NRank: A Unified Platform Independent Approach for Top-K Algorithms

Martin Čech[1] and Jaroslav Pokorný[2]
[1]*Lekis s.r.o., Pražská 126, 256 01 Benešov, Czech Republic*
[2]*Faculty of Mathematics and Physics, Charles University, Malostranské nám. 25, Prague, Czech Republic*

Keywords: Top-K Query, Relational Database, Rank Join.

Abstract: Due to increasing capacity of storage devices and speed of computer networks during last years, it is still more required to sort and search data effectively. A query result containing thousands of rows from a relational database is usually useless and unreadable. In that situation, users may prefer to define constraints and sorting priorities in the query, and see only several top rows from the result. This paper deals with top-k queries problems, extension of relational algebra by new operators and their implementation in a database system. It focuses on optimization of operations join and sort. The work also includes implementation and comparison of some algorithms in standalone .NET library NRank.

1 INTRODUCTION

Results of relational queries containing more than hundreds of tuples are often not clearly arranged and hence not usable for a user. Therefore, it is suitable to sort these results according to a criterion and to offer to users only the best ones. This reminds the behaviour of search engines, where a user usually defines neither a way of sorting results (this is, after all, fairly heavy task), nor restriction on the result size. The paper is focused on such queries, whose criteria of sorting as well as the required result size are defined by a user. Such queries are overall denoted by top-k queries (we define them formally in Section 2).

Top-k queries need not be restricted only to relational databases. They can concern also an aggregation of several web services; each of them provides some data, whereas central application performs joins on the data, their filtering and sorting (Horničák et al., 2011). Principles of optimization are the same and, moreover, they can take into account speeds of particular services or the fact that data can be loaded in parallel (Abid and Tagliasacchi, 2011).

A good introduction to top-k queries and basic algorithms for their processing is in the works (Bruno et al., 2002), (Fagin et al., 2003). Solutions in context of relational databases can be found in (Ilias et al., 2003) and later in the survey (Ilias et al., 2008).

In principle, two approaches can occur in practice:

- *plug-in* – in which a top-k functionality is built on top of the database engine, and
- *native* – which directly manipulate the database engine by injecting new preference operators

We follow the former in this paper. We focus on so called *top-k join queries* (or shortly *rank joins*) in our work, i.e. queries doing join of two or more (ranked) relations with output ordered by values of a scoring function calculated from tuple scores of input relations. We use some algorithms described in (Ilias et al., 2004), (Ilias et al., 2008). There are other attempts to top-k joins, based on modification of so called "no random accesses" (NRA) algorithms (Mamoulis et al., 2007).

We developed a .NET library NRank enabling a simple formulation and effective evaluation of top-k queries. The library is primarily intended for querying over a relational database, but it can be used also for querying over other data structures.

Section 2 concerns a definition and properties of top-k queries. Section 3 is devoted to description of four representative top-k algorithms used in the NRank. NRank is a unified platform independent system for top-k algorithms. Section 4 describes its implementation. In Section 5 we compare these algorithms and in Section 6 we summarize achieved results.

2 TOP-K QUERIES

A *top-k query* is the 4-tuple (S, c, f, k), where $S = \{S_i\}$ is a set data sources (a data source can be an arbitrary set of objects), $c = \Pi\ S_i \rightarrow \{0, 1\}$ is a scoring condition, $f = \Pi\ S_i \rightarrow \mathbf{R}$ is a scoring function ($\mathbf{R}$ is the set of real numbers), and $k \in \mathbf{N}$ is the maximum result size (the required number of returned objects).

This is the most general form of top-k query that we can meet in most of top-k query models. It includes both top-k join queries and top-k selection. 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 (Ilyas et al., 2003).

The *result of a top-k query* over sources' instances S^* is then an ordered set $O \subseteq \Pi\ S_i^*$ such that:

$\forall t \in O$: $c(t) = 1$,
$\forall t_i, t_j \in O$: $i < j \Rightarrow f(t_i) \geq f(t_j)$,
$\forall t \in O, s \in \Pi\ S_i^* - O$: $c(s) = 1 \Rightarrow f(t) \geq f(s)$
$|O| = k$.

