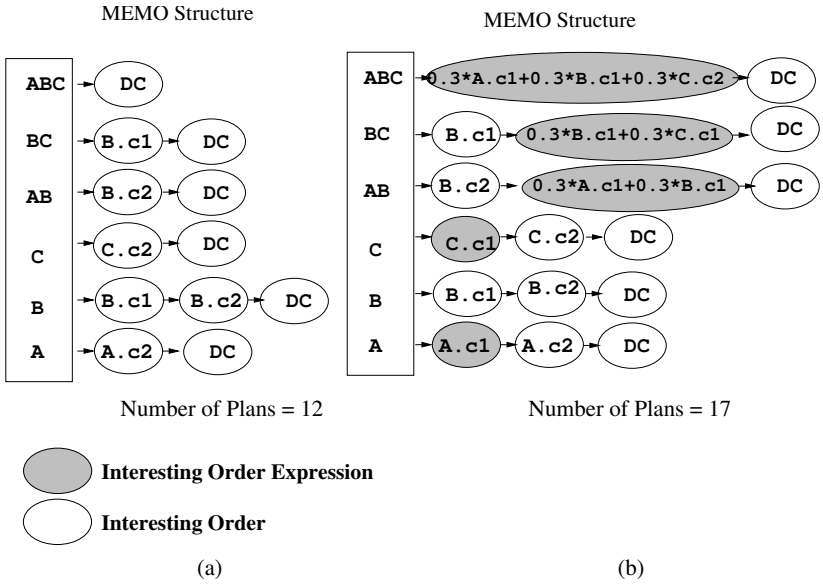


Figure 5.2. Enumerating Rank-aware Query Plans

subplans when optimizing **Q2**. Each oval in the figure represents the best plan with a specific order property. Figure 5.2 (a) gives the MEMO structure for the traditional application of the *DP* enumeration without the proposed extension. For example, we keep two plans for Table `A`; the cheapest plan that does not have any order property (`DC`) and the cheapest plan that produces results ordered on `A.c2` as an interesting order. Figure 5.2 (b) shows the newly generated classes of plans that preserve the required ranking. For each interesting order expression, the cheapest plan that produces that order is retained. For example, in generating plans that join Tables `A` and `B`, we keep the cheapest plan that produces results ordered on `0.3*A.c1 + 0.3*B.c1`.

5.2.3 Pruning Plans

A subplan P_1 is pruned in favor of subplan P_2 if and only if P_1 has both higher cost and weaker properties than P_2. In Section 5.2.2, we discussed extending the interesting order property to generate rank-aware plans. A key property of top-k queries is that users are interested only in the first k results and not in a total

ranking of all query results. This property directly impacts the optimization of top-k queries by optimizing for the first k results. Traditionally, most real-world database systems offer the feature of *First-N-Rows-Optimization*. Users can turn on this feature when desiring fast response time to receive results as soon as they are generated. This feature translates into respecting the "pipelining" of a plan as a physical plan property. For example, for two plans P_1 and P_2 with the same physical properties, if P_1 is a pipelined plan (e.g., nested-loops join plan) and P_2 is a non-pipelined plan (e.g., sort-merge join plan), P_1 cannot be pruned in favor of P_2, even if P_2 is cheaper than P_1.

In real-world query optimizers, the cost model for different query operators is quite complex and depends on many parameters. Parameters include cardinality of the inputs, available buffers, type of access paths (e.g., a clustered index) and many other system parameters. Although cost models can be very complex, a key ingredient of accurate estimation is the accuracy of estimating the size of intermediate results.

In traditional join operators, the input cardinalities are independent of the operator itself and only depend on the input subplan. Moreover, the output cardinality depends only on the size of the inputs and the selectivity of the logical operation. On the other hand, since a rank-join operator does not consume all of its inputs, the actual input size depends on the operator itself and how the operator decides that it has seen "enough" information from the inputs to generate the top k results. Hence, the input cardinality depends on the number of ranked join results requested from that operator. Thus, the cost of a rank-join operator depends on the following:

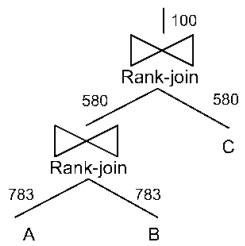


Figure 5.3. Example Rank-join Plan

- *The number of required results k and how k is propagated in the pipeline.* For example, Figure 5.3 gives a real similarity query that uses two rank-join operators to combine the ranking based on three features, referred to as A , B and C. To get 100 requested results (i.e., $k = 100$), the top operator has to retrieve 580 tuples from each of its inputs. Thus, the number of required results from the child operator is 580 in which it has to retrieve 783 tuples from its inputs. Notice that while $k = 100$ in the top rank-join operator, $k = 580$ in the child rank-join operator that joins A and B. In other words, in a pipeline of rank-join operators, the input depth of a rank-join operator is the required number of ranked results from the child rank-join operator.
- *The number of tuples from inputs that contain enough information for the operator to report the required number of answers, k.* In the previous example, the top operator needs 580 tuples from both inputs to report 100 rankings, while the child operator needed 783 tuples from both inputs to report the required 580 partial rankings.
- *The selectivity of the join operation.* The selectivity of the join affects the number of tuples propagated from the inputs to higher operators through the join operation. Hence, the join selectivity affects the number of input tuples required by the rank-join operator to produce ranked results.

There are two ways to produce plans that join two sets of input relations, L and R, and produce ranked results: (1) by using rank-join operators to join L and R subplans, or (2) by gluing a sort operator on the cheapest join plan that joins L and R without preserving the required order. One challenge is in comparing two plans when one or both of them are rank-join plans. For example, in the two plans depicted in Figure 5.4, both plans produce the same order property. Plan (b) may or may not be pipelined depending on the subplans of L and R. In all cases, the cost of the two plans need to compared to decide on pruning. While the current traditional cost model can give an estimate total cost of Plan (a), it is hard to estimate the cost of Plan (b) because of its strong dependency on the number of required ranked results,

k. Thus, to estimate the cost of Plan (b), we need to estimate the propagation of the value of k in the pipeline (refer to Figure 5.3). In Section 5.3, we give a probabilistic model to estimate the depths (d_L and d_R in Figure 5.4 (b)) required by a rank-join operator to generate top k ranked results. The estimate for the depths is parameterized by k and by the selectivity of the join operation. It is important to note that the cost of Plan (a) is (almost) independent of the number of output tuples pulled from the plan since it is a blocking sort plan. In Plan (b), the number of required output tuples determines how many tuples will be retrieved from the inputs and that greatly affects the plan cost.

