RankSQL: Query Algebra and Optimization for Relational Top-k Queries*

Chengkai Li[1] Kevin Chen-Chuan Chang[1] Ihab F. Ilyas[2] Sumin Song[1]
[1]Department of Computer Science, University of Illinois at Urbana-Champaign
cli@uiuc.edu, kcchang@cs.uiuc.edu, ssong4@uiuc.edu
[2]School of Computer Science, University of Waterloo
ilyas@uwaterloo.ca

ABSTRACT

This paper introduces RankSQL, a system that provides a systematic and principled framework to support efficient evaluations of ranking (*top-k*) queries in relational database systems (RDBMS), by extending relational algebra and query optimization. Previously, *top-k* query processing is studied in the middleware scenario or in RDBMS in a "piecemeal" fashion, *i.e.*, focusing on specific operator or sitting outside the core of query engines. In contrast, we aim to support ranking as a first-class database construct. As a key insight, the new ranking relationship can be viewed as another logical property of data, parallel to the "membership" property of relational data model. While membership is essentially supported in RDBMS, the same support for ranking is clearly lacking. We address the fundamental integration of ranking in RDBMS in a way similar to how membership, *i.e.*, Boolean filtering, is supported. We extend relational algebra by proposing a *rank-relational* model to capture the ranking property, and introducing new and extended operators to support ranking as a first-class construct. Enabled by the extended algebra, we present a pipelined and incremental execution model of ranking query plans (that cannot be expressed traditionally) based on a fundamental *ranking principle*. To optimize *top-k* queries, we propose a dimensional enumeration algorithm to explore the extended plan space by enumerating plans along two dual dimensions: ranking and membership. We also propose a sampling-based method to estimate the cardinality of rank-aware operators, for costing plans. Our experiments show the validity of our framework and the accuracy of the proposed estimation model.

1. INTRODUCTION

Ranking queries (or *top-k* queries) are dominant in many emerging applications, *e.g.*, similarity queries in multimedia databases, searching Web databases, middlewares, and data mining. Top-k queries aim at providing only the top k query results, according to a user-specified ranking function, which in many cases is an aggregate of multiple criteria.

The increasing importance of *top-k* queries warrants an efficient support of ranking in the relational database management system (RDBMS) and has recently gained the attention of the research community. However, most of the available solutions to supporting ranking queries are in the middleware scenario [10, 26, 16, 11, 2, 4], or in RDBMS in a "piecemeal" fashion, *i.e.*, focusing on specific types of operator [3, 22, 23] and queries [25, 21], or sitting outside the core of query engines [6, 5, 29, 14, 15, 20, 31]. Hence, *top-k* queries are not treated as first-class query type, losing the advantages of integrating *top-k* operations with other relational operations.

Fundamental support of ranking queries is lacking mainly because relational algebra has no notion for ranking. Therefore, supporting ranking queries in RDBMS's as a first-class query type is a significant research challenge. In this paper, we present the RankSQL system, which aims at providing a seamless support and integration of *top-k* queries with the existing SQL query facility in relational database systems. The following is an example of *top-k* query.

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

```
SELECT   *
FROM     Hotel h, Restaurant r, Museum m
WHERE    c1 AND c2 AND c3
ORDER BY p1 + p2 + p3
LIMIT    k
```

■

With current relational query processing capabilities, the only way to execute the previous query is to: (1) consume all the records of the three inputs; (2) join the three inputs and materialize the whole join results; (3) evaluate the three predicates p_1, p_2, and p_3 for each valid join result; (4) sort the join results on $p_1 + p_2 + p_3$;

*This material is based upon work partially supported by NSF Grants IIS-0133199, IIS-0313260, and a 2004 IBM Faculty Award. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the author(s) and do not necessarily reflect the views of the funding agencies.

SIGMOD 2005 June 14-16, 2005, Baltimore, Maryland, USA.

and (5) report only the top k results to the user. Processing the query in this way suffers from the following problems:

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

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

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

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

The rest of the paper is organized as follows. We start in Section 2 by defining and motivating ranking queries as first-class construct. Section 3 introduces the rank-relational algebra. Section 4 introduces the execution model and physical implementation of ranking query plans. We present our proposed rank-aware query optimization in Section 5. We describe the experimental evaluation in Section 6 and review related work in Section 7. Finally, we conclude the paper in Section 8.

2. RANKING QUERY MODEL

This section defines rank-relational queries (Section 2.1), and motivates the need for supporting ranking as a first-class construct (Section 2.2).

2.1 Rank-Relational Queries

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

$$Q = \pi_* \lambda_k \tau_{\mathcal{F}(p_1,\ldots,p_n)} \sigma_{\mathcal{B}(c_1,\ldots,c_m)}(R_1 \times \cdots \times R_h) \qquad (1)$$

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

- **Filtering**: a *Boolean function* $\mathcal{B}(c_1, \ldots, c_m)$ filters the results by the *selection* operator $\sigma_{\mathcal{B}}$ (*e.g.*, $\mathcal{B} = c_1 \wedge c_2 \wedge c_3$ for Example 1), and
- **Ranking**: a *monotonic scoring function* $\mathcal{F}(p_1, \ldots, p_n)$ ranks the results by the sorting[1] operator $\tau_{\mathcal{F}}$ (*e.g.*, $\mathcal{F} = p_1 + p_2 + p_3$ for Example 1).

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

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

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

Our goal is to support such rank-relational queries efficiently. As our discussion above reveals, such queries add a *ranking* dimension to query processing and optimization, which in many ways parallels the traditional dimension of *filtering*: While filtering restricts tuple "*membership*" by applying a function $\mathcal{B}$ of Boolean selection or join predicates, ranking restricts "*order*" by applying a function $\mathcal{F}$ of corresponding ranking predicates. While Boolean predicates can be of various costs, ranking predicates share the same concern. We thus ask, while conceptually parallel, are they both well supported in RDBMS?

2.2 Ranking as First-Class Construct

Unlike Boolean "filtering" constructs, which are essentially supported in RDBMS, the same support for "ranking" is clearly lacking. To motivate, observe that as Eq. 1 shows, relational algebra provides the *selection* operator $\sigma_{\mathcal{B}}$ for filtering, and the *sorting* operator $\tau_{\mathcal{F}}$ for ranking. However, as we will see, there is a significant gap between their support in current systems.

Relational algebra models Boolean filtering, *i.e.*, $\sigma_{\mathcal{B}(c_1,\ldots,c_m)}$, as a *first-class* construct in query processing. (Such filtering includes both selections on a single table as well as joins.) With algebraic support for optimization, Boolean filtering is virtually never processed in the canonical form (of Eq. 1)– Consider, for instance, $\mathcal{B} = c_1 \wedge c_2$, for c_1 as a selection over R and c_2 a join condition

[1]Note that sorting is defined in the *extended* relational algebra to model the **ORDER BY** of SQL.

