RankSQL: Supporting Ranking Queries in Relational Database Management Systems*

Chengkai Li[1] Mohamed A. Soliman[2] Kevin Chen-Chuan Chang[1] Ihab F. Ilyas[2]

[1]Department of Computer Science, University of Illinois at Urbana-Champaign
cli@uiuc.edu, kcchang@cs.uiuc.edu
[2]School of Computer Science, University of Waterloo
m2ali@.uwaterloo.ca, ilyas@uwaterloo.ca

1 Introduction

Ranking queries (or *top-k* queries) are dominant in many emerging applications, *e.g.*, similarity queries in multimedia databases, searching Web databases, middleware, and data mining. 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. 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 following is an example *top-k* query.

Example 1: This is a trip planning query in PostgreSQL syntax. The user 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 should cost less than \$100 (c_2: $h.price$+$r.price < 100$); the museum and the restaurant should be in the same city area (c_3: $r.area$=$m.area$). Results qualifying these conditions are ranked by a scoring function that sums up the numeric scores of several ranking "predicates"– p_1: $cheap\ (h.price)$ for a low hotel price; p_2: $close\ (h.addr, r.addr)$ for a close distance between the hotel and the restaurant; and p_3: $related\ (m.collection,$ "$dinosaur$"$)$ for matching user's interests with the museum's collections.

SELECT $*$
FROM Hotel h, Restaurant r, Museum m
WHERE c_1 AND c_2 AND c_3
ORDER BY $p_1 + p_2 + p_3$
LIMIT k ■

Current RDBMSs can only execute the above query in the following way: (1) exhaust the input tables and materialize the whole join results; (2) evaluate the three predicates p_1, p_2, and p_3 on each join result; (3) sort the join results by $p_1 + p_2 + p_3$; and (4) report the top k results. Such a naïve *materialize-then-sort* scheme can be prohibitively expensive. Since the user is only interested in the top k results instead of a total order, the full materialization is an overkill and introduces unnecessary overhead of fully scanning and joining the three inputs, which can be arbitrarily large. Moreover, all the ranking predicates have to be evaluated against every results of the full materialization. However, the ranking predicates, much like their boolean counterparts, can be expensive as they can be user-defined or built-in functions that require accessing online sources (*e.g.*, evaluating p_1 by a Web hotel database), comparing geographical data (*e.g.*, p_2), information retrieval style operation (*e.g.*, p_3), *etc.*

In RankSQL, we aim at providing a seamless support of *top-k* queries as a first-class query type and integrating this type of queries in the existing SQL query engines. Our system achieves orders of magnitude performance improvements when compared with traditional query processing techniques. This effort addresses the significant research challenge of making RDBMSs rank-aware, a fundamental notion that is missing in current database systems design. In general, supporting ranking imposes significant impacts on the whole system, including the underlying data model and algebra, the query operators, and the optimization techniques. In this demonstration, we show how to address these challenges in the RankSQL prototype.

New techniques: As the foundation, we extend relational algebra to be "rank-relational" algebra [3], to capture ranking as a first-class construct. We realize the algebra by an efficient query execution model [3] and new physical rank-aware operators [1, 2] in which "rank-relations" are processed incrementally. To enable rank-aware query optimization for constructing efficient ranking query plans, we propose a two-dimensional plan enumeration approach

*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.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

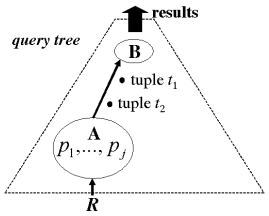


Figure 1: Ranking principle.

and a sampling-based cost estimation method [3] to address the key challenges in integrating the parallel dimensions of ranking and boolean filtering (*e.g.*, join order selection).

Contributions: To the best of our knowledge, RankSQL is the first RDBMS that fully integrates ranking support as a first-class functionality. In contrast, most related works are in the middleware scenario, or in RDBMS in a "piecemeal" fashion, *e.g.*, focusing on specific operator or specific types of queries, or sitting outside the core of query engine. Hence, *top-*k queries are not treated as a first-class query type, losing the advantages of integrating *top-*k operations with other relational operations.

2 Overview of RankSQL

The RankSQL prototype is built upon the open source database system PostgreSQL. In this section we present the algebraic foundation and the novel techniques developed for various system components.

