The Rank Join Problem

Neoklis Polyzotis

University of California - Santa Cruz

Abstract. In the rank join problem, we are given a set of relations and a scoring function, and the goal is to return the K join results with the highest scores. It is often the case in practice that the inputs may be accessed in ranked order and the scoring function is monotonic. These conditions allow for efficient algorithms that solve the rank join problem without reading all of the input. In this chapter, we review recent efforts in the development and analysis of such rank join algorithms. First, we present some theoretical results that state the inherent complexity of the rank join problem and essentially reveal that any rank join algorithm has to trade off between I/O efficiency and computational efficiency. We then review a specific rank join algorithm that adjusts this trade-off at runtime, depending on the data and the scoring function, in order to strike a balance between I/O overhead and computation.

1 Background and Basic Definitions

A relational ranking query (or a top-K join query) specifies a scoring function over the results of a join and returns the K tuples with the highest scores. As an example, the following query (written in an SQL-like language) retrieves the top 10 hotels and restaurants located in the same city, giving priority to the cheap hotels and the best restaurants with live music.

SELECT *h.name*, *r.name*
FROM *Hotel h*, *Restaurant r*
WHERE *h.city* = *r.city*
RANK BY $0.4/h.price + 0.4 * r.rating + 0.2 * r.hasMusic$
LIMIT 10

Ranking queries have become increasingly popular in many application domains, from multimedia retrieval [2] to uncertain databases [1], as they allow a user to focus on the most relevant query results.

Rank join evaluation, i.e., computing the top K results of a relational join according to a specific scoring function, form an integral component of ranking query processing. Several recent studies have considered specialized rank join algorithms [1,2,3,4,5,7,8,9], the integration of such operators in the query optimizer [6], and the computation of statistics for query optimization [10]. In what follows, we provide a formal definition of rank joins and some necessary background in order to discuss the problem further.

Consider a natural join of relations $R_1, \ldots, R_n$ where each $\tau_i \in R_i$ is composed of *named attributes* and *base scores*. The base scores are denoted as a vector

$\mathbf{b}(\tau_i) \in [0,1]^{e_i}$ for some $e_i \geq 0$, and signify the importance of the tuple according to criteria specified by the ranking query. Returning to the previous example, we observe that *Restaurant* has two base scores, corresponding to the rating and the music event respectively. Base scores are aggregated using a *scoring function* $\mathcal{S}$ that computes the score of a join result τ as $\mathcal{S}(\mathbf{b}(\tau))$. We may also use $\mathcal{S}(\tau)$ as a shorthand for the score of τ. Following common practice, we assume that $\mathcal{S}$ is monotonic, i.e., $\mathcal{S}(x_1,\ldots,x_e) \leq \mathcal{S}(y_1,\ldots,y_e)$ if $x_i \leq y_i$ for all i.

Let τ be a join result such that $\tau = \tau' \bowtie \rho$ for some intermediate results τ' and ρ. We define $\overline{\mathcal{S}}(\tau')$ to be the value of $\mathcal{S}$ using the base scores of τ', and substituting 1 for any that are missing. The monotonicity of $\mathcal{S}$ implies that $\mathcal{S}(\tau) \leq \overline{\mathcal{S}}(\tau')$ since each base score of ρ is at most 1. Thus we call $\overline{\mathcal{S}}(\tau')$ the *score bound* of τ', since it is an upper bound on the scores of join results derived from τ'.

The rank join problem can now be stated as follows. We are given relations $R_1,\ldots,R_n$ and a monotonic scoring function $\mathcal{S}$, such that each relation is accessed sequentially in decreasing order of $\overline{\mathcal{S}}$, and the goal is to compute the K results of $R_1 \bowtie \ldots \bowtie R_n$ with the highest score ($1 \leq K \leq |R_1 \bowtie \ldots \bowtie R_n|$). We can efficiently implement the particular access model for several scoring functions that are frequently used in practice, by relying on commonly available access methods such as B-trees. In what follows, we use $I = (R_1,\ldots,R_n,\mathcal{S},K)$ to denote an instance of the rank join problem.

The previous definition requires that at least K join results exist, which guarantees that it is possible to fulfill a request for the top K results. In addition, note that the solution to an instance of the problem may not be unique, due to the existence of ties in the computed score values. However, the *terminal score*, that is, the score of the K-th result, is uniquely determined for a given instance, and is denoted as $\mathcal{S}^{\text{term}}$.

Given an algorithm A that solves the rank join problem, we use $cost(A,I)$ to denote the cost that A incurs on a specific problem instance I. A commonly used cost metric is based on the idea of *depth*. The depth on an input relation R_i is the number of tuples read sequentially from R_i before returning a solution. We use $depth(A,I,i)$ to denote this depth, and define $sumDepths(A,I)$ as the sum of depths on all inputs. Clearly, *sumDepths* is an interesting cost metric as it indicates the amount of I/O performed by an algorithm.