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

over $R \times S$. The algebra framework supports *splitting* of selections (*e.g.*, $\sigma_{c_1 \wedge c_2}(R \times S) \equiv \sigma_{c_1}\sigma_{c_2}(R \times S) \equiv \sigma_{c_1}(R \bowtie_{c_2} S)$) and *interleaving* them with other operators (*e.g.*, $\sigma_{c_1}(R \bowtie_{c_2} S) \equiv \sigma_{c_1}(R) \bowtie_{c_2} S$). These algebraic equivalences thus enable query optimization to transform the canonical form into efficient query plans by splitting and interleaving.

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

Such naïve *materialize-then-sort* scheme should not be the only choice– in fact, in many cases, it can be prohibitively expensive. If we want only the top k results, full materialization may not be necessary. As we shall see in Section 4, ranking predicates can significantly cut the cardinality of intermediate results. Moreover, all the ranking predicates have to be evaluated against every results of the full materialization under this naïve scheme. With the various costs, it may be beneficial in many cases to evaluate ranking predicates one by one, and interleave them with Boolean filtering. Thus, in a clear departure from the monolithic sorting τ, we believe rank-relational queries call for essentially supporting ranking as a first-class construct– in parallel with filtering. Such essential support, as we have observed, consists of two *requirements*:

1. **Splitting:** Ranking should be evaluated in stages, predicate by predicate– instead of monolithic.
2. **Interleaving:** Ranking should be interleaved with other operators– instead of always after filtering.

There are two major challenges in supporting ranking as a first-class operation. *First*, as foundation, we must extend relational algebra to handle ranking and define algebraic laws for equivalence transformations (Section 3). Meanwhile, to realize the algebra, we must define the corresponding query execution model and physical operators in which "rank-relations" are processed incrementally (Section 4). *Second*, we need to generalize query optimization techniques for integrating the parallel dimensions of Boolean filtering (*e.g.*, join order selection) and ranking (Section 5).

3. RANK-RELATIONAL ALGEBRA

To enable rank-aware query processing and optimization, we extend relational algebra to be rank-relational algebra, where the relations, operators, and algebraic laws "respect" and take advantage of the essential concept of "rank". In this section, we define rank-relational model (Section 3.1) and extend relational algebra (Section 3.2). The new rank-relational algebra enables and determines our query execution model and operator implementations. We also discuss several laws (Section 3.3) of the new algebra to lay the foundation of query optimization.

3.1 Rank-Relations: Ranking Principle

To fundamentally support ranking, the notion of *rank* must be captured in the relational data model. Thus, to start with, we must extend the semantics of relations to be rank-aware. In this extended model, we define *rank-relation* as a relation with its tuples *scored* (by some ranking function) and *ordered* accordingly.

In this model, how should we rank a relation?– Note that our algebra extension is to support rank-relational queries: Given a scoring function $\mathcal{F}(p_1, \ldots, p_n)$ for such a query (as in Eq.1), what are the rankings of tuples as they progress in processing? Consider a base relation R. Figure 1 conceptually illustrates the query tree.

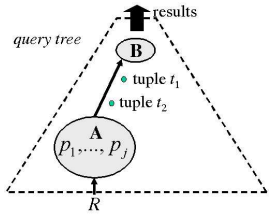


Figure 1: Ranking of intermediate relations.

To begin with, when no ranking predicate p_i is evaluated, R as tuples "on the disk" has an arbitrary order. As the splitting requirement (Section 2.2) motivates, these ranking predicates will generally be processed in stages. We thus ask, when some predicates, say $\mathcal{P} = \{p_1, \ldots, p_j\}$ (for $j < n$) are evaluated (Figure 1, cloud "A"), what should be the ranking? Note that although the final results are to be ranked by $\mathcal{F}(p_1, \ldots, p_n)$, at this stage we do not have the complete scores of all the predicates. Therefore, we want to define a *partial* ranking of tuples by their current incomplete scores, so that the resulted order is consistent with their "desired" order of further processing. As queries are evaluated incrementally by "iterators" (Section 4), this ranking will order the output tuples to subsequent operations (Figure 1, cloud "B"). Thus, refer to Figure 1, when should a tuple t_1 be ranked before t_2? It turns out that we have the following *ranking principle*.

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

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

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

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

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

- (**Scores**) The *score* for a tuple t is the maximal-possible score of t under $\mathcal{F}$, when the predicates in $\mathcal{P}$ have been evaluated, *i.e.*, $\overline{\mathcal{F}}_{\mathcal{P}}[t]$. It is an implicit attribute of the rank-relation.
- (**Order**) An order relationship $<_{R_{\mathcal{P}}}$ is defined over the tuples in $R_{\mathcal{P}}$ by ranking their scores, *i.e.*, $\forall t_1, t_2 \in R_{\mathcal{P}}$: $t_1 <_{R_{\mathcal{P}}} t_2$ iff $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] < \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$. ∎

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

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

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

(a) R

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

(b) R'

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

(c) S

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

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

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

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

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

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

Figure 2: Examples of rank-relations.

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

The extended *rank-relational algebra* generally operates on rank-relations. Thus, base relations, intermediate relations, and the results are all rank-relations. That is, rank-relations are *closed* under the algebra operators, which Section 3.2 will define, since all operators will account for the new ranking property (in addition to "membership"). Note that a base or intermediate relation, when no predicates are evaluated ($\mathcal{P} = \phi$), is consistently denoted R_ϕ or simply R. On the other hand, when $\mathcal{P} = \{p_1, \ldots, p_n\}$, the partial score is effectively complete, resulting in the final ranking with respect to $\mathcal{F}$.

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

3.2 Operators

We next extend the relational-algebra operators for manipulating rank-relations. Recall that, by Definition 1, a rank-relation $R_\mathcal{P}$ essentially possesses two logical properties– 1) *membership* as defined by the relation R, and 2) *order* induced by predicates $\mathcal{P}$ (with respect to some scoring function $\mathcal{F}$). For manipulating these two properties, we extend relational algebra by adding a new rank operator μ and generalizing the existing operators to be "rank-aware". Figure 3 summarizes the definitions of these operators, and Figure 4 illustrates them with examples (as continued from Example 2), which we explain below in more details.

New Operator μ: For supporting ranking as a first-class construct, we propose to add a new operator, *rank* or μ. As Section 2.2 motivated, our goal is to satisfy the two requirements: splitting and interleaving. Essentially, we must be able to evaluate ranking predicates (p_i's in $\mathcal{F}$) one at a time– thus ranking is effectively split and can be interleaved with other operations.