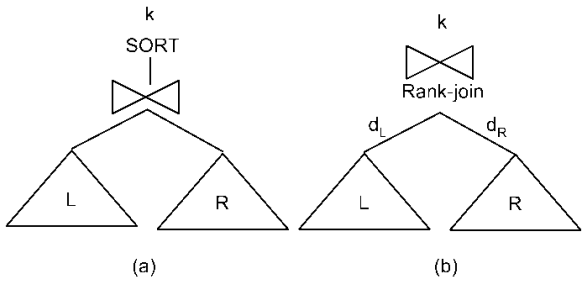


Figure 5.4. Two Enumerated Plans

Plan Pruning According to our enumeration mechanism, at any level, there will be only one plan similar to Plan (a) of Figure 5.4 (by gluing a sort on the cheapest non-ranking plan). At the same time, they may be many plans similar to Plan (b) of Figure 5.4 (e.g., by changing the type of the rank-join operator or the join order).

For all rank-join plans, the cost of the plan depends on k and the join selectivity s. Since these two parameters are the same for all plans, the pruning among these plans follows the same mechanism as in traditional cost based pruning. For example, pruning a rank-join plan in favor of another rank-join plan depends on the input cardinality of the relations, the cost of the join method, the access paths, and the statistics available on the input scores.

We assume the availability of an estimate of the join selectivity, which is the same for both sort-plans and rank-join plans. A challenging question is how to compare

between the cost of a rank-join plan and the cost of a sort plan, e.g., Plans (a) and (b) in Figure 5.4, when the number of required ranked results is unknown. Note that the number of results, k, is known only for the final complete plan. Because subplans are built in a bottom-up fashion, the propagation of the final k value to a specific subplan depends on the location of that subplan in the complete evaluation plan.

We introduce a mechanism for comparing the two plans in Figure 5.4 using the estimated total cost of Plan (a) and the estimated cost of Plan (b), parametrized by k. Section 5.3 describes how to obtain the parametrized cost of Plan (b). For Plan (a), we can safely assume that $Cost_a(k) = TotalCost_a$ where $Cost_a(k)$ is the cost to report k results from Plan (a), and $TotalCost_a$ is the cost to report all join results of Plan (a). This assumption follows directly from Plan (a) being a blocking sort plan. Let k^* be that value of k at which the cost of the two plans are equal. Hence, $Cost_a(k^*) = Cost_b(k^*) = TotalCost_a$. The output cardinality of Plan (a) (call it n_a) can be estimated as the product of the cardinalities of all inputs multiplied by the estimated join selectivity. Since k cannot be more than n_a, we compare k^* with n_a. Let k_{min} be the minimum value of k for any rank-join subplan. A reasonable value for k_{min} would be the value specified in the query as the total number of required answers. Consider the following cases:

- $k^* > n_a$: Plan (b) is always cheaper than Plan (a). Hence Plan (a) should be pruned in favor of Plan (b).
- $k^* < n_a$ and $k^* < k_{min}$: Since for any subplan, $k \geq k_{min}$, we know that we will require more that $k*$ output results from Plan (b). In that case Plan (a) is cheaper. Depending on the nature of Plan (b) we decide on pruning:
  - If Plan (b) is a pipelined plan (e.g., a left deep tree of rank-join operators), then we cannot prune Plan (b) in favor of Plan (a) since it has more properties, the pipelining property.
  - If Plan (b) is not a pipelined tree, then Plan (b) is pruned in favor of Plan (a).

- $k^* < n_a$ and $k^* > k$: We keep both plans since depending on k, Plan (a) may be cheaper than Plan (b) and hence cannot be pruned.

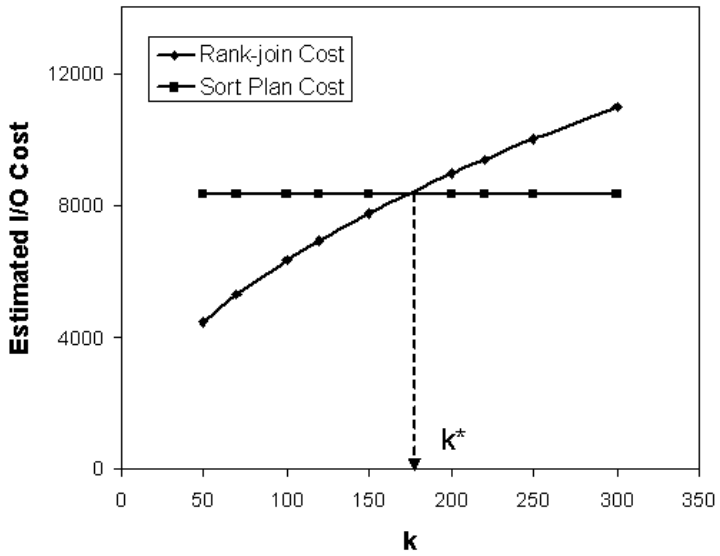


Figure 5.5. The Effect of k on the Rank-join Cost

As an example, we show how the value of k affects the cost of rank-join plans and hence the plan pruning decisions. We compare two plans that produce ranked join results of two inputs. The first plan is a *sort plan* similar to that in Figure 5.4(a), while the second plan is a *rank-join plan* similar to that in Figure 5.4(b). The sort plan sorts the join results of an index nested-loops join operator while the rank-join plan uses HRJN as its rank-join operator. The estimated cost formula for the sort plan uses the traditional cost formulas for external sorting and index nested-loops join, while the estimated cost of the rank-join plan is based on our model to estimate the input cardinality (as will be shown in Section 5.3). Both cost estimates use the same values of input relations cardinalities, total memory size, buffer size, and input tuple sizes. Figure 5.5 compares the estimate of the costs of the two plans for different values of k. While the sort plan cost can be estimated to be independent of k, the cost of the rank-join plan increases with increasing the value of k. In this example, $k^* = 176$.

5.3 Estimating Input Cardinality of Rank-join Operators

In this section, we give a probabilistic model to estimate the input cardinality (depth) of rank-join operators. The estimate is parameterized with k, the number of required answers from the (sub)plan, and s, the selectivity of the join operation. We describe the main idea of the estimation procedure by first considering the simple case of two ranked relations. Then, we generalize to the case of a hierarchy of rank-join operators.

Let L and R be two ranked inputs to a rank-join operator. Let m and n be the table cardinalities of L and R, respectively. Our objective is to get an estimate of depths d_L and d_R (see Figure 5.6) such that it is sufficient to retrieve only up to d_L and d_R tuples from L and R, respectively, to produce the top k join results. We denote the top i tuples of L and R as $L(i)$ and $R(i)$, respectively. We outline our approach to estimate d_L and d_R in Table 5.2.

