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.
b(τ_i) ∈ [0, 1][^e_i] for some e_i ≥ 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 \( S \) that computes the score of a join result \( \tau \) as \( S(b(\tau)) \). We may also use \( S(\tau) \) as a shorthand for the score of \( \tau \). Following common practice, we assume that \( S \) is monotonic, i.e., \( S(x_1, \ldots, x_e) \leq S(y_1, \ldots, y_e) \) if \( x_i \leq y_i \) for all \( i \).

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

The rank join problem can now be stated as follows. We are given relations \( R_1, \ldots, R_n \) and a monotonic scoring function \( S \), such that each relation is accessed sequentially in decreasing order of \( \overline{S} \), 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 \( I = (R_1, \ldots, R_n, S, K) \) 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 \( S_{\text{term}} \).

Given an algorithm \( A \) that solves the rank join problem, we use \( \text{cost}(A, I) \) 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 \( \text{depth}(A, I, i) \) to denote this depth, and define \( \text{sumDepths}(A, I) \) as the sum of depths on all inputs. Clearly, \( \text{sumDepths} \) 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 \[12345789\]. 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 \( B \), a class of problem instances \( J \), and a cost metric \( \text{cost} \), we say that a rank join algorithm \( A \in B \) is optimal if \( \text{cost}(A, I) \leq \text{cost}(A', I) \) for all rank join algorithms \( A' \in B \) and problem instances \( I \in J \). An optimal algorithm may not be feasible in specific settings,