The first condition only restricts the condition on tuples in the result (that is equivalent to SQL clause `JOIN ON c` and/or `WHERE c′`). The second condition defines a set ordering according to the scoring function f. The third condition says that the result contains tuples with the highest possible score. The last condition restricts the result size to k tuples. Of course, the number of tuples in result can be less than k. Then it is possible to modify the definition to ensure the result set was as large as possible.

Top-k algorithms solve the top-k query without reading all of the input. Their optimality is often measured in the number of input objects from particular sources, denoted as a *depth*. In fact, it is the minimal depth into which the algorithm must access its inputs to report the result. Other criterion of optimality is a space complexity; the algorithms distinguish significantly mutually in the amount of data they need to be hold in memory.

Determining time complexity of these algorithms is more complicated. Asymptotic time complexity of these algorithms is usually the same. Because of the sort-based approach these algorithms have no other possibility than reading inputs in linear time. Algorithms then distinguish only in processing data in buffers, but from the view of asymptotic complexity they are mostly „equally fast“. Asymptotic complexity is therefore too weak tool for comparing the effectiveness of top-k algorithms.

The recent approaches to rank join problem focus mostly on instance optimal algorithms in terms of I/O cost. Then this cost is a quantity proportional to the overall number of fetched tuples. The notion of an *instance optimal algorithm* (originally introduced in (Fagin et al., 2003) expresses a relative optimality of an algorithm with respect to a class of algorithms A and a class of instances D. An algorithm b is instance optimal over A and D if for every $a \in A$ and every $d \in D$ we have $t(b, d) = O(t(a, d))$ (where $t(b, d)$ is the number of algorithm steps), in other words, there are two constants c_1 and c_2 such that $t(b, d) \leq c_1 {*} t(a) + c_2$ for every $a \in A$ and every $d \in D$. c_1 is denoted as the *optimality ratio*. When the algorithm complexity is determined, usually the worst or average case is studied only. However, while an algorithm is instance optimal, it is guaranteed, that it will be „reasonably fast“ in all defined cases. While the sets A and D are well defined, the instance optimality is a very strong criterion. For example, the Hash Rank Join algorithm (see Section 3) is an instance optimal algorithm.

An interesting finding is that most of instance optimal algorithms are in practice slower than other algorithms. Although they have a good asymptotic complexity, their speed is really low, more precisely lower, than the speed of some algorithms, which are not instance optimal. During determination of the algorithms complexity we often meet cases, where the algorithm has a good asymptotic complexity, but its real speed is low; top-k algorithms, which are instance optimal, have often the same asymptotic complexity, but their real speed is often higher. Therefore we are often driven by a quest to find a reasonable compromise between instance optimality, which guarantees a robustness, and real algorithm effectiveness. Some works, e.g. (Finger and Polyzotis, 2009), endeavour to find compromises between these extremes.

In the case of rank joins we use functions f aggregating (or combining) ranked inputs. We will consider only monotone aggregation functions to ensure some properties of algorithms followed. The main idea of the algorithms is sorting the inputs in descending order according to ranking and quest to provide the result as soon as possible.

3 TOP-K JOIN ALGORITHMS

We will consider four algorithms based on hashing and nested loops.

3.1 Hash Rank Join

The algorithm Hash Rank Join (Ilyas et al., 2003), (Ilyas et al., 2004) requires ranked data (in a descending order according to *f*) in both inputs *L* and *R* and join condition only of form `L.X = R.Y`. The scoring function for calculating the combined score is `L.score + R.score`.

The algorithm alternately reads both inputs and stores them into hash tables HL and HR, where the hash value is calculated from the value of attribute, which occurs in equality condition. After reading new values from an input, in the hash table of the second input there are looked up the tuples, which are joinable. The join results are inserted into the priority queue, where they are ordered by their combined score. Furthermore, the algorithm maintains the top and bottom scores of retrieved inputs L_{max}, L_{min}, R_{max}, R_{min}, respectively, from which it calculates so called *score threshold*

$$t = \max(f(L_{max}, R_{min}), f(L_{min}, R_{max}))$$

This is an upper-bound on the score of all join combinations not yet seen. The soundness of the estimation is trivial and follows from the monotonicity of *f*. Values L_{max} and R_{max} are initialized during reading the first tuple of *L* and *R*, respectively, values L_{max} and R_{max} change continuously with new tuples from L and *R* being fetched. All results, which have score greater than *t*, can be immediately reported. Obviously, the value of *t* is also computed from couples, which do not fulfil the join condition.