2 Analysis of Rank Join Algorithms

Several recent studies have explored deterministic algorithms to solve the rank join problem [1,2,3,4,5,7,8,9]. In this section, we review the main theoretical results in the complexity and properties of these algorithms. The review is based on the analysis presented in [9].

We begin by stating two desirable optimality properties for a rank join algorithm. Given a class of algorithms $\mathcal{B}$, a class of problem instances $\mathcal{J}$, and a cost metric *cost*, we say that a rank join algorithm $A \in \mathcal{B}$ is *optimal* if $cost(A,I) \leq cost(A',I)$ for all rank join algorithms $A' \in \mathcal{B}$ and problem instances $I \in \mathcal{J}$. An optimal algorithm may not be feasible in specific settings, which leads us to a relaxed form of optimality known as instance optimality. We say that A is *instance-optimal* if there exist constants c_1 and c_2 such that $cost(A, I) \leq c_1 \cdot cost(A', I) + c_2$ for all $A' \in \mathcal{B}$ and $I \in \mathcal{J}$. The constant c_1 is called the *optimality ratio.*

```
Algorithm template PBRJ($R_1, \ldots, R_n, \mathcal{S}, K$)
Template parameters: pulling strategy $P$; bounding scheme $B$
Input: relations $R_1, \ldots, R_n$; scoring function $\mathcal{S}$; result size $K$
Output: set of $K$ join results with highest score
Data structures: input buffers $HR_1, \ldots, HR_n$; output buffer $O$
1.  $t \leftarrow \infty$
2.  while $|O| < K$ OR $\min_{\omega \in O} \mathcal{S}(\omega) < t$ do
3.      $i \leftarrow P.chooseInput()$
4.      $\rho_i \leftarrow$ next unseen tuple of $R_i$
5.      $R \leftarrow HR_1 \bowtie \ldots HR_{i-1} \bowtie \{\rho_i\} \bowtie HR_{i+1} \bowtie \ldots \bowtie HR_n$
6.      Add each member of $R$ to $O$, retaining only the top $K$ tuples
7.      Add $\rho_i$ to $HR_i$
8.      $t \leftarrow B.updateBound(\rho_i)$
9.  end while
10. return $O$
```

Fig. 1. PBRJ Template

Given these two properties, we can ask whether there exist (instance-)optimal algorithms within a specific family $\mathcal{B}$. To effectively perform this analysis for several possible classes, we introduce the *Pull-Bound Rank Join* algorithm template that can express any deterministic rank join algorithm. The PBRJ template, shown in Figure 1, is instantiated by a deterministic *pulling strategy* P and a deterministic *bounding scheme* B. On each loop iteration of PBRJ, the pulling strategy P chooses a relation R_i to read, and the new tuple ρ_i is stored in an input buffer HR_i (typically a hash table). New join results are generated by joining ρ_i with the tuples in the other input buffers HR_j for $j \neq i$. The generated results are pushed to an output buffer O that holds the top K results seen so far. After each tuple is processed, it is given to the bounding scheme B via the method *updateBound*, which returns a new upper bound on the score of unseen join results. The results are returned when the K-th buffered score is at least as large as the bound t provided by the bounding scheme, since this indicates that the buffered results cannot be improved by reading more tuples.

It is straightforward to see that PBRJ is correct if we require that P returns the index of an un-exhausted relation, and that B returns a correct upper bound on the scores of join results that use at least one unread tuple. Furthermore, PBRJ can model any deterministic rank join algorithm by an appropriate choice of P and B, and hence it makes a convenient vehicle for the analysis of rank join algorithms.

The analysis in [9] considered two choices for the bounding scheme B, namely, the *corner bound*, which is used in the HRJN [5] family of rank join algorithms, and the *feasible region bound*, which was introduced in [9]. There were also two choices for P, namely, *round-robin* and *corner-bound adaptive*. The latter is specific to the corner bound and prioritizes access to the inputs based on the information maintained in this specific bounding scheme. The main results of the analysis can be summarized as follows:

– Within the family of PBRJ instantiations with the corner bound, both round-robin and adaptive pulling yield an instance-optimal algorithm. Moreover, the adaptive strategy becomes optimal under the absence of a specific type of score-value ties related to $\mathcal{S}^{\text{term}}$.
– Within the same family, corner-bound adaptive is always no worse than round-robin in terms of the per-input depth metric.
– No PBRJ instantiation that uses the corner-bound is instance optimal within the extended family of all PBRJ instantiations.
– The instantiation of PBRJ with the feasible region bound and the round-robin strategy yields an instance optimal rank join algorithm within the family of all deterministic rank join algorithms.

The last two results indicate that the corner bound may yield an unpredictably high *sumDepths* metric, whereas the feasible region bound enables a property of robustness (again, with respect to *sumDepths*). The basic difference between the two bounding schemes is that only the feasible region bound is *tight*, i.e., it computes a score value that can be actually achieved by the unseen data, which in turn allows PBRJ to stop as early as possible without committing mistakes. However, as shown in [9], robustness in terms of *sumDepths* comes at a price in terms of computational complexity. Essentially, computing a tight bound is provably hard–the corresponding decision problem is NP-Complete. In other words, the rank join problem exhibits an inherent trade-off between computational and I/O efficiency.

