Supporting top-k join queries in relational databases

Richa Varshney, Ajay Gupta
Entry No. 2010mcs3517, 2010SIY7555

1 Motivation

1.1 Problem

Current join operators cannot generally benefit from orderings on their inputs to produce ordered join results. For example:

- Sort-merge joins (MGJN) only preserves order of joined column data.
- Nested-loop joins (NLJN) only orders on the outer relations are preserved through the join.
- Hash join (HSJN) doesn’t preserve order if hash tables do not fit in memory.

Consider the following example ranking query:

```
SELECT A.1, B.2
FROM A, B, C
WHERE A.1 = B.1 and B.2 = C.2
ORDER BY (0.3*A.1 + 0.7*B.2)
STOP AFTER 5;
```

where A, B, and C are three relations and A.1, B.1, B.2, and C.2 are attributes of these relations. The Stop After operator limits the output to the first five tuples. In Query, the only way to produce ranked results on the expression 0.3*A.1 + 0.7*B.2 is by using a sort operator on top of the join. Despite the fact that individual orders exist on A.1 and B.2, current join operators cannot make use of these individual orders in producing the join results ordered on the expression 0.3*A.1 + 0.7*B.2. Hence, the optimizer ignores these orders when evaluating the order by clause. Therefore, a sort operator is needed on top of the join.

Two major problems arise when processing the previous rank-join query using current join implementations:

1. sorting is an expensive operation that produces a total order on all the join results while the user is only interested in the first few tuples.

2. sorting is a blocking operator, and if the inputs are from external sources, the whole process may stall if one of the sources is blocked.
1.2 Related Work

A closely related problem is supporting top-k selection queries. In top-k selection queries, the goal is to apply a scoring function on multiple attributes of the same relation to select tuples ranked on their combined score. The problem is tackled in different contexts. In middleware environments,

- Fagin [1] and Fagin et al. [2]: Introduce the first efficient set of algorithms to answer ranking queries.
- TA algorithm [2]: Assume the availability of random access to object scores in any list besides the sorted access to each list.
- NRA algorithm [2]: Assumes only sorted access is available to individual lists.
- J* algorithm [3]: This algorithm join multiple ranked inputs to produce a global rank. Although J* shares the same goal of joining ranked inputs, this paper is more flexible in terms of join strategies, more general in using the available access capabilities, and more easily adopted by practical query processors.
- Importance-based join processing [4]: This is another related problem of producing important join results as early as possible. The importance value of a tuple is provided from some sorting attribute. Important tuples are given higher priority to proceed through the query evaluation plan.
- NRA-RJ [5]: This algorithm gives efficient rank join query operator that joins multiple ranked inputs based on a key-equality condition and cannot handle general join conditions.

2 Solutions Proposed

For solving the two aforementioned problems:

1. Result from decoupling the sorting (ranking) from the join operation
2. Losing the advantage of having already ranked inputs.

Authors propose new rank algorithm called **rank-join** which gives top-k join query results. To realize the new rank-join algorithm as a physical join operator, authors introduced two variants of ripple join, 1. **Hash ripple join(HRJN operator)**  2. **Block ripple join(Solution for Local ranking problem)**. They introduced a join operator which switch between the hash join and nested-loops join strategies. This operator is called **HRJN**. For solving the online input problem, they also introduced **adaptive join strategy**. For choosing the best join order, they also introduce another heuristic, **Rank-join order heuristic**. All of these steps are covered in following subsections.

2.1 Rank-Join algorithm

The algorithm takes m ranked inputs, a join condition, and a monotone combining ranking function f and the number of desired ranked join results k. The algorithm reports the top-k ranked join results in descending order of their combined score. Please refer to paper to see the actual steps of the algorithm. Authors also prove correctness of the algorithm and also proved space required is also bounded by this algorithm. Authors also proved that rank-join algorithm is instance optimal[2] which establishes a strong performance guarantee for the rank-join algorithm when compared with any other way to evaluate top-k queries.

2.2 HRJN

HRJN is built on idea of hash ripple join. The HRJN operator is initialized by specifying four parameters: two inputs, the join condition, and the combining function. Any of the two inputs or both of them can be another HRJN operator. The join condition is a general equality condition to evaluate valid join combinations. The combining function is a monotone function that computes a global score from the scores of each input. The algorithm maintains a threshold value that gives an upper bound of the score of all join combinations not yet seen. To compute the threshold, the algorithm remembers the two top scores and the two bottom scores (last scores seen) of its inputs. These are the variables Ltop, Rtop, Lbottom, and Rbottom, respectively.
Lbottom and Rbottom are continuously updated as they retrieve new tuples from the input relations. At any time during execution, the threshold upper-bound value (\( T \)) is computed as the maximum of \( f(L\text{top} , R\text{bottom} ) \) and \( f(L\text{bottom} , R\text{top} ) \).

HRJN implementation has two issues:

1. Buffer Problem
2. Local Ranking Problem

Please refer the paper to see the solution of above issues.

### 2.3 HRJN*: score-guided join strategy

The way rank-join algorithm schedules the next input to be polled can affect the operator response time significantly. One way is to switch between the two inputs at each step. However, this balanced strategy may not be the optimal one. One heuristic is to try to narrow the gap between the two terms in computing the threshold value. The threshold is computed as the maximum between two virtual scores: \( T1 = f(L\text{top} , R\text{bottom} ) \) and \( T2 = f(L\text{bottom} , R\text{top} ) \), where \( f \) is the ranking function. If \( T1 > T2 \), more inputs should be retrieved from \( R \) to reduce the value of \( T1 \) and hence the value of the threshold, leading to possibly faster reporting of ranked join results.