In the following subsections, we elaborate on steps of the outline in Table 5.2. Table 5.3 gives Algorithm *Propagate* used by the query optimizer to compute the values of d_L and d_R at all levels in a rank-join plan. We set k to the value specified in the query when we call the algorithm for the final plan.

We assume the following to simplify the analysis: (1) the combining scoring function is a linear combination of the scores (e.g., a weighted sum of the input scores), and (2) each tuple in L is equally likely to join with sn tuples in R and each tuple in R is equally likely to join with sm tuples in L.

5.3.1 Estimating Any-k Depths

In the first step of the outline in Table 5.2, we estimate the depths c_L and c_R in L and R, respectively, required to get any k join results. "Any k" join results are valid join results, but not necessarily among the top k answers in score.

Theorem 5.3.1 *If c_L and c_R are chosen such that $sc_Lc_R \geq k$, then the expected number of valid join results between $L(c_L)$ and $R(c_R)$ is $\geq k$.*

Table 5.2
Outline of the Estimation Technique

Outline EstimateTop-kDepth

INPUT: Two ranked relations L and R

The number of required ranked results, k

The join selectivity, s

Any-k Depths

1. Compute the value of c_L and c_R, where
   c_L is the depth in L and c_R is the depth in R such that,
   $\exists$ expected k valid join results between $L(c_L)$ and $R(c_R)$

Top-k Depths

2. Compute the value of d_L and d_R, where
   d_L is the depth in L and d_R is the depth in R such that,
   $\exists$ expected k top-scored join results between $L(d_L)$.
   and $R(d_R)$. d_L and d_R are expressed in terms of c_L and c_R.

Minimize Top-k Depths

3. Compute the values of c_L and c_R to minimize d_L and d_R.
   c_L, c_R, d_L and d_R are paramterized by k

Proof Let $X_{i,j}$ denote a random variable that is equal to the number of join results produced by joining the first i tuples from L and the first j tuples from R. Since every tuple in L is likely to join with sj tuples in $R(j)$, then the expected value of this random variable is $E[X_{i,j}] = sij$. Let $c_L = i$ and $c_R = j$, hence, if $sc_Lc_R \geq k$, then we can expect at least k valid join results between $L(c_L)$ (the top c_L tuples in L) and $R(c_R)$. ■

Table 5.3
Propagating the Value of k

Algorithm Propagate (subplan P, k)

INPUT: The number of required ranked results, k

The root of a subplan, P

OUTPUT: d_L and d_R for the operator rooted at P

1. Compute d_L and d_R according to the formulas in Section 5.3.3
2. Call Propagate(left subplan of P, d_L)
3. Call Propagate(right subplan of P, d_R)

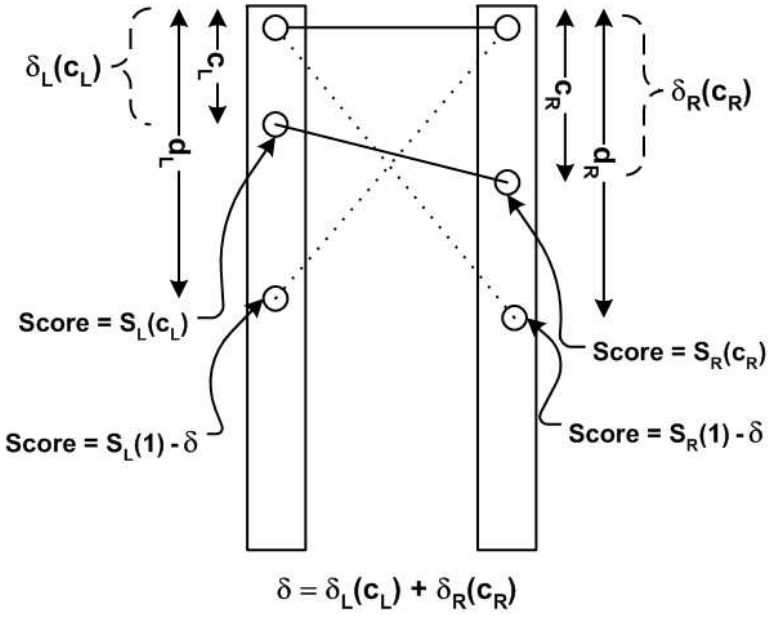


Figure 5.6. Depth Estimation of Rank-join Operators

In general, the choice of c_L and c_R can be arbitrary as long as they satisfy $sc_Lc_R \geq k$. We show that we choose values for c_L and c_R in Section 5.3.3.

5.3.2 Estimating Top-k Depths

In the second step in the outline given in Table 5.2, we aim at obtaining good estimates for d_L and d_R, where d_L and d_R are the depths into L and R, respectively, needed to produce an expected number of top k join results. For the simplicity of presentation, the formulas presented in this section assume that the scoring function is the summation of individual scores.

Let $S_L(i)$ and $S_R(i)$ be the scores of the tuples at depth i in L and R, respectively. Moreover, let $\delta_L(i)$ and $\delta_R(i)$ be the score difference between the top ranked tuple and the score of the tuple at a depth i in L and R, respectively, i.e., $\delta_L(i) = S_L(1) - S_L(i)$ and $\delta_R(i) = S_R(1) - S_R(i)$

Theorem 5.3.2 *If there are k valid join results between $L(c_L)$ and $R(c_R)$, and if d_L and d_R are chosen such that $\delta_L(d_L) \geq \delta_L(c_L) + \delta_R(c_R)$ and $\delta_R(d_R) \geq \delta_L(c_L) + \delta_R(c_R)$, then the top k join results can be obtained by joining $L(d_L)$ and $R(d_R)$.*

Proof Refer to Figure 5.6 for illustration. Let $\delta = \delta_L(c_L) + \delta_R(c_R)$ and $S = S_L(1) + S_R(1)$. Since, there are k join tuples between $L(c_L)$ and $R(c_R)$, the final score of each of the join results is $\geq S - \delta$. Consequently, the scores of all of the top k join results are $\geq S - \delta$. Assume that one of the top-k join results, J, joins a tuple t at depth d in L with some tuple in R such that $\delta_L(d) > \delta$. The highest possible score of J is $S_L(d) + S_R(1) = S - \delta_L(d) < S - \delta$. By contradiction, Tuple t cannot participate in any of the top k join results. Hence, any tuple in L (similarly R) that is at a depth $> d_L$ (d_R) cannot participate in the top k join results. ■

Since the choice of c_L and c_R can be arbitrary as long as they satisfy the condition in Theorem 5.3.1, Step (3) of the outline in Table 5.2 chooses the values of c_L and c_R that minimize the values of d_L and d_R. Note that both d_L and d_R are minimized when $\delta = \delta_L(c_L) + \delta_R(c_R)$ is minimized. Hence we minimize δ subject to the constraint $sc_Lc_R \geq k$. The rationale behind this minimization is that an optimal rank-aggregation algorithm does not need to retrieve more than the minimum d_L and d_R tuples from L and R, respectively, to generate the top k join results.