The new rank operator (μ) is thus a critical basis of our algebra. As Figure 3 defines, $\mu_p(R_\mathcal{P})$ evaluates an additional predicate p upon rank-relation $R_\mathcal{P}$, ordered by evaluated predicate set $\mathcal{P}$ as Definition 1 states, and produces a new order by $\mathcal{P} \cup \{p\}$– That is,

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

Figure 3: Operators defined in the algebra.

by definition, $\mu_p(R_\mathcal{P}) = R_{\mathcal{P}\cup\{p\}}$. For instance, when μ_{p_2} operates on $R_{\{p_1\}}$ in Figure 2(d), the result rank-relation is shown in Figure 4(a), which equals $R_{\{p_1,p_2\}}$. Note that $R_{\{p_1,p_2\}}$ is already the final result for ranking R with $\mathcal{F}_1$ because $\mathcal{F}_1 = \sum(p_1, p_2)$.

Extended Operators $\pi, \sigma, \cup, \cap, -, \bowtie$: We extend the original semantics of existing operators with rank-awareness, and thus enable the interaction between the new μ and traditional Boolean operations. As we will see, in the extended algebra, the operations will now be aware of and compute on dual logical properties– both membership (by Boolean predicate) and order (by ranking predicate). (Note that we omit projection π in Figure 3, since it is obvious. We also omit the discussion on Cartesian-product since it is similar to join.)

To begin with, unary operators such as *selection* (and π not shown in Figure 3) process the tuples in the input rank-relation as in their original semantics, and simply maintains the same order as the input. Thus, in our notation, $\sigma_c(R_\mathcal{P}) \equiv (\sigma_c R)_\mathcal{P}$. That is, the selection with c on $R_\mathcal{P}$ manipulates only the *membership* of R, by applying c, and maintains the same *order* as induced by $\mathcal{P}$. An example is shown in Figure 4(b).

Further, most binary operators, such as *union* ($\cup$), *intersection* ($\cap$), and *join* ($\bowtie$), perform their normal Boolean operations, and at the same time output tuples in the "aggregate" order of the operands– Such aggregate order is induced by *all* the evaluated predicates from both operands. Thus, for instance, $R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2} \equiv (R \cap S)_{\{\mathcal{P}_1\cup\mathcal{P}_2\}}$, which similarly holds for $\cup$ and $\bowtie$. Examples are shown in Figure 4(c), (d), and (f).

Finally, *difference* ($-$) outputs tuples in the order of the outer input operand– since the other is effectively discarded. Thus, $R_{\mathcal{P}_1} - S_{\mathcal{P}_2} \equiv (R - S)_{\mathcal{P}_1}$. An example is shown in Figure 4(e).

3.3 Algebraic Laws

Query optimizers essentially rely on algebraic equivalences to enumerate or transform query plans in search of efficient ones. In

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

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Results of operators.

the extended rank-relational model and algebra, as the dual logical properties dictate, algebraic equivalences should result in not only the same membership but also the same order. By definition of our algebra, as just discussed, we can assert many algebraic equivalence laws. As we extended the algebra specifically to support ranking, Figure 5 gives several such equivalences relevant to ranking. Essentially, these laws concretely state the new freedom of *splitting* and *interleaving*, thus achieving our motivating requirements (Section 2.2)– That is, the rank-relational algebra indeed supports ranking as first-class, in parallel with Boolean filtering. These laws are directly from the definition of the algebra, therefore to save space, we leave the proof to the extended version of this paper and only briefly discuss their usage in query optimization. In particular, we explain the laws specifically centering around our two requirements:

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

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

The new algebraic laws lay the foundation for query optimization of ranking queries as algebraic equivalences define equivalent plans in the search space of query optimizers. As we will see in Section 5, these algebraic laws guide the designing of transformation rules in rule-based optimizers, as well as the plan enumeration and heuristics in bottom-up optimizers.

4. RANKING QUERY PLANS: EXECUTION MODEL AND PHYSICAL OPERATORS

In common database query engines, a query execution plan is a tree of physical operators as *iterators*, which have three interface methods that allow the consumer operator of a physical operator to fetch one result tuple at a time. The three basic interface methods are: (1) *Open* method that initializes the operator and prepares its internal state; (2) *GetNext* method that reports the next result upon each request; (3) *Close* method that terminates the operator and performs the necessary cleanup. During the execution, query results are drawn from the root operator, which draws tuples from

Proposition 1: Splitting law for μ
- $R_{\{p_1,p_2,...,p_n\}} \equiv \mu_{p_1}(\mu_{p_2}(...(\mu_{p_n}(R))...))$

Proposition 2: Commutative law for binary operator
- $R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2} \equiv S_{\mathcal{P}_2} \Theta R_{\mathcal{P}_1}, \forall \Theta \in \{\cap, \cup, \bowtie_c\}$

Proposition 3: Associative law
- $(R_{\mathcal{P}_1} \Theta S_{\mathcal{P}_2}) \Theta T_{\mathcal{P}_3} \equiv R_{\mathcal{P}_1} \Theta (S_{\mathcal{P}_2} \Theta T_{\mathcal{P}_3}), \forall \Theta \in \{\cap, \cup, \bowtie_c{}^a\}$

Proposition 4: Commutative laws for μ
- $\mu_{p_1}(\mu_{p_2}(R_{\mathcal{P}})) \equiv \mu_{p_2}(\mu_{p_1}(R_{\mathcal{P}}))$
- $\sigma_c(\mu_p(R_{\mathcal{P}})) \equiv \mu_p(\sigma_c(R_{\mathcal{P}}))$

Proposition 5: Pushing μ over binary operators
- $\mu_p(R_{\mathcal{P}_1} \bowtie_c S_{\mathcal{P}_2})$
  $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c S_{\mathcal{P}_2}$, *if only* R *has attributes in* p
  $\equiv \mu_p(R_{\mathcal{P}_1}) \bowtie_c \mu_p(S_{\mathcal{P}_2})$, *if both* R *and* S *have*
- $\mu_p(R_{\mathcal{P}_1} \cup S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cup S_{\mathcal{P}_2}$
- $\mu_p(R_{\mathcal{P}_1} \cap S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap \mu_p(S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) \cap S_{\mathcal{P}_2}$
- $\mu_p(R_{\mathcal{P}_1} - S_{\mathcal{P}_2}) \equiv \mu_p(R_{\mathcal{P}_1}) - S_{\mathcal{P}_2} \equiv \mu_p(R_{\mathcal{P}_1}) - \mu_p(S_{\mathcal{P}_2})$

Proposition 6: Multiple-scan of μ
- $\mu_{p_1}(\mu_{p_2}(R_\phi)) \equiv \mu_{p_1}(R_\phi) \cap_r \mu_{p_2}(R_\phi)$

[a] When join columns are available.

