Summary of Supporting Top-k Join Queries in Relational Databases

By S. Xiao Fernández Marín

1 Summary

The query processors can not handle ranking queries with a great profitably, especially if they have joined.

For avoiding that, the paper describes a way to achieve this using the rank-join algorithm. It ranks the results of the join operation while the latter is still executing. In that way, it is avoided computing the whole join results as most times since most of the times users only need the first k results of a join query. The following SQL query is a possible notation for expressing a top-k query.

`SELECT * FROM` $R_1, R_2, ..., R_m$
`WHERE join_condition(`$R_1, R_2, ..., R_m$`)`
`ORDER BY f(`$R_1.score, R_2.score, ..., R_m.score$`)`
`STOP AFTER` k`;`

The computation of only the top-k results is possible with the use of individual orderings of the input relations. The join operators like sort-merge join (MGJN), nested-loop joins (NLJN) or hash joins (HSJN), can not benefit from orderings on the inputs as they decouple the join.

The main problem when talking about processing the rank-join query using the notation of the query above is that sorting is an expensive operation. If you want only the top-k, you still need to compute the whole input. That leads the sorting into being a blocking operator as it must read the whole input.

This paper proposes a new **rank-join** algorithm that:
- Has pipelined operators based on *ripple join*, explained later.
- Produce the top-k results reducing the range of Cartesian space and combinations to evaluate.

On the one hand, we have the traditional joins, which are made to minimize the time and the **ripple joins**, which minimize the time until a k results are available. There are also different variants of the ripple joins, *block ripple join*, where the algorithm reads blocks of tuples and *Hash ripple join*, where it has a hash join in memory, which keeps the tuples. In the case, it exceeds the memory size, the hash ripple into a block ripple join.

On the other hand, we have the **rank join** (HRJN), where it is assumed that the inputs are ordered already on their scores. This algorithm produces results in rank order after the minimum score is lesser than the maximum score of the next to join result row.
In this way, the buffer that holds the results has a size-independent of the size of the inputs. It is independent of the join algorithm and, as the algorithm produces the results in different orders, the join used have to produce the results early (in k-rows).

One alternative to ways of realising the new rank-join algorithm as a physical join operator is the **hash rank join**, a variant of the ripple join that reads the inputs in descending order, can use pipelined as it is symmetric so that can report the results as fast as possible and keeping a priority queue. It stops whenever the k-best results have been produced.

The HRJN is composed by 3 methods. *Open()*: Initializes the the operator and prepares it for the internal state. *GetNext()*: Gets the next ranked join result. *Close()*: Ends the operator.

There is still a problem: As they are reading first left, then right, the base tables are not read evenly so they scale this for reading more or less the same number of rows in the immediate left and right children, for being more efficient and find the top-k faster. For this, they use heuristics

(HRJN*) and will switch between the hash join and nested-loops join strategies. The so-called local ranking problem.

This algorithm assumes the input is always sorted, and when it has indexes, it is possible to random access these inputs and reduces the upper-bound of the score of unseen join combinations,

The final evaluation is based on the algorithms HRJN and HRJN*, compared with another rank-join operator based on J*. It is done by changing some variables:

- Number of result rows:
  - Total time to evaluate the query: HRJN and HRJN* faster execution.
  - Number of accessed disk pages: J* and HRJN* better performance.
  - Number of maintained buffer space: HRJN and HRJN* have low space overhead.
- Join selectivity:
  - Total time to report 50 ranked results.
  - Number of accessed disk pages.
  - Extra space overhead.

  HRJN* best performance. J* better performance for high selectivity values. HRJN performs better for low selectivity values.
- Number of join inputs:
  - Effect of pipelining on the total query time: HRJN and HRJN* better scalability.
  - Number of pages: J* and HRJN* better performance.
  - Space overhead: HRJN* most salable.

They also take into account if it is a left-deep or pipelined bushy tree, as the latter does not suffer from the local ranking problem and has a faster termination.

2 Questions not answered by this text

In the final evaluation, there are no explanations about if it is used a basic or index variant of the algorithm HRJN. It is also not mentioned if they use bushy or left-deep trees.

3 What has changed since this was written

This paper has been an inspiration for creating other complex operators like "Compare"[1] or [2].

[1] T. Siddiqui, S. Chaudhuri, V Narasayya. COMPARE: Accelerating Groupwise Comparison in Relational Databases for Data Analytics (Extended Version).

[2] N. Tziavelis and D. Ajwani. Optimal Algorithms for Ranked Enumeration of Answers to Full Conjunctive Queries.