5.3.3 Estimating the Minimum d_L and d_R

Till now, we did not have any assumptions on the score distributions of L and R. We showed that d_L and d_R are related to c_L and c_R in terms of the scores of the tuples at these depths.

To have a closed formula for the minimum d_L and d_R, we assume that the rank scores in L and R are from some uniform distribution. Let x be the average decrement slab of L (i.e., the average difference between the scores of two consecutive ranked objects in L) and let y be the average decrement slab for R. Hence, the expected value of $\delta_L(c_L) = xc_L$ and the expected value of $\delta_R(c_R) = yc_R$. To minimize $\delta = \delta_L(c_L) + \delta_R(c_R)$, we minimize $xc_L + yc_R$, subject to $sc_Lc_R \geq k$. The minimization is achieved by setting $c_L = \sqrt{(yk)/(xs)}$ and $c_R = \sqrt{(xk)/(ys)}$. In this case, $d_L = c_L + (y/x)c_R$ and $d_R = c_R + (x/y)c_L$

In the simplistic case, where both the relations come from the same uniform distribution, i.e., $x = y$, then $c_L = c_R = \sqrt{k/s}$ and $d_L = d_R = 2\sqrt{k/s}$.

In a hierarchy of joins, where the output of one rank-join operator serves as input to another operator, the score distributions of the second level join are no longer uniform. Assuming the scoring function is the sum of two scores, the scores of rank join with two uniform distributions follows a triangular distribution. As we go higher up in the join hierarchy, the distribution tends to be normal (bell-shaped curve) by central limit theorem (see Figure 5.7).

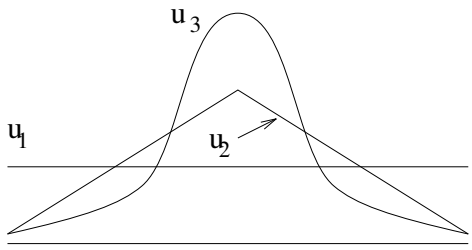


Figure 5.7. Central Limit Theorem

Let X, Y be two independent random variables from the uniform distribution $[0, n]$. We refer to this uniform distribution as u_1. We refer to the summation of j independent random variable from u_1 as u_j. The random variable $Z = X + Y$, which follows the distribution u_2, is a triangular distribution over $[0, 2n]$ with a peak at n. If we choose n elements from the u_2 distribution, the score of the ith element ($i \leq n/2$), in a decreasing order of the scores, is expected to be $2n - \sqrt{2in}$. In general, if we choose m elements from u_j, which ranges from $[0, jn]$, then the score of the ith element is expected to be

$$score_i = jn - (j!in^j/m)^{1/j} \tag{5.1}$$

Using the described distribution scores, we estimate the values of c_L and c_R that give the minimum values of d_L and d_R for the general rank-join plan in Figure 5.4 (b). Let the output of L be the output of rank-joining l ranked relations. Let the output of R be the output of rank-joining r ranked relations. Let k be the number of output ranked results required from the subplan, and s be the join selectivity. Then minimizing $\delta = \delta_L(c_L) + \delta_R(c_R)$ amounts to minimizing $\delta = (l!c_L n^{l-1})^{1/l} + (r!c_R n^{r-1})^{1/r}$. We substitute $c_R = \frac{k}{sc_L}$ and minimize δ with respect to c_L. The minimizations yield:

$$c_L^{r+l} = \frac{(r!)^l k^l n^{r-l} l^{rl}}{s^l (l!)^r r^{rl}} \tag{5.2}$$

$$c_R^{r+l} = \frac{(l!)^r k^r n^{l-r} r^{rl}}{s^r (r!)^l l^{rl}} \tag{5.3}$$

$$d_L = c_L[1 + r/l]^l \tag{5.4}$$

$$d_R = c_R[1 + l/r]^r \tag{5.5}$$

Note that d_L and d_R are strict upper-bounds assuming worst-case behavior. For an average case analysis, assume that L follows a u_l distribution and R follows a u_r distribution with each having n tuples. The join of L and R produces another relation, G with a u_{l+r} distribution and sn^2 tuples. Using Equation 5.1 and setting $j = l + r$ and $m = sn^2$, the score of the top kth tuple in G is $score_k = (l+r)n - ((l+r)!kn^{l+r-2}/s)^{1/(l+r)}$. Hence, we need to check in L (R) up to a tuple that joins with

R (L) to produce $score_k$. We can show that on average, d_L and d_R can be computed as follows:

$$d_L^{l+r} = \frac{((l+r)!)^l k^l n^{r-l}}{(l!)^{l+r} s^l} \quad and \quad d_R^{l+r} = \frac{((l+r)!)^r k^r n^{l-r}}{(r!)^{l+r} s^r}$$

Because the distribution of the depths is tight around the mean, we can apply the formulas recursively in a rank-join plan, as shown in the algorithm in Table 5.3, by replacing k of the left and right subplans by d_L and d_R, respectively. The value of k for the top operator is the value specified by the user in the query.

5.4 Experimental Verification of the Estimation Model

In this section, we experimentally verify the accuracy of our model for estimating the depths (input size) of rank-join operators and estimating an upper-bound of the buffer size maintained by these operators. Estimating the input size and the space requirements of a rank-join operator make it easy to estimate the total cost of a rank-join plan according to any practical cost model.

5.4.1 Implementation Issues and Setup

All experiments are based on a research platform for a complete video database management system running on a Sun Enterprise 450 with 4 UltraSparc-II processors running SunOS 5.6 operating system. The prototype is built on top of an open-source database management system that allows us to implement a simple cost-based rank-aware optimizer in the query engine (details are omitted for expository reasons). We have implemented a simple *DP* join enumerator that generates all possible rank-join plans in a bottom-up fashion.