Algorithm Hash Rank Join:

```
Input: data sources L and R,
    aggregation function f:(L×R)->R,
    join condition c:(L×R) -> {0, 1} of
   form L.X = R.Y,
    result size k
Output: the first k tuples from L*cR,
    ordered in descending order
    according to ranking f.
Variables: hash tables HL and HR, the
    hash key (L.X or R.Y), L_min,
    L_max, R_min, R_max calculated
    from retrieved tuples, score
    threshold t, priority queue T
   1. HL = HR = Ø;
   2. L_min = R_min = ∞;
   3. L_max = R_max = -∞;
   4. while (L≠Ø nebo R≠Ø)
   5. {
         I := one of inputs L, R
      (inputs are selected alternately.
       The second input is denoted J,
      its hash table HJ);
6.     read the next i ∈ I;
7.     foreach j ∈ HJ[i]:
              T.Insert(join(i, j));
8.     I_min = i.Score;
9.     I_max = max(I_max, i.Score);
10.    update t value;
11.    while(T≠Ø and T[0].Score>=t)
12.   {report T[0] as output and
       remove it from T;
13.       if k tuples are reported
       then stop;
14.    }
15. }
// emptying out the queue
16. while(T≠Ø)
17. {
        report T[0] as output and
      remove it from T;
18.     if k tuples are reported
        then stop;
19. }
```

The algorithm must hold in memory all no reported results (in priority queue) and all retrieved data from both inputs (in the hash table). While the number of resulted tuples *k* is not known, the queue can grow without restrictions. Otherwise, while *k* is known, the queue can hold only *k* of best tuples. A compromise is possible – the queue can be restricted on a value greater than *k* (e.g., for *k* = 10 we restrict the queue to 100 objects). The user then gains top *k* tuples, but he/she can require additional tuples, up to maximal queue size. This ensures that the queue does not exceed the given size.

The basic variant of Hash Rank Join can be optimized by the input choice in particular steps of the algorithm. Since the basic variant alternates the both inputs mechanically, it becomes easily, that after reading the next object the score threshold *t* does not decrease. Minimization of *t* essentially influences a speed of reporting the results, i.e. the overall algorithm speed. During calculating *t* the maximum from two virtual values $t_1 = f(L_{max}, R_{min})$ and $t_2 = f(L_{min}, R_{max})$ is chosen. The threshold decreases only in case that in the next step the greater from values t_1 and t_2 decreases (if $t_1 > t_2$, then the right input is chosen, otherwise the left one) – in other words, it is not necessary to decrease the lesser from values t_1, t_2, the resulted *t* will be not changed. The optimization lies in strategy, in which the input is chosen from *L* and *R* in the way that the value *t* always decreases. This strategy is further denoted as *adaptive*.

However, this strategy of minimizing *t* is not always better than classical alternating the inputs (*round robin* strategy). There are inputs, where alternating inputs is more suitable, as it is mentioned in (Finger and Polyzotis, 2009).

3.2 General Rank Join

The algorithm Hash Rank Join can be generalized in the way that the join condition need not be necessarily only equality condition, but arbitrary. In such case algorithm will not store output into hash tables (moreover, there is no key to be hashed), but simply into a buffer (each input has its buffer). When a tuple is read from one input, all tuples in second buffer must be iterated and join condition must be checked at each step. This process is obviously slower than the Hash Rank Join, which does not check join condition at all (the hash table provides just the right tuples to join). The General Rank Join is more universal, but slower.

3.3 Nested Loop Rank Join

The algorithm (Ilyas et al., 2003), (Ilyas et al., 2004) is a variant of the classical nested loop join. One input (denote it L) is read consecutively. For each tuple $l \in L$, the algorithm iterates through all tuples from the second input (R), tries to join them with l and the results stores again into the priority queue. Similarly as in the general rank join, the threshold value t is maintained and updated with each reading a tuple from L. Resulted tuples in the priority queue, which have the resulted score greater than t, are immediately outputted.