3 The Trade-Off between I/O Robustness and Computational Efficiency

The previous theoretical results raise an interesting question: Can we design a rank join algorithm that explores the trade-off between I/O robustness and computational efficiency? In this section, we review some initial developments in this direction, based on the results presented in [4].

We first repeat the formalization of I/O robustness as instance optimality with respect to the *sumDepths* metric (see previous section). We say that a rank join algorithm A is robust, if there exist constants c_1 and c_2 such that $sumDepths(A, I) \leq c_1 \cdot sumDepths(A', I) + c_2$ for any other algorithm A' and rank join instance I. As a starting point in our exploration of robust and efficient rank join algorithms, we can examine the actual performance of PBRJ using the feasible region bound. The experimental study presented in [4] shows that PBRJ performs very badly in terms of total execution time, even though the algorithm does less I/O due to instance optimality. There are two reasons for this overall inefficiency: the costly computation of the feasible region bound, and the "blind" access to inputs from the round-robin pulling strategy. Therefore, one possible direction is to optimize PBRJ along these two dimensions. Indeed, there exists an alternative (yet equivalent) definition of the feasible region bound that performs far fewer computations than the original variant. Moreover, the feasible region bound can be coupled with an adaptive pulling strategy, which is the counterpart of the corner-bound adaptive strategy, that allows PBRJ to prioritize access to its inputs. Overall, these modifications yield an instantiation of PBRJ that remains instance optimal, does fewer pulls than the round-robin strategy, and is more computationally efficient.

The previous improvements can reduce the overhead of the feasible region bound, but they cannot lift the complexity barrier of computing a tight bound. Hence, the next step is to consider alternative bounding schemes that can explore the trade-off between tightness and computational efficiency. The study proposes the *adaptive feasible region bound* that achieves precisely this property. The new bounding scheme follows the same logic as the original feasible region bound, but it performs its computations on quantized base scores. The level of quantization determines directly the space complexity of the feasible region bound, and in effect its time complexity. When the level of quantization is infinitely small, the bound is essentially the same as the feasible region bound. This means that it is provably tight but also potentially costly to compute. A coarser quantization implies lower overhead, but it also means that the bound is no longer tight. An interesting property is that the adaptive bound coincides with the corner bound at the coarsest quantization. By utilizing this hybrid bounding scheme, PBRJ can essentially regulate the overhead of bound computation, and can explore adaptively the space between instance-optimality and computational efficiency.

Experimental results demonstrate that the instantiation of PBRJ with the hybrid bounding scheme and the adaptive strategy really offers the best of both worlds. The algorithm's I/O performance is the same as an instance optimal rank join algorithm for the class of inputs where the tight bound is cheap to compute, and degrades gracefully in other cases. In overall efficiency, the new algorithm outperforms other instantiations of PBRJ that correspond to existing rank join algorithms, thus validating the idea of an adaptive trade-off between I/O robustness and computational complexity.

References

1. Agrawal, P., Widom, J.: Confidence-aware join algorithms. In: International Conference on Data Engineering, pp. 628–639 (2009)
2. Fagin, R.: Combining fuzzy information from multiple systems. J. Comput. Syst. Sci. 58(1), 83–99 (1999)
3. Fagin, R., Lotem, A., Naor, M.: Optimal aggregation algorithms for middleware. J. Comput. Syst. Sci. 66(4), 614–656 (2003)
4. Finger, J., Polyzotis, N.: Robust and efficient algorithms for rank join evaluation. In: Proceedings of SIGMOD (2009)
5. Ilyas, I.F., Aref, W.G., Elmagarmid, A.K.: Supporting top-k join queries in relational databases. International Journal on Very Large Databases (VLDBJ) 13(3), 207–221 (2004)

6. Li, C., Chang, K.C.C., Ilyas, I.F., Song, S.: RankSQL: query algebra and optimization for relational top-k queries. In: Proceedings of ACM SIGMOD, pp. 131–142 (2005)
7. Mamoulis, N., Yiu, M.L., Cheng, K.H., Cheung, D.W.: Efficient top-k aggregation of ranked inputs. ACM Transactions on Database Systems 32(3), 19 (2007)
8. Natsev, A., Chang, Y.C., Smith, J.R., Li, C.S., Vitter, J.S.: Supporting incremental join queries on ranked inputs. In: Proceedings of VLDB, pp. 281–290 (2001)
9. Schnaitter, K., Polyzotis, N.: Evaluating rank joins with optimal cost. In: Proceedings of the 27th Symposium on Principles of Database Systems, pp. 43–52 (2008)
10. Schnaitter, K., Spiegel, J., Polyzotis, N.: Depth estimation for ranking query optimization. In: Proceedings of VLDB, pp. 902–913 (2007)