In the experiments conducted in this section, the user query provides the system with an example image and requests the most similar video objects (segments or snapshots) to the query image based on multiple visual features. The visual features are extracted from the video data and are stored in separate relations. High-dimensional index access paths are available on these relations to rank the objects

according to each of the corresponding features. Example features include color histograms (*ColorHist*), color layout (*ColorLayout*), texture (*Texture*) and edge orientation (*Edges*). Hence, for a multi-feature similarity query, each input ranks the stored video objects according to a single feature. The top-k query produces the k objects with the top combined scores. We use the following top-k query:

Q: *Retrieve the k most similar video shots to a given image based on m visual features*

In the implemented prototype, we automatically show the generated evaluation (sub)plans at each level of the *DP* algorithm. We only display "templates" of the execution plans. Each of these plan templates generates several evaluation plans by changing the join implementation choices, switching the join order, or gluing sort operators to enforce interesting order properties.

Figure 5.9 gives a snapshot of the plan generation interface for joining 4 inputs. We focus on the first complete generated plan and annotate it in Figure 5.8 for easy referencing. We refer to this plan as Plan *P*.

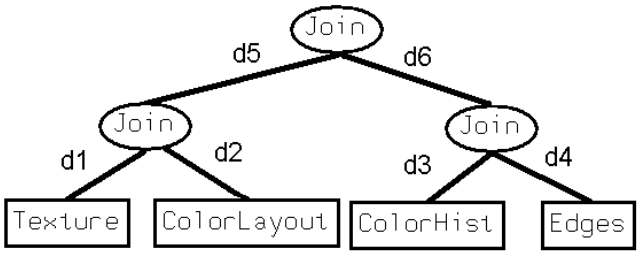


Figure 5.8. Example Rank-join Plan

5.4.2 Verifying Input Cardinality Estimation

In this experiment, we evaluate the accuracy of the depth estimates of rank-join operators. We conducted several experiments on a variety of example evaluation

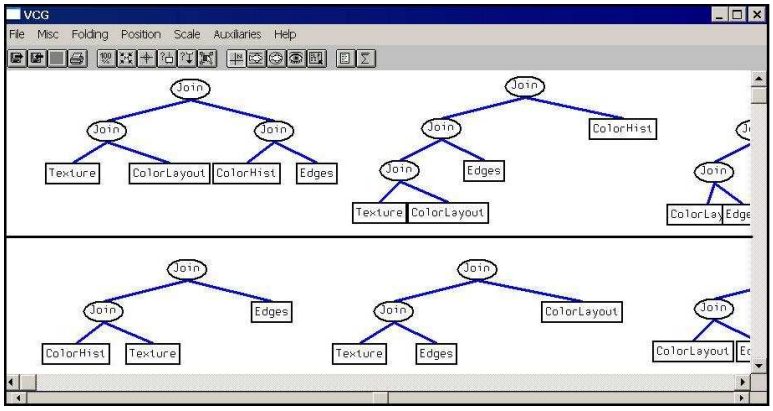
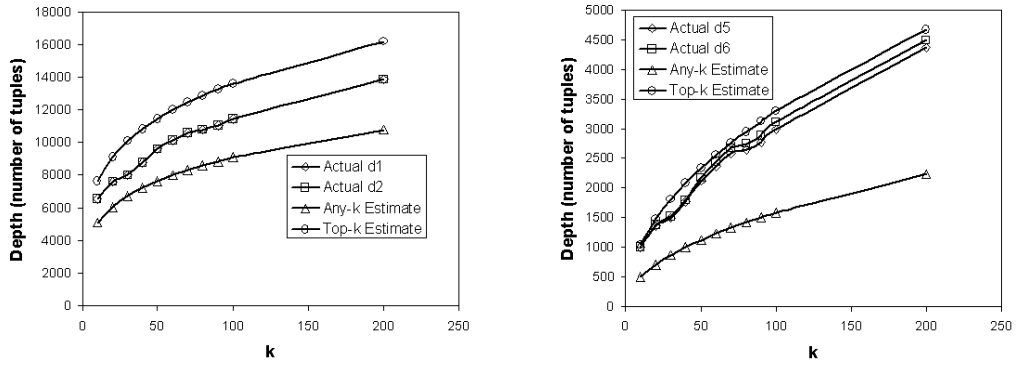


Figure 5.9. A Snapshot of the Plan Generation Interface

plans of Query **Q**. Since all experiments show similar behavior, we show a representative sample results for this experiment. The results shown here represent the estimates for Plan P in Figure 5.8. We use HRJN as the implementation of the rank-join operator. k ranked results are required from the top rank-join operator in the plan.

Varying the Number of Required Answers (k) For different values of k, Figure 5.10 (a) compares the actual values of d_1 and d_2 (refer to Figure 5.8) with two estimates:(1) *Any-k Estimate*, the estimated values for d_1 and d_2 to get any k join results (not necessary the top k), and (2) *Top-k Estimate*, the estimated values for d_1 and d_2 to get the top k join results. *Any-k Estimate* and *Top-k Estimate* are computed according to Section 5.3. The actual values of d_1 and d_2 are obtained by actually running the query and by counting the number of retrieved input tuples by each operator. Figure 5.10 (b) gives similar results for comparing the actual values of d_5 and d_6 to the same estimates. The figures show that the estimation error is less than 25% of the actual depth values. In general, for all conducted experiments, this estimation error is less than 30% of the actual depth values. Note that the measured values of d_1 and d_2 lie between the *Any-k Estimate* and the *Top-k Estimate*. The

Any-k Estimate can be considered as a *lower-bound* on the depths required by a rank-join operator.



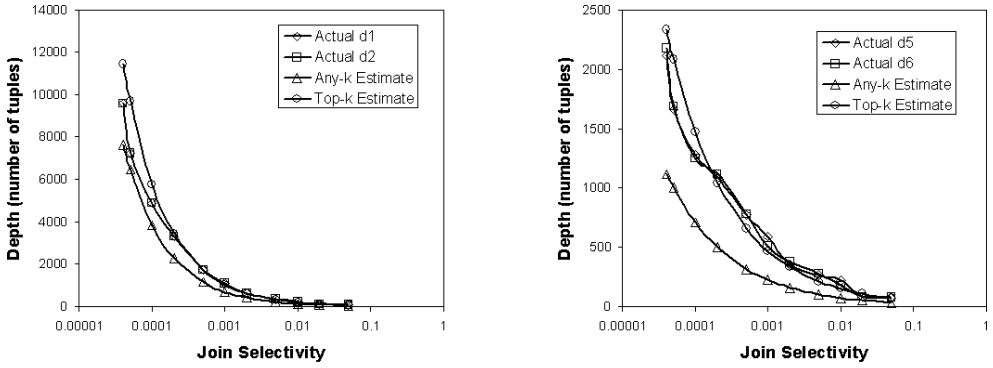
(a) (b)

Figure 5.10. Estimating the Input Cardinality for Different Values of k

Varying the Join Selectivity Figure 5.11 compares the actual and estimated values for the depths of Plan P in Figure 5.8 for various values of the join selectivity. For low selectivity values, the required depths increase as the rank aggregation algorithm needs to retrieve more tuples from each input to have enough information to produce the top ranked join results. The maximum estimation error is less than 30% of the actual depth values.