In fact, the Nested Loop Rank Join is a variant of General Rank Join, but firstly all tuples from R are read and stored into a buffer and after that the tuples from L are read.

Input R does not have to be ordered, since all tuples are loaded and stored in a buffer, and thus $R_{\max}$ and $R_{\min}$ are known in advance and the threshold score can be easily estimated; in addition, the values $R_{\max}$ and $R_{\min}$ do not change during reading L. Tuples from L do not have to be maintained in memory; on the other hand, as all data from R are held in memory, this algorithm is suitable rather in case, where data from R are rather sparse, or when it is possible to access them directly (random access), so that it is not necessary to hold data from R in memory.

There is one more interesting optimization. The value $L_{\max}$ can be decreased in each step to the value $L_{\min}$. After reading a tuple $l \in L$ there are created all result tuples, where l can occur (tuple l is even forgotten in the next step, since it is not stored in the buffer). Therefore the value $L_{\max}$ can be decreased on the score of tuple l. In each step of the algorithm, $L_{\max} = L_{\min} = f(l)$. Then the value of t decreases faster, than using the Hash Rank Join. Consequently, this algorithm can be faster in some situations.

3.4 Hash Nested Loop Rank Join

The algorithm Nested Loop Rank Join can be optimized for such top-k queries, where the join condition is an equality condition. Similarly to the Hash Rank Join algorithm can store the retrieved data in hash tables. In this case a tedious throughput of the whole buffer drops out. The hash table of the input occurring in memory is created during initialization of the algorithm; the hash table of the second input is filled by sequel during reading particular tuples, in the same way as in Hash Rank Join.

4 IMPLEMENTATION

The library NRank is implemented in .NET/C# 4.0. Its main objective is to provide all the mentioned algorithms to users, while it stays independent on the data sources. Whereas most implementations focus on a specific algorithm and are closely linked to some DBMS, we have implemented generic versions of the algorithms, which accept any data collection as a data source.

This approach opens up many new possibilities, i.e. performing the algorithms on two completely independent web sources, combining data from different databases, etc. Inspired by C# query language LINQ, we also stressed simple usage. When performing a join query in LINQ, a user typically specifies the collections to be joined, the join condition, and the result function. In NRank, the calling convention is very similar and is extended only by ranking functions, so that algorithms are able to count tuple's rank.

While we cannot achieve such fast algorithms as if they were integrated into a database, we still obtain very interesting results. A big difference in using traditional LINQ queries and NRank queries is that whereas LINQ queries are translated to SQL (if called on database objects), NRank queries are performed with no translation and optimization. That means, when LINQ query is called, user typically does not care about what algorithm is used. When NRank query is called, user must specify the algorithm; this step has crucial impact on evaluation speed.

The library is accessible at address http://code. google.com/p/nrank/.

5 COMPARISON OF ALGORITHMS

The project includes a benchmark database containing tables `Town` (50.000 tuples), `Hotel` (1.000.000 tuples), `Club` (1.000.000 tuples), and `Room` (20.000.000 tuples). Experiments have been done on Microsoft SQL Server 2008 Express Edition, OS Windows 7 64bit, Intel ®Core™ 2 Duo, CPU 8400 2.26GHz, 4GB RAM. First, the algorithms evaluated a top-k binary join that can be expressed by the SQL query

```
SELECT T.IdTown, C.IdClub,
               T.Score + C.Score
FROM Town T JOIN Club C ON
          T.IdTown = C.IdTown
ORDER BY DESC T.Score + C.Score
STOP AFTER k
```

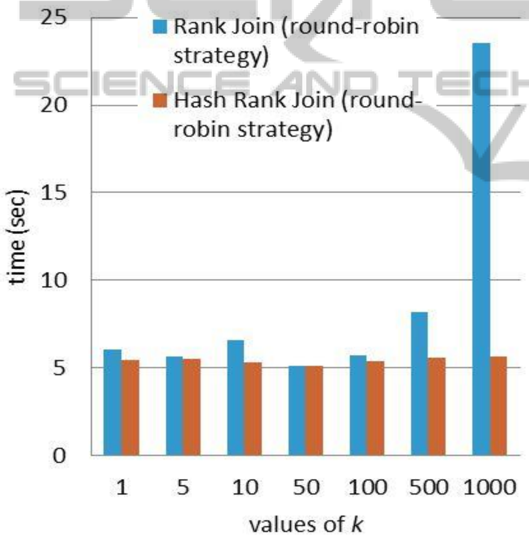

Figure 1: Comparison of General Rank Join and Hash Rank Join.