2.1 The Foundation: Rank-Relational Algebra

As motivated by Section 1, essential support of ranking 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 (selections and joins).

Our rank-relational algebra incorporates "rank" as an essential concept according to these requirements. We extend the semantics of relations to be rank-aware by defining *rank-relation* as a relation with its tuples *scored* and *ordered* according to their maximal-possible scores (upper-bound) with respect to the set of evaluated ranking predicates. As the key insight, our ranking principle [3] indicates that given two tuples t_1 and t_2, if the upper-bound of t_1 is higher than that of t_2, then t_1 must be further processed if we necessarily further process t_2 for query answering (refer to Figure 1). Therefore the order of tuples based on their upper-bound scores is consistent with their "desired" order of further processing, as the more promising tuples are processed earlier, toward retrieving top-k results efficiently.

The rank-relation by the above extension essentially possesses two logical properties: (1) *membership* as defined by traditional boolean filtering operations (selection and join), and (2) *order* induced by evaluated predicates. Therefore we extend the relational-algebra operators for manipulating these two properties of rank-relations.

We first introduce a new operator, *rank* (μ), that evaluates an additional predicate p upon a rank-relation ordered by evaluated predicate set $\mathcal{P}$, and produces a new order by $\mathcal{P} \cup \{p\}$. The μ operator enables us to evaluate ranking predicates one at a time– thus ranking is effectively split and interleaved with other operations, achieving the aforementioned requirements.

We then extend the original semantics of existing operators ($\pi, \sigma, \cup, \cap, -, \bowtie$) with rank-awareness, and thus enable the interaction between the new μ and traditional boolean operations. For example, *rank-join* performs the normal boolean join operation, 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.

In the extended rank-relation model and algebra, as the dual logical properties dictate, algebraic equivalence should result not only in the same membership but also in the same order. By definition of our algebra, we can assert many equivalence laws. These new algebraic laws lay the foundation of optimizing ranking queries as they define equivalent plans in the search space of query optimizers.

2.2 The RankSQL Execution Engine

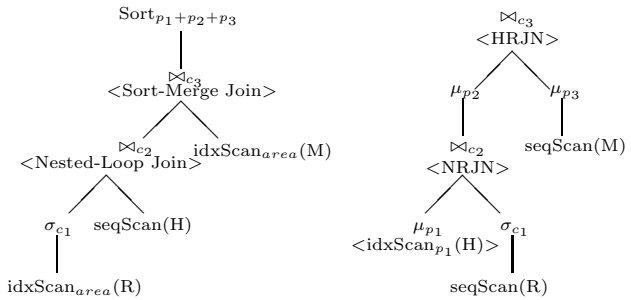
To realize the rank-relational algebra, we extend the common execution model to handle ranking query plans [3], with two properties: (1) operators incrementally output rank-relations, *i.e.*, tuple streams pass through operators in the order of maximal-possible scores according to our ranking principle; and (2) the query has an explicitly requested result size, k. The execution stops when k results are reported or no more results are available.

The implementation of μ utilizes the *MPro* algorithm [1]. For *rank-join*, we adopt the *HRJN* and the *NRJN* physical rank-join operators [2]. We also implement *rank-scan* for accessing a base table in the order of a ranking predicate p when there exists a B+tree index on p. Such an index can be available when p is some attribute, expression, or function, as all are supported in PostgreSQL. Discussions on implementing new algorithms for other operators such as $\cap$ can be found in [3].

The incremental nature of such ranking query plans enables returning top results progressively upon user requests. Therefore, the execution cost is proportional to k, in contrast to the blocking materialize-then-sort scheme which can only report the first result after all results (much more than k in general) are produced and sorted.

2.3 The RankSQL Query Optimizer

The rank-relational algebra and the new implementation of physical operators enable 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 (*e.g.*, Figure 2(a)). In contrast, rank-relational algebra enables equivalent plans by algebraic laws (*e.g.*, Figure 2(b)), which realizes the splitting



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

Figure 2: Two alternative plans for Example 1.

and interleaving requirements and may achieve significant improvements in performance.

To fully incorporate the new algebra into a cost-based query optimizer, we must address the significant impact of the extended plan space on plan enumeration and costing for effective pruning.