5.4.3 Estimating the Maximum Buffer Size

Rank-join operators usually maintain a buffer of all join results produced and cannot yet be reported as the top k results. Estimating the maximum buffer size is an important parameter in estimating the total cost of a rank-join operator. In this experiment, we use Plan P in Figure 5.8. The left child rank-join operator in Plan P needs d_1 and d_2 tuples from its left and right inputs, respectively, before producing the top k results. The worst case (maximum) buffer size occurs when the rank-join operator cannot report any join result before retrieving all the d_1 and d_2 tuples.



(a) (b)

Figure 5.11. Estimating the Input Cardinality for Different Values of Join Selectivity

Hence, an upper bound on the buffer size can be estimated by $d_1 d_2 s$, where s is the join selectivity. We use our estimates for top-k depths, d_1 and d_2, to estimate the upper bound of the buffer size. We compare the actual (measured) buffer size to the following two estimates: (1) *Actual upper-bound*, the upper bound computed using the measured depths d_1 and d_2, and (2) *Estimated upper-bound*, the upper bound computed using our estimation of top-k depths.

Figure 5.12 shows that the estimated upper-bound has an estimation error less than 40% of the actual upper-bound (computed using the measured values of d_1 and d_2). Figure 5.12 also shows that the actual buffer size is less than the upper-bound estimates. The reason being that in the average case, the operator progressively reports ranked join results from the buffer before completing the join between the d_1 and d_2 tuples. The gap between the actual buffer size and the upper-bound estimates increases with k, as the probability of the worst-case scenario decreases.

5.5 Related Work

Another approach to evaluate top-k queries is the filter/restart approach [6, 20–22]. Ranking is mapped to a filter condition with a cutoff parameter. If the filtering produces less than k results, the query is *restarted* with a less restrictive condition.

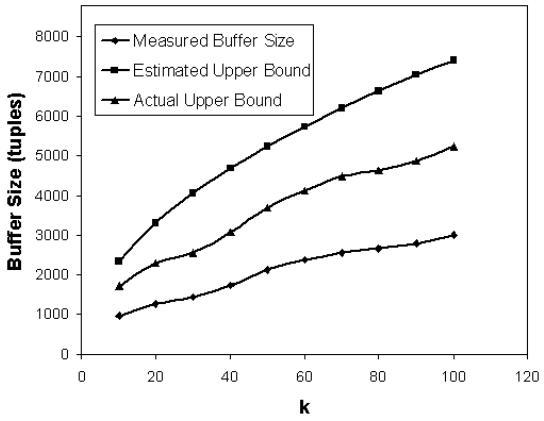


Figure 5.12. Estimating the Buffer Size of Rank-join

The final output results are then sorted to produce the top k results. A probabilistic optimization of top-k queries is introduced in [22] to estimate the optimal value of the cutoff parameter that minimizes the total cost including the risk of restarts. Optimizing top-k queries that contain only selection has been studied in [44] in the context of querying multimedia repositories. The optimization in [44] focuses on determining the best way to execute a set of filtering conditions given different costs of searching and probing the available indexes.

In contrast to previous work, we focus on optimizing ranking queries that involve joins. Moreover, our ranking evaluation encapsulates optimal rank aggregation algorithms. To the best of our knowledge, this is the first work that tries to estimate the cost of optimal rank aggregation algorithms and incorporate them in relational query optimization. We believe that the proposed optimization model for filtering operations in [44] can be used in tandem with our proposed optimization technique for selection predicates.

5.6 Summary

In this cahptaer, we introduced a framework for integrating rank-join operators in real-world query optimizers. Our framework was based on two key steps. First,

we extended the enumeration phase of the query optimizer to generate rank-aware plans. The extension was achieved by providing rank-join operators as possible join choices, and by defining ranking expressions as a new physical plan property. The new property triggered the generation of a new space of ranking plans either "naturally" by using rank-join operators or "enforced" by gluing sort operators to sort the partial results. Next, we provided a probabilistic technique to estimate the minimum required input cardinalities by rank-join operators to produce top k join results. Estimating the minimum required input cardinalities emerged from realizing the unique "early-out" property of rank-join operator. Unlike traditional join operators, rank-join operators do not need to consume all their inputs. Hence, estimating the cost of rank-join operator depends on estimating the number of tuples required from the input.

Our proposed estimation model captured this property with estimation error less than 30% of the actually measured input cardinality under some reasonable assumptions on the score distributions. We also estimated the space needed by rank-join operators with estimation error less than 40%. We conducted several experiments to evaluate the accuracy of our estimation model and the validity of our enumeration extension. The results proved the concept and showed the robustness of our estimation to several parameters such as the number of required answers and the join selectivity.

6 CONCLUSION

The main goal of this thesis is to leverage relational query processors to efficiently support ranked retrieval, an increasingly important requirement by many emerging applications. Top-k queries provide the user with the k most important results of a given query ranked according to some scoring function. Top-k queries are dominant in applications such as multimedia databases, information retrieval, web databases and middleware design.

This dissertation introduces a new query processing paradigm that increases the awareness of databases of user preferences. The dissertation also fills the gap between theoretical development of ranking algorithms and the practical system-oriented incorporation of these techniques inside real-world database systems.

6.1 Summary of Contribution

The dissertation introduces three main contributions in rank-aware query processing. First, the dissertation introduces a new algorithm for combining multiple ranked inputs into one global ranking, where the inputs are joined according to arbitrary join conditions. The new ranking algorithm advances the state of the art of ranking algorithms by supporting arbitrary join conditions and exploiting all available access methods. The main idea is to consume only that part of the inputs that has enough information to produce the top k results. We theoretically prove the optimality of the algorithm in terms of the number of retrieved objects from inputs. Experimental evaluation shows that the new algorithm outperforms current naïve approaches by orders of magnitude and achieves significant performance benefit over previously proposed algorithms.

Second, the dissertation proposes an efficient implementation of the new algorithm in terms of physical join operators for easy integration in practical query processors. We addressed all the practical and engineering issues that allow for adopting the operators in real-world database systems. These issues include pipelining, scalability and adaptive join strategies.