We varied the values of *k* in the sequence 1, 5, 10, 50, 100, 500, and 1000.

Experiments represented by Figures 1 and 2 show, that top-k algorithms exploiting hash tables for cashing input data are much faster than top-k algorithms exploiting only a simple buffer. The use of buffer even for small values of *k* significantly slows down the algorithm. The only difference between general and hash algorithms is in the join condition.

Our experiments also shown that concerning the General Rank Join algorithm, the strategy of input choice is crucial only for higher values of *k*. Time saving is subtle for small values of *k* since both algorithms read from their inputs only small amount of data. The Hash Rank Join almost does not depend on a strategy of input choice. The use of hashing brings a significant saving while the strategy in itself does. Due to the algorithms after reading an object from input need not look through a huge amount of data, both strategies of input choice are comparably fast. Both algorithms read order of magnitude the same amount of data independently on the strategy used. Thus, in use of Hash Rank Join, the strategy of input choice has a negligible influence.

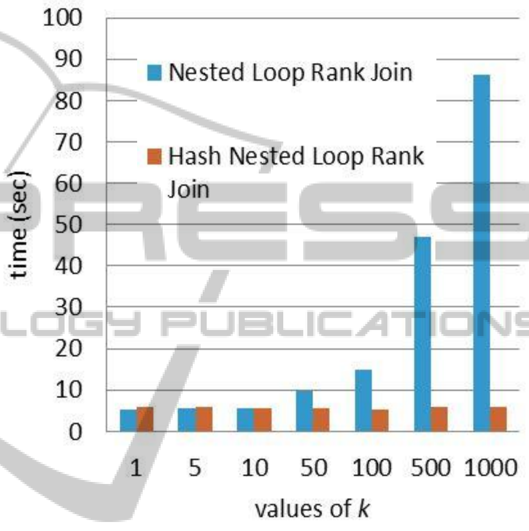

Figure 2: Comparison of Nested Loop Rank Join and Hash Nested Loop Rank Join.

We also tested the algorithms evaluating top-k join of three tables (see Figure 3) with the query

```
SELECT T.IdTown, H.IdHotel, C.IdClub
FROM Town T JOIN Club C ON
      T.IdTown = C.IdTown
JOIN Hotel H ON T.IdTown = H.IdTown
ORDER BY DESC
      T.Score + C.Score + H.Score
STOP AFTER k
```

For small *k* values the algorithm Nested Loop Rank Join is the fastest. Other algorithms are also usable, nevertheless, e.g., Nested Loop Rank Join has higher demands on the memory (keeps all data from one input in memory). Algorithms General Rank Join and Nested Loop Rank Join, i.e. algorithms, which do not use hash tables, are for larger values of *k* not usable.

In general, the compared top-k algorithms are beneficial in situations with joins of more tables (or other data sources) and for smaller values of *k*. While the join condition is formulated as equality, hash algorithms are fast also for large values of *k*.

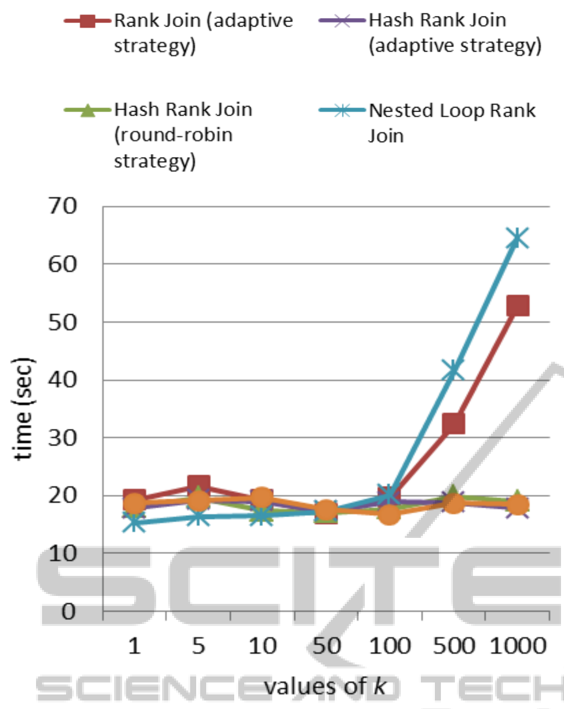

Figure 3: Comparison of all algorithms for joining three tables.