### 2.4 An adaptive join strategy

When inputs are from external sources, one of the inputs may stall for some time. An adaptive join algorithm makes use of the tuples retrieved from the other input to produce valid join results. This processing environment is common in applications that deal with ranking, e.g., a mediator over web-accessible sources and distributed multimedia repositories. In these variable environments, the join strategy of the rank-join operators may use input availability as a guide instead of the aforementioned score-guided strategy. If both inputs are available, the operator may choose the next input to process based on the retrieved scores. Otherwise, the available input is processed.

### 2.5 Rank-join order heuristic

In this heuristic, we find in which order we have to join the inputs so that we can get maximum similarity between the inputs to do less access for finding the top \( k \) results. Get a ranked sample of size \( S \) from \( L \) and \( R \). Calculate the similarity using footrule distance

\[
F(L,R) = \sum_{(i,j)} |L(i) - R(j)|
\]

where \((i,j)\) is a valid join result that joins object \( i \) from \( L \) with object \( j \) from \( R \). Using the ranked sample and the definition of the distance metric, \( F(L,R) \), authors lay out the rank-join order technique.

### 3 Evaluation highlights

The experiments are based on their research platform for a complete video database management system (VDBMS). The database tables have the schema \((\text{Id}, \text{JC}, \text{Score}, \text{Other Attributes})\). Each table is accessed through a sorted access plan, and tuples are retrieved in a descending order of the \text{Score} attribute. \text{JC} is the join column (not a key) having \( D \) distinct values. They use a simple ranking query that joins four tables on the non-key attribute \text{JC} and retrieves the join results ordered on a simple function. The function combines individual scores, which in this case is a weighted sum of the scores (\( w_i \) is the weight associated with input \( i \)). Only the top \( k \) results are retrieved by the query. The following is a SQL-like form of the query:

```sql
SELECT T1.id, T2.id, T3.id, T4.id
FROM T1, T2, T3, T4
WHERE T1.JC=T2.JC and
T2.JC=T3.JC and
T3.JC=T4.JC
ORDER BY w1 *T1.Score + w2 *T2.Score +
w3 *T3.Score + w4 *T4.Score
STOP AFTER k;
```
3.1 Changing the number of required answers

In this experiment, they vary the number of required answers, $k$, from 5 to 100 while fixing the join selectivity to 0.2% and compares the total time to evaluate the query, the number of accessed disk pages and the number of maintained buffer space.

- HRJN and HRJN* show a faster execution by an order of magnitude for large values of $k$. The high CPU complexity of the J* algorithm is due to the fact that it retrieves one join combination in each step.
- In terms of the number of pages retrieved, J* and HRJN* achieve better performance because retrieving a new tuple is guided by the score of the inputs, which makes both algorithms retrieve only the tuples that causes a significant decrease in the threshold value and hence less I/O.
- In terms of the number of maintained buffer space, HRJN and J* have low space overhead because they use the buffer only for ranking the join combinations, while J* maintains all the retrieved tuples in its buffer.

3.2 Changing the join selectivity

In this experiment, they fix the value of $k$ at 50 and vary the join selectivity from 0.12 to 2% and compares the total time to report 50 ranked results, the number of accessed disk pages and the extra space overhead.

- For all selectivity, HRJN* shows the best performance.
- For high selectivity values, J* has a better performance than HRJN.
- For low selectivity values, HRJN performs better.

3.3 The effect of pipelining

In this experiment, they evaluate the scalability of the rank-join operators. They vary the number of join inputs, $m$, from 3 to 6 and fix $k = 50$ and the join selectivity to 0.2%.

- HRJN and HRJN* show much better scalability than that of J* by orders of magnitude. The CPU complexity of J* increases significantly as $m$ increases.
- J* and HRJN* show better performance in terms of the number of accessed pages compare to HRJN because of the score-guided strategy they use.
- HRJN* is the most scalable in terms of the space overhead.

4 Discussion

Following points were discussed in the class during the presentation.

- What effects come to the algorithm if we normalized the score? In this paper authors didn’t mention explicitly whether scores are normalized or not. Ideally authors should have mention score details.
- In the paper [6] authors proposed a probabilistic method to find the top-k results from a single relation. Can we apply same kind of approach for finding the top-k join queries and whether it will work or not?
- In this paper authors didn’t disclosed any details about the dataset. Authors could have used standard publicly available dataset so that interested researchers can reproduce the experiments. They have used various parameters in various experiments like k(no. of results), join selectivity etc but they didn’t provide any details like number of relations in database, size of attributes of relations. These are the weakest aspect in this paper.
- In this paper authors didn’t describe how threshold will be calculated. Its clear that threshold will be upper bound of the scores of any join combination not seen so far but how its calculated not clear in the paper.
• In the paper it is assumed that attributes included in the order clause are sorted or indexed. What will happen to threshold calculation if attributes are not sorted?

• In the paper order by clause includes set of attributes which are sorted and each attribute belongs to different relation. If more than one attribute belongs to a relation then how expression will be evaluated. One way is to break relation into multiple relation and do the calculation. Is there any better way to do this?

• In local ranking solution balancing factor P plays an important role. This paper doesn’t provide any detail to calculate value of P. Similarly in rank join order heuristic details of how top S objects are chosen is not provided.

• In the paper authors assumed that scoring function is monotonic. In case if scoring function is non-monotonic then how will algorithm change?

References