The third contribution in this dissertation is developing a rank-aware query optimization framework that enables a full integration of a rank-join algorithm and operators inside real-world query processors. The framework includes novel approaches to extend current cost-based optimization techniques to enumerate rank-aware query execution plans. The framework also introduces a cost estimation model for rank aggregation algorithms. To the best of our knowledge, this is the first model to estimate the required input size of optimal rank aggregation algorithms. The model by itself is very useful outside the context of query optimization and gives a better understanding of the complexity of rank aggregation algorithms.

The research in this dissertation was primarily motivated by a prototype of a video database management system developed by the database group at Purdue University. The system was built by modifying and augmenting an open-source database management system to support video storage, streaming and retrieval by content. In a typical query, the user provides an example image or multiple images and requests the most similar video objects based on multiple extracted features. Current relational query processors are based on boolean logic and have no notion for fuzziness or ranking.

6.2 Future Extensions

Throughout the study presented in this dissertation, several research questions were raised both in supporting ranking and user preference in query processing, and in the bigger picture of ameliorating the database systems "awareness" of available

context information. In this section, we give an overview of several directions for future research.

6.2.1 Approximate Ranking

The algorithms and techniques discussed in this dissertation focus on reporting the "exact" top k answers to a user query. In several scenarios, this can be unnecessarily expensive. For example, consider an image database where we are interested in retrieving the top 10 similar images to a given query. Now assume that there are only $n < 10$ images that are considered "similar" to the query image; all other images in the database are almost the same in terms of their similarity to the query. This scenario is not uncommon in similarity search according to [45]. In this particular scenario, the ranking algorithm will spend most of the execution time in distinguishing among "irrelevant" results trying to get the exact top k answers. This example motivates two research challenges:

- How to identify the number "relevant" top k answers ?
- How to adapt the behavior of the ranking algorithm to enhance the performance using the relevance information?

One approach is to compute approximate top k answers with some guarantees on the quality of the results.

6.2.2 Budget-guided Ranking

An important research challenge arises when we are interested in computing the top k answers, given some constraints on the available resources or "budget". For example, in online applications, users may be interested in getting query answers within some time limits. Another example in limited resources environments is when we are interested in computing the top k results with limited space or access

capabilities. In these examples, the proposed techniques to answer ranking queries may suffer from these constraints on time and space complexities and may become inapplicable.

We can see the connection between this challenge and approximate ranking; one way to address resources constraints is through the support of approximate query answers. What remains to be challenging is how to continuously adapt the approximation techniques to the available resources. We also intend to explore other approaches that includes sampling and using history information to adapt to resource constraints.

6.2.3 Learning the Scoring Function

In all the algorithms presented in this dissertation, we assumed that the combining scoring function is specified by the user. Although this assumption may be acceptable in simple scenarios, requiring users to specify how to combine multiple criteria can be very hard and even impractical. An interesting challenge is to be able to learn the combining function by collecting user preferences. Tackling this challenge will involve techniques from computer human interaction and machine learning. One obvious difficulty is how to ensure that the learned combining function has the properties required by the algorithm.

Moreover, this research direction can trigger other challenges with respect to the scoring function. For example, if the learned function is not monotone with respect to the individual scores, what are the other properties we can exploit to enhance over the naïve sorting approach.

6.2.4 Other Extensions

Other possible research directions include: (1) secure top-k computation from multiple sources, raising several challenges in privacy preserving query processing and insuring that users cannot infer more information than specified by the query;

(2) Supporting preference queries as a generalization of ranking queries, including supporting partial order on database objects and efficient evaluation of other query types, e.g., skyline queries; (3) Handling large ranking dimensions for web settings.

LIST OF REFERENCES

LIST OF REFERENCES

[1] C. J. van Rijsbergen. *Information Retrieval.* Butterworths, 1979.

[2] A. P. de Vries and A. N. Wilschut. On the integration of IR and databases. In *8th IFIP 2.6 Working Conference on Database Semantics*, January 1999.

[3] Navin Kabra, Raghu Ramakrishnan, and Vuk Ercegovac. The QUIQ engine: A hybrid IR-DB system. In *Proceedings of the* 19^{th} *International Conference on Data Engineering (ICDE), Bangalore, India*, pages 741–743. IEEE Computer Society, 2003.

[4] Pat Selinger. Information integration and XML in IBM's DB2. In *Proceedings of the* 28^{th} *International Conference on Very Large Databases (VLDB), Hong Kong, China*, pages 906–907. Morgan Kaufmann, 2002.

[5] Cynthia Dwork, S. Ravi Kumar, Moni Naor, and D. Sivakumar. Rank aggregation methods for the web. In *Proceedings of the* 10^{th} *International Conference on World Wide Web, Hong Kong, China*, pages 613–622, 2001.

[6] Nicolas Bruno, Surajit Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Transactions on Database Systems (TODS)*, 27(2):369–380, 2002.

[7] 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 ACM SIGMOD International Conference on Management of Data, Dallas, Texas*, pages 391–402. ACM, May 2000.

[8] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, California*, pages 259–270. ACM, May 2001.

[9] Panayiotis Tsaparas, Themistoklis Palpanas, Yannis Kotidis, Nick Koudas, and Divesh Srivastava. Ranked join indices. In *Proceedings of the* 19^{th} *International Conference on Data Engineering (ICDE), Bangalore, India*, pages 277–288. IEEE Computer Society, 2003.

[10] Walid G. Aref, Ann C. Catlin, Ahmed K. Elmagarmid, J. Fan, Moustafa A. Hammad, Ihab F. Ilyas, Mirette Marzouk, Sunil Prabhakar, and X. Zhu. VDBMS: A testbed facility for research in video database benchmarking. *ACM Multimedia Systems Journal, Special Issue on Multimedia Document Management Systems*, 2003.

[11] Praveen Seshadri and Mark Paskin. Predator: An OR-DBMS with enhanced data types. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, pages 568–571. ACM, May 1997.

[12] Storage manager architecture. *Shore documentation, Computer Sciences Department, UW-Madison*, June 1999.

[13] Joseph M. Hellerstein, Jeffrey F. Naughton, and Avi Pfeffer. Generalized search trees for database systems. In *Proceedings of* 21^{th} *International Conference on Very Large Data Bases (VLDB), Zurich, Switzerland*, pages 562–573. Morgan Kaufmann, 1995.

[14] Berthold Reinwald and Hamid Pirahesh. SQL open heterogeneous data access. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Seattle, Washington*, pages 506–507. ACM, 1998.

[15] Berthold Reinwald, Hamid Pirahesh, Ganapathy Krishnamoorthy, George Lapis, Brian T. Tran, and Swati Vora. Heterogeneous query processing through SQL table functions. In *Proceedings of the* 15^{th} *International Conference on Data Engineering (ICDE), Sydney, Australia*, pages 366–373. IEEE Computer Society, 1999.