6 CONCLUSIONS

Including of top-k algorithms into a real relational environment is not usual by that time. For exclusion see, e.g., (Li, Chang, Ilyas, and Song, 2005) extending relational algebra and query optimization. Other examples are prototype implementations in PostgreSQL (Khalefa et al., 2011), (Kini and Naughton, 2007).

We have made comparison of several algorithms implementing rank join operator. Our results confirmed that huge processing time can be saved using optimal algorithm and appropriate source-choosing strategy. In simple join conditions it is possible to boost the algorithms with use of a hash table, which brings significant improvement, especially for higher values of *k*.

We have developed the .NET library NRank, which implements these algorithms and is ready to be used in a real-world application. For a future work, a rank-aware optimization framework would be beneficial enabling to use data statistics stored in a usual RDBMS. The library can be used practically for processing arbitrary data.

ACKNOWLEDGEMENTS

This research has been partially supported by the grant GACR No. P202/10/0761.

REFERENCES

Abid, A. and Tagliasacchi, M. (2011). Parallel Data Access for Multiway Rank Joins. In *ICWE 2011*, S. Auer, O. Diaz, and G.A. Papadopoulos (Eds.), LNCS 6757, 44–58.

Bruno, N., Chaudhuri, S., and Gravano, L. (2002). Top-k Scoring Queries over Databases: Mapping strategies and Performance Evaluation. *ACM Transactions on Database Systems*, Vol. 27, No. 2, 153-187.

Fagin, R., Lotem, A. and Naor, M. (2003). Optimal aggregation algorithms for middleware. *Journal of Computer and System Sciences* 66, 614–656.

Finger, J. and Polyzotis, N. (2009). Robust and Efficient Algorithms for Rank Join Evaluation. In *ACM SIGMOD International Conference on Management of Data*, Providence, Rhode Island, USA, June 29 - July 2, 415-428.

Horničák, E., Ondreička, M., Pokorný, J., Vojtáš, P. (2011). Multi-user Searching of Top-k Objects with Data on Remote Servers. In: *ADBIS 2011 – Research Communications*, J. Eder, M. Bielikova, and A Min Thoa (Eds.), Oesterreichische Comp. Gesellschaft, Vienna, Austria, Sep. 20-30, 200-211.

Ilyas, I. F., Aref, W. G. and Elmagarmid, A. K. (2004). Supporting Top-k Join Queries in Databases. *The VLDB Journal (2004)* 13: 207–221.

Ilyas, I. F., Shah, R., Aref, W. G., Vitter, J. S. and Elmagarmid, A. K. (2004). Rank-aware query optimization. In *ACM SIGMOD International Conference on Management of Data*, Paris, France, June 13-18, 203-214.

Ilyas, I. F., Beskales, G. and Soliman, M. A. (2008). A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys*, Vol. 40, No. 4, Article 11, 58 p.

Khalefa, M. E., Mokbel, M. F. and Levandoski, J. J. (2011). PrefJoin: An Efficient Preference-aware Join Operator. In *27th International Conference on Data Engineering (ICDE)*, IEEE, 995-1006.

Kini, A. and Naughton, J. F. (2007). Database Support for Weighted Match Joins. In: Proceedings of 19th International Conference on Scientific and Statistical Database Management (SSDBM 2007), IEEE, 10 p.

Li, Ch., Chang, K. Ch., Ilyas, I. F. and Song, S. (2005). RankSQL: Query Algebra and Optimization for Top-k Queries. In *ACM SIGMOD International Conference on Management of Data*, Baltimore, Maryland, USA, June 14-16, 131-142.

Mamoulis, N., Yiu, M. L., Cheng, K. H., Cheung, D. W. (2007). Efficient Top-k Aggregation of Ranked Inputs. *ACM Transactions on Database Systems*, Vol. 32, August, Article 19.