Figure 5: Some algebraic equivalence laws.

underlying operators recursively, till the scan operators. This provides an efficient *pipelining* evaluation strategy, unless the flow of tuples is stopped by a blocking operator such as sort or a blocking join implementation, in which case, intermediate results have to be materialized.

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

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

Fortunately rank-relational algebra both advocates and enables non-blocking plans. In this section, we show how ranking query plans, consisting of the new and extended operators, execute according to the ranking principle (Property 1) in Section 4.1 and present their physical implementations in Section 4.2.

4.1 Incremental Execution Model

To realize the rank-relational algebra, we extend the common execution model to handle ranking query plans, with two differences from traditional plans. *First*, operators incrementally output rank-relations $R_{\mathcal{P}}$ (Definition 1), *i.e.*, tuple streams pass through operators in the order of maximal-possible scores (upper-bounds) $\overline{\mathcal{F}}_{\mathcal{P}}[t]$ with respect to the associated ranking predicate set $\mathcal{P}$. As the ranking principle indicates, it is desirable that t_1 precedes t_2 in further processing if $\overline{\mathcal{F}}_{\mathcal{P}}[t_1] > \overline{\mathcal{F}}_{\mathcal{P}}[t_2]$. *Second*, the query has

an explicitly requested result size, k. The execution stops when k results are reported or no more results are available.

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

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

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

Select $*$ **From** S **Order By** $p_3 + p_4 + p_5$ **Limit** 1.

Figure 6 illustrates three equivalent plans. Plan (a) is a traditional plan consisting of a sorting and a sequential scan operator. It scans tuples from S, evaluates all predicates (p_3,p_4,p_5) for each tuple, buffers and sorts them based on their scores till all tuples are scanned. Plan (b) is a new plan enabled by the rank-relational algebra, with an index scan followed by two μ operators. The index scan accesses tuples in the order of p_3 values, where p_3 can be as simple as attribute or as complex as external or built-in function. (Such index is supported in DBMS's such as PostgreSQL.)

In these plans, the rank-relation R above each operator op contains the tuples that have ever been processed by op. The portion of R in gray color is the incremental output rank-relation from op to its upper operator op', thus is the incremental input rank-relation to op'. Therefore the rank-relation R' above op' contains the same tuples as the gray portion of R, although may in different order, since op' can apply one more predicate and thus result in a new order.

For example, consider μ_{p_4} in plan (b). It processed 3 tuples (s_2,s_1,s_3) during execution. Among them, s_2 and s_1 were drawn to μ_{p_5}, which processed 2 tuples (s_2 and s_1) and output s_2 as the top-1 answer since μ_{p_5} is the top operator in the plan tree.

Note that the order of tuples in the rank-relations are decided by semantics, according to the definition of rank-relation (Definition 1) and operators (Section 3.2). For example, μ_{p_4} must output tuples in the order by $\overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}$ since p_3 is accessed by the underlying operator $idxScan_{p_3}(S)$ and p_4 is evaluated by μ_{p_4}. Therefore s_2 must precede s_1 when output from μ_{p_4} since $\overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}[s_2] = 2.75 > \overline{\mathcal{F}}_{2_{\{p_3,p_4\}}}[s_1] = 2.5$.

We further illustrate how tuples flow, still using plan (b) as an example. The operator μ_{p_5} first draws s_2 from μ_{p_4}, then evaluates $p_5[s_2]$ and gets $\overline{\mathcal{F}}_{2_{\{p_3,p_4,p_5\}}}[s_2] = 2.55$. At this point μ_{p_5} cannot output s_2 yet (refer to our explanation in the paragraph right above Example 3). Therefore μ_{p_5} buffers s_2 in its ranking queue and draws the next tuple, s_1, from μ_{p_4}. It is sure at this point that μ_{p_5} can output s_2 as the top answer (again, refer to the paragraph above Example 3). After evaluating $p_5[s_1]$ and getting $\overline{\mathcal{F}}_{2_{\{p_3,p_4,p_5\}}}[s_1] = 2.4$, s_1 is buffered. The execution goes on in this way to get more query results. Other operators in plan (b) work in the same way and the whole plan tree is executed in pipeline by recursively drawing tuples, resulting in the diagram in Figure 6(b). ■

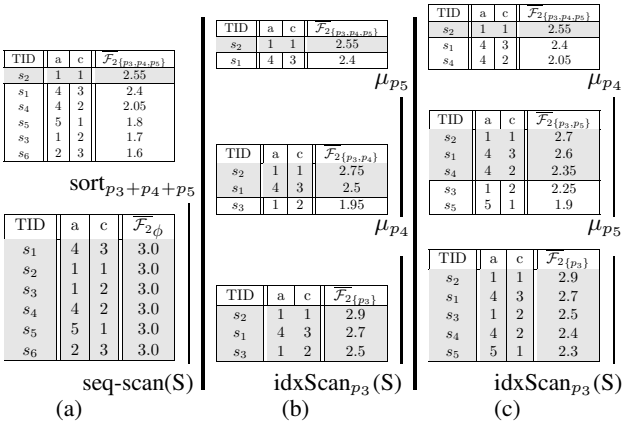


Figure 6: Ranking query plans vs. traditional plan.

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

Illustrated by the previous example, the execution model indicates that rank-aware operators are *selective*, *i.e.*, they reduce the cardinality of intermediate results as they do not output all of their processed tuples. For instance, the selectivity of μ_{p_4} in Figure 6(b) is $2/3$ as the rank-relation above it clearly shows.

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

Being selective enables operators to both reduce the evaluation of predicates that have various costs and reduce the cost of join, therefore ranking query plans do not need to materialize a query, in contrast to the traditional materialize-then-sort scheme of processing ranking queries. This makes ranking query plans much more efficient than traditional ones, which can be prohibitively expensive. Moreover, different scheduling and interleaving of rank-aware operators will result in different number of tuples being processed, therefore query optimizers have to non-trivially explore the new type of ranking plans (Section 5). Furthermore, the context-sensitiveness of selectivities indicate that cardinality estimation of these ranking plans will be challenging (Section 5.2).

Example 4: Continuing Example 3, this example shows that ranking query plans (Figure 6(b)(c)) outperform traditional plans (Figure 6(a)) and different ranking plans have different costs, thus it calls for query optimization.