[16] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Joining ranked inputs in practice. In *Proceedings of the* 28^{th} *International Conference on Very Large Databases (VLDB), Hong Kong, China*, pages 950–961. Morgan Kaufmann, 2002.

[17] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of the* 29^{th} *International Conference on Very Large Databases (VLDB), Berlin, Germany*, pages 754–765. Morgan Kaufmann, 2003.

[18] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *The VLDB Journal, to appear.*

[19] Ihab F. Ilyas, Rahul Shah, Walid G. Aref, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware query optimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France*. ACM, June 2004.

[20] Michael J. Carey and Donald Kossmann. On saying "Enough already!" in SQL. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Tucson, Arizona*, pages 219–230. ACM, May 1997.

[21] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *Proceedings of the* 24^{th} *International Conference on Very Large Databases (VLDB), New York*, pages 158–169. Morgan Kaufmann, August 1998.

[22] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top N queries. In *Proceedings of the* 25^{th} *International Conference on Very Large Databases (VLDB), Edinburgh, Scotland, UK*, pages 411–422. Morgan Kaufmann, 1999.

[23] Ronald Fagin. Combining fuzzy information from multiple systems. *Journal of Computer and System Sciences (JCSS)*, 58(1):216–226, February 1999.

[24] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the* 20^{th} *ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), Santa Barbara, California*, pages 102–113. ACM, May 2001.

[25] Surya Nepal and M. V. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of the* 15^{th} *International Conference on Data Engineering (ICDE), Sydney, Australia*, pages 22–29. IEEE Computer Society, 1999.

[26] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Optimizing multi-feature queries for image databases. In *Proceedings of the* 26^{th} *International Conference on Very Large Databases (VLDB), Cairo, Egypt*, pages 419–428. Morgan Kaufmann, 2000.

[27] Ulrich Güntzer, Wolf-Tilo Balke, and Werner Kießling. Towards efficient multi-feature queries in heterogeneous environments. In *Proceedings of the International Symposium on Information Technology (ITCC), Las Vegas, Nevada*, pages 622–628. IEEE Computer Society, 2001.

[28] Kevin Chen-Chuan Chang and Seung won Hwang. Minimal probing: supporting expensive predicates for top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Madison, Wisconsin*, pages 346–357. ACM, 2002.

[29] 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* 27^{th} *International Conference on Very Large Databases (VLDB), Rome, Italy*, pages 281–290. Morgan Kaufmann, 2001.

[30] M.-J. Condorcet. *Éssai sur l'application de l'analyse à la probabilité des décisions rendues à la puralité des voix*, 1785.

[31] P. Diaconis and R. Graham. Spearman's footrule as a measure of disarray. *Journal of the Royal Statistical Society*, 39(2):262–368, 1977.

[32] P. Diaconis. Group representation in probability and statistics. *IMS Lecture Series 11, IMS*, 1988.

[33] J. C. Borda. Memoire sur les elections au scrutin. *Histoire de l'Academie Royale des Sciences*, 1981.

[34] Mary Tork Roth, Manish Arya, Laura M. Haas, Michael J. Carey, William F. Cody, Ronald Fagin, Peter M. Schwarz, Joachim Thomas II, and Edward L. Wimmers. The garlic project. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Montreal, Quebec, Canada*, page 557. ACM, June 1996.

[35] Nicolas Bruno, Luis Gravano, and Amelie Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the* 18^{th} *International Conference on Data Engineering (ICDE), San Jose, California*, pages 153–187. IEEE Computer Society, 2002.

[36] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Philadelphia, Pennsylvania*, pages 287–298. ACM, june 1999.

[37] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in XPRS. *Distributed and Parallel Databases*, 1(1), Jan. 1993.

[38] Annita N. Wilschut and Peter M. G. Apers. Dataflow query execution in a parallel main-memory environment. *Distributed and Parallel Databases*, 1(1):68–77, 1993.

[39] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path election in a relational database management system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts*, pages 23–34. ACM, 1979.

[40] Goetz Graefe and William J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *Proceedings of the Ninth International Conference on Data Engineering (ICDE), Vienna, Austria*, pages 209–218. IEEE Computer Society, 1993.

[41] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD international conference on Management of data, San Francisco, California*, pages 160–172. ACM Press, 1987.

[42] Guy M. Lohman. Grammar-like functional rules for representing query optimization alternatives. In *Proceedings of the ACM SIGMOD international conference on Management of data, Chicago, Illinois*, pages 18–27. ACM Press, 1988.

[43] Norio Katayama and Shin'ichi Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2), 1997.

[44] Surajit Chaudhuri, Luis Gravano, and Amelie Marian. Optimizing top-k selection queries over multimedia repositories. *IEEE Transactions on Knowledge and Data Engineering, to appear.*

[45] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. When is "nearest neighbor" meaningful? *Lecture Notes in Computer Science*, 1540:217–235, 1999.

VITA

VITA

Ihab Ilyas was born in the beautiful Alexandria, Egypt in 1973. He spent his early school years engaging in a number of activities. He became a boy scout in 1986 and spent most of the summers camping and traveling all over Egypt.

Ihab earned his high school diploma in 1990. He was ranked third in Egypt in the nation-wide high school diploma exam. Being so passionate for computer science, Ihab joined Alexandria Faculty of Engineering where the first computer science department in Egypt was established in 1969. After a hard and a very competitive freshman year, Ihab was ranked fourth and joined the Computer Science Department. In 1995, Ihab was granted his B.S. degree with the highest degree of honor, ranked first in his class. Upon graduation, Ihab was awarded a teaching assistantship in the Department of Computer Science, Alexandria University. After spending one year serving in the military, Ihab pursued graduate study in the same department and was granted the masters degree in 1999 with the highest degree of honor. Ihab got married to his sole-mate, Mirette Marzouk in 1997. Ihab Ilyas then decided to go to the United States to pursue his Ph.D. degree. He joined Purdue in 1999 after he was granted a research assistantship by his advisor, Ahmed Elmagarmid. At Purdue, Ihab shaped his research attitude and developed significant interest in database systems. Working with two wonderful advisors, Ahmed Elmagarmid and Walid Aref, Ihab published several papers in core database technology. In 2002 and 2003, Ihab spent two summers in IBM Almaden Research Center, one of the top research labs in the field world-wide. During his summer internships, Ihab interacted with many world-class researchers, built his systems experience and appreciated the combination of theoretical soundness and system practicality. Ihab Ilyas graduated with a Ph.D. degree from Purdue in August 2004 and joined the School of Computer Science at the University of Waterloo, Canada to become an assistant professor.