Plan Enumerator: We take a principled way to extend System-R bottom-up dynamic programming (DP) approach of plan enumeration by treating ranking predicates as another dimension of enumeration in addition to boolean predicates, based on the insight that ranking (order) relationship is another logical property, parallel to membership, as explained in Section 2.1. In a ranking query plan, the predicate set 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 expression, the optimizer must be able to select the most efficient plan among various plans that schedule and interleave μ operators differently, just like it must be able to select the best join order.

Cost Estimator Using Sampling: A key ingredient of the accuracy of a cost model is cardinality estimation of intermediate results. Cardinality estimation is much more difficult in ranking query plans than in traditional plans. In conventional query plans, the input size of an operator is independent from the operator itself and depends only on the input subplans. 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. This imposes a big challenge to System-R style optimizers that build subplans in bottom-up fashion, because the input sizes consumed by operators depend on the location of that subplan in the complete plan, which is unavailable during enumeration. We propose a sampling-based cardinality estimation method to address this challenge. Further details can be found in [3].

3 Demonstration

The RankSQL system is implemented in PostgreSQL 7.4.3. In addition to the core query engine, we build a suite of useful tools for system builders to explore the process of query optimization and execution in our system. We also develop a Java GUI with JDBC connection to demonstrate

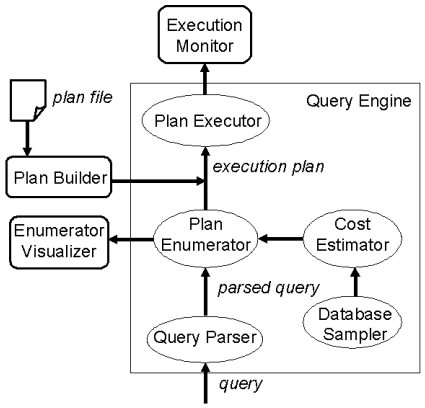


Figure 3: The architecture of RankSQL.

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

Figure 4: A sample plan file.

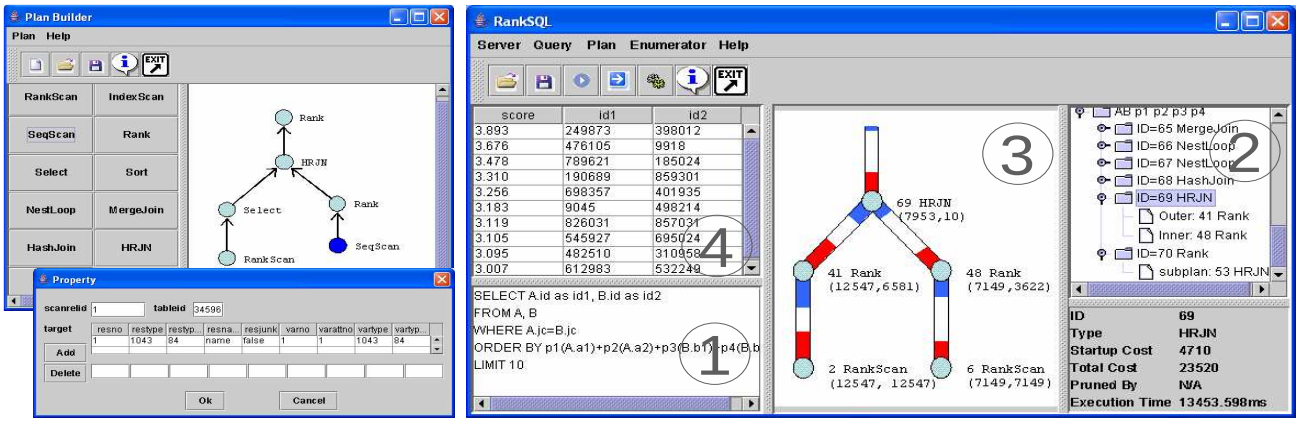
the system and tools, showing how RankSQL can improve query performance by orders of magnitude. We first introduce the tools in Section 3.1, and then we describe the demonstration scenario in Section 3.2.

3.1 The RankSQL Tools

Figure 3 illustrates the architecture of RankSQL tools, including *Plan Builder*, *Enumerator Visualizer*, and *Execution Monitor*. The tools are convenient for system builders to understand, to debug, and to improve various components of a query engine. Therefore, their usefulness goes beyond demonstrating and visualizing our prototype. We briefly introduce theses new tools below.