Assume the costs of predicates p_3, p_4, and p_5 are C_3, C_4, and C_5, then the predicate evaluation cost of plan (a) is $6(C_3 + C_4 +$

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

4.2 Implementing Physical Operators

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

The implementation of μ is straightforward from Example 3 and it is a special case (because it schedules one predicate) of the algorithms (*MPro* [4], *Upper* [2]) for scheduling random object accesses in middleware *top-k* query evaluation. The implementation of $\bowtie_C$ adopts the *HRJN* (hash rank-join) and *NRJN*(nested-loop rank-join) algorithms in [22] [23], which are built upon symmetrical hash join [19, 30] or hash ripple join [17].

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

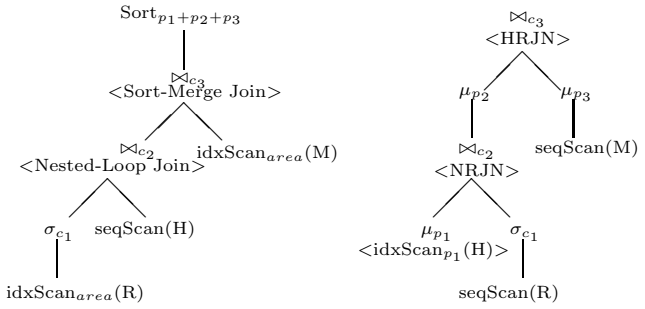
As another example, *scan* must be provided as a physical operator although it is not in relational algebra. Index-scan can be used to access tuples of a table in the order of some predicate p when there exists an index such as B+tree on p. (Thus we name it *rank-scan*.) Such index can be available when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

5. A GENERALIZED RANK-AWARE OPTIMIZER

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

Extending relational algebra to support ranking as introduced in Section 3 and Section 4 has direct impact on query optimization. In this section, we motivate the need for extending the query optimizer to support ranking and study the significant challenges associated with the extension. Then we show how to incorporate ranking into practical query optimizers used by real-world database systems.

The rank-relational algebra enables an extended plan space with plans that cannot be expressed traditionally. For instance, for the query in Example 1, traditional optimizers only allow *materialize-then-sort* plans such as the one in Figure 7(a). In contrast, the rank-relational algebra enables equivalent plans such as the one in Figure 7(b). The equivalence is guaranteed by the algebraic laws in Figure 5. First, the ranking function in the $sort$ operator is split into μ_{p_1}, μ_{p_2}, μ_{p_3} by Definition 1 and Proposition 1 of Figure 5. The μ operators are pushed down across join operators by Proposition 4 and 5. Note that μ_{p_1} is combined with scan operation to form an $idxScan$. Such splitting and interleaving may achieve significant improvements in performance as discussed in Section 4.1.



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

Figure 7: Two alternative plans for Example 1.

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

There are two categories of cost-based query optimizers used by real-world database systems, namely the top-down rule-based optimizers exemplified by Volcano [13] and Cascade [12], and the System R-style bottom-up dynamic programming optimization framework [28].

In Volcano and Cascade, *transformation* and *implementation* rules are the two key constructs used for searching the plan space. The transformation rules transform between equivalent algebraic expressions, and the implementation rules map logical operators into physical implementations to realize a plan tree. For extending rule-based optimizers with the rank-relational algebra, the algebraic laws presented in Section 3.3 naturally enable the introduction of new transformation rules to enumerate ranking plans. Implementation rules can be devised to trigger the mapping of physical algorithms presented in Section 4.2. Cost estimation in top-down optimizers can apply similar techniques for extending bottom-up optimizers since it only costs complete plans instead of subplans.

Extending bottom-up optimizers to incorporate ranking, however, is more challenging as plans are constructed and pruned in bottom-up fashion without global information of a complete plan. Therefore focusing on bottom-up optimizers, we show how to extend the System-R style bottom-up dynamic programming (DP) approach for plan enumeration (Section 5.1) and how to cost and prune plans during enumeration (Section 5.2).

5.1 Two-Dimensional Plan Enumeration

We take a principled way to extend DP plan enumeration by treating ranking predicates as another dimension of enumeration in addition to Boolean predicates, based on the insight that the ranking (order) relationship is another logical property of data, parallel to membership (Section 2.1). Recall that, by Definition 1, a rank-relation $R_{\mathcal{P}}$ essentially possesses two logical properties: Boolean membership (R) and ranking order ($\mathcal{P}$). In a ranking query plan, new ranking predicates are only introduced in μ operators. Therefore the predicate set $\mathcal{P}$ of a subplan, *i.e.*, the μ operators in a subplan, determines the order, just like how join conditions (together with other operations) determine the membership. Moreover, for the same logical algebra expression, the optimizer must be able to

Procedure 2_Dimension_Enumeration

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

Figure 8: 2-Dimension Enumeration Algorithm.

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

The concept of dimensional enumeration is general and extensible for naturally including more dimensions, *e.g.*, ordering other operators such as selection, union, intersection, *etc.* For example, scheduling selection predicates is traditionally considered less important than join order selection and is rather handled by heuristics such as selection pushdown. Under the situation that it is necessary to handle such task, as motivated in [9, 18, 8], dimensional enumeration can incorporate the scheduling of both selection and ranking predicates by treating Boolean predicates as another dimension. Due to space limitations, we focus on how to integrate the scheduling of ranking predicates and join order selection and omit the consideration of other operators.

The DP 2-dimensional enumeration algorithm is shown in Figure 8. For each subplan, we define its *signature* (S_R, $S_\mathcal{P}$) as the pair of two logical properties, the set of relations S_R and the set of ranking predicates $S_\mathcal{P}$ in the subplan. Subplans with the same signature result in the same rank-relation. The algorithm first enumerates the number of joined relations, $||S_R||$, then the number of ranking predicates, $||S_\mathcal{P}||$. Plans with the signature $(S_R, S_\mathcal{P})$ are generated by joining two plans with the signature $(S_{R1}, S_{\mathcal{P}1})$ and $(S_{R2}, S_{\mathcal{P}2})$ (joinPlan), adding a μ_p upon a plan with the signature $(S_R, S_\mathcal{P} - \{p\})$ (rankPlan), or using a scan operator (scanPlan). Based on the principle of optimality, no sub-optimal subplan can be part of the optimal execution strategy, hence for all the plans with the same signature, only the best plan is kept.

Example 5: We illustrate how the algorithm optimizes a simple query over the tables in Figure 2,

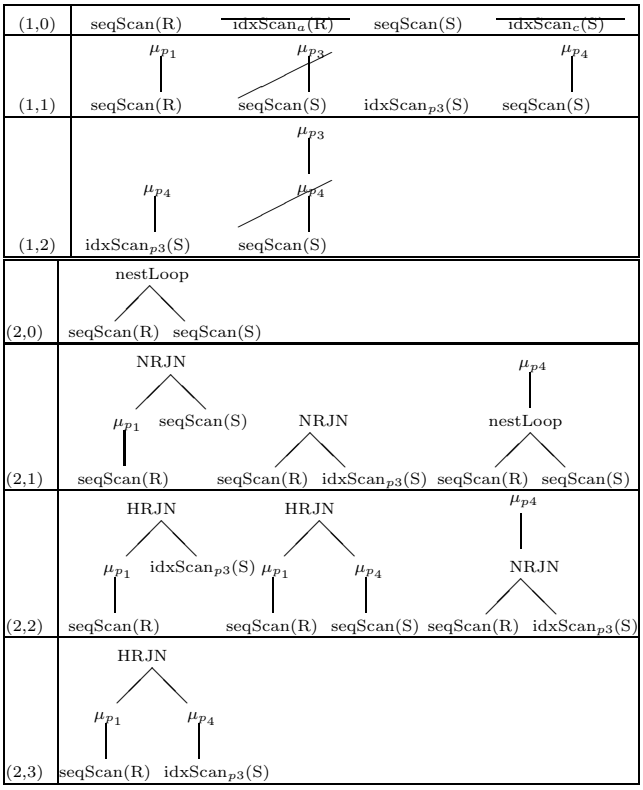


Figure 9: Plan Enumeration.

Select $*$ **From** R, S **Where** $R.a = S.a$
Order By $p_1 + p_3 + p_4$ **Limit** k.

In Figure 9, each row contains the best plans for signatures of the same size, with one best plan per signature. For instance, row $(2, 1)$ show the best plans for $(\{R, S\}, \{p_1\})$, $(\{R, S\}, \{p_3\})$, and $(\{R, S\}, \{p_4\})$ respectively. We also show the pruned plans (as crossed out) on single table, but omit that for joined relations due to space limitations.

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

One important detail of System-R algorithm is that multiple plans with the same logical properties may be kept if they have different physical properties. Example physical properties are *interesting orders* [28] that are potentially beneficial to subsequent operations. For instance, $idxScan_a$ can be kept since its sorted access on $R.a$ can be useful for sort-merge join when R is joined with S. In the dimensional enumeration algorithm, the support of physical properties is not affected. It can keep multiple plans that have different physical properties for the same signature. Note that interesting order will only be possessed by plans with empty predicate set (*i.e.*, $S_\mathcal{P} = \phi$), since by definition rank-relations must be output in the order with respect to $\mathcal{P}$, which is not the kind of order that is useful to operators such as sort-merge join.

The 2-dimensional enumeration algorithm is exponential in the number of the ranking predicates as well as the number of relations,

Procedure 2_Dimension_Enumeration_with_Heuristics
1: replace line 4 of Figure 8 with the following
2: **for** each pair S_{R1}, S_{R2} s.t. $S_R = S_{R1} \cup S_{R2}$, $||S_{R2}|| \leq 1$, $S_{R1} \cap S_{R2} = \phi$ **do**
3:
4: insert the following into Figure 8, between line 10 and 11
5: **if** $||S_{R2}|| = 0$, $S_{P2} = \{p_u\}$ and $\exists p_v$ s.t. $p_v \in P - S_P$ and $rank(\mu_{p_v}) > rank(\mu_{p_u})$ **then**
6: continue to line 10

Figure 10: Heuristics for improving efficiency.

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

Therefore, we propose a heuristic to reduce the space on the dimension of ranking predicates, as shown in Figure 10. The algorithm in Figure 10 modifies that in Figure 8 by incorporating the left-deep join heuristic (Line 2) and our new heuristic on the ranking predicate dimension (Line 4). The ranking predicate scheduling heuristic greedily appends μ operators in a sequence instead of considering all valid permutations of μ operators. Given a subplan $plan$, suppose $plan'$ is to be built by adding one μ upon $plan$. The optimizer does not use μ_{p_u} to build $plan'$ if there exists another applicable μ_{p_v} such that appending μ_{p_v} is (likely) better than appending μ_{p_u}. The goodness of appending μ_{p_u} upon $plan$, is based on its selectivity and cost, defined as $rank(\mu_{p_u}) = \frac{1-card(plan')/card(plan)}{cost(\mu_{p_u})}$, where $cost(\mu_{p_u})$ is the evaluation cost of p_u, and $card(plan')$ and $card(plan)$ are the output cardinalities of $plan'$ and $plan$. (This $rank$ should not be confused with the concept of rank in our algebra.) Therefore μ_{p_u} is appended upon $plan$ only if there exists no other applicable μ_{p_v} that has a higher rank. Intuitively the rank of a μ is higher if its cost is lower and its selectivity is smaller, *i.e.*, its power of reducing cardinality is higher. In the formula, $cost(\mu_{p_u})$ is one component of the cost model of μ_{p_u} that should be defined together with its implementation. Techniques for estimating the cardinality of a subplan is presented in Section 5.2.

The above greedy scheduling heuristic for ranking predicates is inspired by the $rank$ metric in [18] for scheduling independent selection predicates and the adaptive approach in [1] for ordering correlated filters in streaming data processing. The $rank$ metric in [18] guarantees an optimal fixed order of independent selection predicates, that is, a selection predicate should always be applied before another one if it has higher rank. However, the same property cannot be guaranteed for scheduling μ operators simply because of their context-sensitive selectivities (Section 4.1). We adopt $rank$ metric as a heuristic, just like applying left-deep join heuristic, which sacrifices optimality for efficiency as a common practice of query optimizers.

5.2 Costing Ranking Query Plans

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

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

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

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

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

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

Our experimental study (Section 6) shows that the simple sampling method with a small sample ratio (*e.g.*, 0.1%) gives accurate cardinality estimates. With small sample size, sampling method does not introduce much overhead to query optimization.

Accurate random sampling over joins has been known to be difficult [7]. We plan to investigate the possibilities of using techniques such as [7] in future work to improve our sampling method.

6. EXPERIMENTS

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

The experiments are conducted on a PC having a 1.7GHz Pentium-4 CPU with 256KB cache, 768MB RAM, and 30GB disk, running Linux 2.4.20 operating system. The `shared_buffers` (shared memory buffer size) and `sort_mem` (internal memory for sorting and hashing) settings in PostgreSQL are configured as 24MB and 20MB, respectively. We use a synthetic data set of three database tables (A, B, C) having the same size and schema. Table A and B each have one Boolean attribute with 0.4 as their selectivities. The three tables have $2, 2$, and 1 ranking predicates, respectively. The ranking predicates have the same cost. They are implemented as user-defined functions, taking attributes of the tables as parameters. Scores of different ranking predicates are within the range between 0 and 1 and are independently generated by different distributions, including uniform, normal (with mean 0.5 and variance 0.16), and cosine distributions. Each table has two attributes jc1 and jc2 as join columns.