The Plan Builder directly constructs a physical execution plan (bypassing the query optimizer) from a plan file in XML format. For example, Figure 4 shows a simple plan file that scans a table with ID 34596 in the catalog and returns tuples with the projected attribute *name*. The system builder provides a drag-and-drop GUI for editing the plan file visually, as shown in Figure 5(a). With this tool, system developers can experiment with plans that are not chosen by the query optimizer. In addition, it is useful in evaluating new query algorithms that are not yet incorporated in the optimizer's enumerator or cost-model.

The Enumerator Visualizer displays the information (the estimated cost, better plans with the same logical property, *etc.*) of every enumerated sub-plans including the pruned ones. With it, system developers can identify the reason of pruning a plan and figure out why and how the final execution plan is chosen. Moreover, this tool enables users to control the enumeration procedure by stepping through the enumeration level-by-level (*e.g.*, sub-plans for



(a) The Plan Builder.

(b) The RankSQL GUI.

Figure 5: Demonstration scenario.

n table join, then sub-plans for $n+1$ table join) and manually changing the optimizer's choice by resurrecting some pruned sub-plan and pruning the chosen sub-plan.

The Execution Monitor visualizes the execution of a physical plan, by showing with animation the flow of tuples, the size of intermediate results and internal data structures for each operator, and the execution time. It also displays the estimated cardinality information of every operators in a sub-plan.

3.2 Demonstration Scenario

The screenshot of our GUI is in Figure 5(b). It divides the window into query (area 1), enumerator (area 2) , plan (area 3), and results (area 4). We will demonstrate RankSQL with various data sets and queries. We will show that the new ranking query plans improve the performances by orders of magnitude when compared with traditional plans. In this section, we show one simple example of answering the following *top-k* query, which joins two tables (each with 1 million tuples) on jc and retrieves the top 10 results according to the summation score of 4 ranking predicates.

```
SELECT A.id as id1, B.id as id2
FROM A, B
WHERE A.jc=B.jc
ORDER BY p1(A.a1)+p2(A.a2)+p3(B.b1)+p4(B.b2)
LIMIT 10
```

The scenario of the demonstration is as follows:

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

(2) The user starts the two-dimensional enumerator and the enumerator visualizer displays the enumerated plans including the final execution plan in area 2. For example, for the logical property of *"AB p1 p2 p3 p4"* (meaning A and B are joined and $p1$ to $p4$ are evaluated), several physical plans are enumerated and shown under the node *"AB p1 p2 p3 p4"*. The one with ID 69 is chosen as the execution plan. The input nodes are shown under each plan node. Clicking on an input node (*e.g.*, *"41 Rank"*) moves the focus to the corresponding plan node (with ID 41 in this case).

The user can optionally step-execute the enumerator. At an intermediate level, the user inspects the enumerated plans and manually changes the optimizer's choices. This process goes on until the final execution plan is chosen.

(3) Summary information about the selected (sub-)plan in area 2 is shown on the bottom right panel, including the ID, type, estimated total and startup cost, which other plan prunes this plan, and real execution time. In addition, a plan tree is shown in area 3 to illustrate its structure. For example, the root of plan 69 is a HRJN node with two inputs, each of which is a Rank node above RankScan.

(4) The user executes the plan tree in area 3, and the execution monitor animates the execution process. A red bar (closer to the node itself) and a blue bar (closer to its upper node) are shown on the outgoing edge of each node. The length of the red bar indicates how many tuples are produced by the node, and the length of the blue bar indicates how many of the produced tuples are output to its upper operator (thus their length difference illustrates the number of tuples that are currently in the output buffer of the node). For example, the root operator of plan 69 outputs 10 tuples when execution finishes and produces 7953 joined tuples for producing these 10 tuples.

(5) The user can also use the plan builder to create or load a plan file, upon which the execution plan is directly constructed without the optimizer. The resulting plan is shown and executed in area 3 as described above.

(6) A table of the query results is shown in area 4.

Acknowledgements: We thank Jonathan Bryak for his participation in building the enumerator and the plan builder.

References

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

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

[3] C. Li, K. C.-C. Chang, I. F. Ilyas, and S. Song. RankSQL: Query algebra and optimization for relational top-k queries. In *SIGMOD*, pages 131–142, 2005.