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

```
Q =
SELECT *
FROM A, B, C
WHERE A.jc1=B.jc1 AND B.jc2=C.jc2 AND A.b AND B.b
ORDER BY f1(A.p1)+f2(A.p2)+f3(B.p1)+f4(B.p2)+f5(C.p1)
LIMIT k
```

Figure 11 illustrates four execution plans for the above query. $Plan1$ is a conventional materialize-then-sort plan, in which *filter* is the physical selection operator and sort-merge join is used as the physical join operator. $Plan2-4$ are new ranking query plans. The implementations of μ operator (*rank*), rank-aware join operator (*HRJN*), and rank-scan operator ($idxScan$) were described in Section 4.2. In $plan2$, rank-scans are used for accessing base tables and μ is scheduled before join. $Plan3$ uses sequential scan instead of rank-scan. $Plan4$ applies μ operators above normal sort-merge join to replace one of the *HRJN* operators.

6.1 Cost of Ranking Execution Plans

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

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

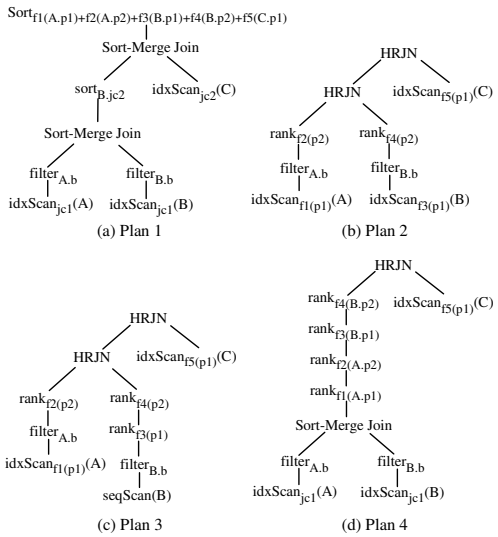


Figure 11: Execution Plans for Query $\mathcal{Q}$.

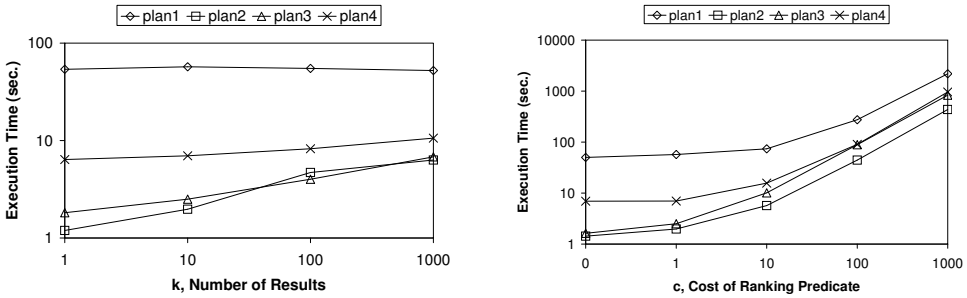
of the other three parameters. We then execute each plan under these parameter settings and measure their execution time. The results are shown in Figure 12. (Note that both x and y axes are in logarithmic scale.)

The figures illustrate that none of the plans is always the best under all situations. Moreover, different plans can have orders of magnitude differences in their costs. The diversity of plan costs verifies the need of query optimization in choosing efficient plans. Apparently, the traditional plan ($plan1$ in Figure 11) is far outperformed by rank-aware plans ($plan2-4$ in Figure 11). Its performance is only comparable to other plans when the size of tables and requested results are small, when joins are very selective, and when predicates are cheap. In many situations, the traditional plan becomes prohibitively expensive.

Specifically, Figure 12(a) shows that the traditional plan for ranking queries is blocking, while the new rank-aware plans are incremental. Figure 12(b) illustrates that the cost difference between plans increases (shown as parallel lines in logarithmic scale) together with the cost of predicates. This is because the predicate cost will dominate the plan execution cost while getting larger and the number of predicate evaluations does not change for a given plan when only predicate cost is changing. Figure 12(c) shows that the traditional plan is efficient when joins are very selective (thus performing join first will result in very small intermediate results, upon which ranking predicates are evaluated). Finally, Figure 12(d) shows that some ranking query plans (*e.g.*, $plan2$) are very efficient even with very large tables, while some others are not. For instance, $plan4$ was relatively acceptable in other situations, but became much less efficient than $plan2$ and $plan3$ when each table has 1 million tuples. Note that we remove $plan1$ from Figure 11(d) since it takes days to finish and is well off the scale.

6.2 Cardinality Estimation

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



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

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

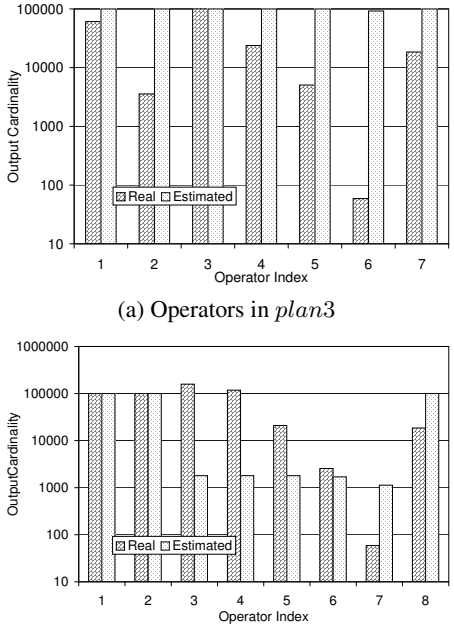
(c) $k = 10$, $s = 100,000$, $c = 1$

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

Figure 12: Performances of Different Execution Plans.

operators are estimated, since we do not estimate for the 2 selection operators and the root operator. Similarly $plan2$ and $plan4$ have the estimated cardinalities for 6 and 8 operators, respectively.

The experiment is based on a sample database with 0.1% sample ratio. Each of the original tables contains $100,000$ tuples and the join selectivity for the original tables is 0.0001. The number k is set to 10 (thus k' is 1). Figure 13 illustrates the estimation results of $plan3$ and $plan4$. The result of $plan2$ is very similar to that of $plan3$ therefore we do not include it. As we can see from the figure, although we used a very small sample, the real and estimated output cardinalities of majority of the operators are in the same magnitude, validating the estimation method.



(a) Operators in $plan3$

(b) Operators in $plan4$

Figure 13: Estimated and Real Output Cardinalities of Operators.

7. RELATED WORK

In this paper we introduce a systematic and principled framework, by extending relational algebra and query optimizers, to support ranking as first-class construct in relational database systems. We believe that our proposed framework is the first piece of work to fully integrate ranking in database systems on both the logical algebra level and the physical implementation level. Previously *top-k* query processing is studied in the middleware scenario or in RDBMS in a "piecemeal" fashion, *i.e.*, focusing on specific operator or sitting outside the core of query engines. In contrast, our framework provides principled algebra foundation and is not limited to a specific operation, thus allows for both expressing and optimizing general *top-k* queries. In the following, we highlight some of the recent effort in rank processing and other related work.

In middleware settings, various algorithms are proposed for rank aggregation on a set of objects, by merging multiple ranked lists [10, 26, 16, 11], or scheduling random accesses efficiently [2, 4], with the goal of minimizing number of accesses to objects. Although in a different setting, the works in [4, 2] explore the concept of upper-bound scores that inspires us to formalize our ranking principle for relational *top-k* queries. A similar sampling approach was applied in [4] to schedule predicates only, whereas we extend the approach to estimate the cost of general ranking query plans.

In RDBMS, there have been several proposals to support answering *top-k* queries at application level or outside the core of query engines [6, 5, 29, 14, 15, 20, 31], or for supporting special types of ranking queries [25, 21]. Recently, supporting *top-k* queries inside the relational query engine, in terms of physical query operators, has been proved to be an efficient approach that treats ranking as a basic database functionality [3, 21, 22, 23]. A *stop* operator is proposed in [3] to limit the cardinality of intermediate and query result, either conservatively by integrity constraints or aggressively

with the risk of restarting the query plan. The order supported by the stop operator is from columns of relations in SQL queries. Aggregation of multiple ranking criteria is not considered.

In [22] a new operator is devised for supporting rank join query, where rank join predicates coexist with Boolean join predicates. Instead of conducting normal join algorithms on Boolean join predicates, the rank-join operator progressively produces the join results. In [23] the relational query optimizer is further extended to utilize the rank-join operator in generating efficient query plans. We complement their work and together provide a systematic support of relational ranking queries, as we use rank-join as one of the rank-aware operators and at the same time supply an algebraic foundation of such support. Our dimensional enumeration framework enumerates plans by two dual logical properties to handle both scheduling of rank operators and join order selection, while [23] extends the "interesting order" (physical property) concept to deal with join enumeration only. The "interesting order" was also extended to support optimizing queries with expensive Boolean predicates [8]. The concept of our dimensional enumeration is general and extensible for more dimensions, including scheduling such Boolean predicates, union, and intersection operators.

With respect to the approach of extending query algebra, [24] proposes an algebra for capturing the semantic of preference queries. In [27] an algebra is proposed for expressing complex queries over *Web relations* that are used to model Web repositories. The algebra extension focuses on capturing the semantic of application-specific ranking and order relationships over Web pages and hyperlinks, instead of enabling efficient query processing.

8. CONCLUSION

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

9. REFERENCES

[1] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.

[2] N. Bruno, L. Gravano, and A. Marian. Evaluating top-k queries over web-accessible databases. In *ICDE*, 2002.

[3] M. J. Carey and D. Kossmann. On saying "enough already!" in SQL. In *SIGMOD*, pages 219–230, 1997.

[4] K. C.-C. Chang and S. Hwang. Minimal probing: Supporting expensive predicates for top-k queries. In *SIGMOD*, pages 346–357, 2002.

[5] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: indexing for linear optimization queries. In *SIGMOD*, pages 391–402, 2000.

[6] S. Chaudhuri and L. Gravano. Evaluating top-*k* selection queries. In *VLDB*, pages 397–410, 1999.

[7] S. Chaudhuri, R. Motwani, and V. Narasayya. On random sampling over joins. In *SIGMOD*, pages 263–274, 1999.

[8] S. Chaudhuri and K. Shim. Optimization of queries with user-defined predicates. In *VLDB*, pages 87–98, 1996.

[9] D. Chimenti, R. Gamboa, and R. Krishnamurthy. Towards an open architecture for $\mathcal{LDL}$. In *VLDB*, pages 195–203, 1989.

[10] R. Fagin. Combining fuzzy information from multiple systems. In *PODS*, pages 216–226, 1996.

[11] R. Fagin, A. Lote, and M. Naor. Optimal aggregation algorithms for middleware. In *PODS*, 2001.

[12] G. Graefe. The cascades framework for query optimization. *IEEE Data Eng. Bull.*, 18(3):19–29, 1995.

[13] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In *ICDE*, pages 209–218, 1993.

[14] S. Guha, D. Gunopulos, N. Koudas, D. Srivastava, and M. Vlachos. Efficient approximation of optimization queries under parametric aggregation constraints. In *VLDB*, pages 778–789, 2003.

[15] S. Guha, N. Koudas, A. Marathe, and D. Srivastava. Merging the results of approximate match operations. In *VLDB*, pages 636–647, 2004.

[16] U. Güntzer, W. Balke, and W. Kießling. Optimizing multi-feature queries in image databases. In *VLDB*, 2000.

[17] P. J. Haas and J. M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD*, pages 287–298, 1999.

[18] J. M. Hellerstein and M. Stonebraker. Predicate migration: Optimizing queries with expensive predicates. In *SIGMOD*, pages 267–276, 1993.

[19] W. Hong and M. Stonebraker. Optimization of parallel query execution plans in xprs. In *ICPDIS*, pages 218–225, 1991.

[20] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. *SIGMOD*, 2001.

[21] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Joining ranked inputs in practice. In *VLDB*, pages 950–961, 2002.

[22] I. F. Ilyas, W. G. Aref, and A. K. Elmagarmid. Supporting top-k join queries in relational databases. In *VLDB*, pages 754–765, 2003.

[23] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware query optimization. In *SIGMOD*, 2004.

[24] W. Kießling. Foundations of preferences in database systems. In *VLDB*, pages 311–322, 2002.

[25] A. Natsev, Y.-C. Chang, J. R. Smith, C.-S. Li, and J. S. Vitter. Supporting incremental join queries on ranked inputs. In *VLDB 2001*, pages 281–290, 2001.

[26] S. Nepal and M. V. Ramakrishna. Query processing issues in image(multimedia) databases. In *ICDE*, pages 22–29, 1999.

[27] S. Raghavan and H. Garcia-Molina. Complex queries over web repositories. In *VLDB*, pages 33–44, 2003.

[28] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *SIGMOD*, pages 23–34, 1979.

[29] P. Tsaparas, T. Palpanas, Y. Kotidis, N. Koudas, and D. Srivastava. Ranked join indices. In *ICDE*, 2003.

[30] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *ICPDIS*, pages 68–77, 1991.

[31] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient maintenance of materialized top-k views. In *ICDE*, 2003.