Optimizing Multiple Top-K Queries over Joins

Dirk Habich and Wolfgang Lehner
Dresden University of Technology
Database Technology Group
01062 Dresden, Germany
{dirk.habich, lehner}@inf.tu-dresden.de

Alexander Hinneburg
Martin-Luther-University of Halle/Wittenberg
Database and Data Mining Group
06099 Halle, Germany
hinneburg@informatik.uni-halle.de

Abstract

Advanced Data Mining applications require more and more support from relational database engines. Especially clustering applications in high dimensional features space demand a proper support of multiple Top-k queries in order to perform projected clustering. Although some research tackles to problem of optimizing restricted ranking (top-k) queries, there is no solution considering more than one single ranking criterion. This deficit - optimizing multiple Top-k queries over joins - is targeted by this paper from two perspectives. On the one hand, we propose a minimal but quite handy extension of SQL to express multiple top-k queries. On the other hand, we propose an optimized hash join strategy to efficiently execute this type of queries. Extensive experiments conducted in this context show the feasibility of our proposal.

1. Motivation

With the advent of data warehousing concepts, knowledge discovery in general became one of the most prominent database application areas. Many extensions were proposed to better support KDD and warehouse queries and optimize their execution. Before the SQL:1999/SQL:2003 standardization, a top-k query with a single rank could be written as SELECT statement containing an ORDER BY plus a limiting clause like STOP AFTER k ROWS [3] or FETCH FIRST k ROWS ONLY (DB2 dialect) to retrieve the top-k results. The standard introduced an alternative formulation with nested SELECT statements making use of OLAP functions like RANK in combination with the OVER clause.

The extension of the OVER()-clause allows the specification of a column-wise ordering, partitioning and windowing scheme. Positions are computed by three additional aggregation functions RANK(), DENSERANK(), and ROWNUMBER() differing in the semantics of breaking ties. Due to simplicity, we do not further consider ties and refer to the RANK()-operator. The limitation with respect to the first k rows must be indirectly specified in a surrounding query. The following example shows how to state one or more top-k queries within a single SQL query.

```
SELECT x.id, x.pos1, x.pos2
FROM (
    SELECT id,
           RANK() OVER (ORDER BY f1()) AS pos1,
           RANK() OVER (ORDER BY f2()) AS pos2
    FROM R INNER JOIN S ON ...
    WHERE ...) x
WHERE x.pos1 <= :k OR x.pos2 <= :k
```

Computing top-k queries using this SQL extension is based on the principle of ordering the underlying data set with regard to the (usually numeric) ranking criterion and returning only the first k values per column. In the above example, after joining tables R and S, two different ranks are determined for each tuple according to sort criteria functions $f_1()$ and $f_2()$. The restriction to the top-k tuples of both rankings is applied in an surrounding $select$ statement.

Example 1: The concept of multiple top-k queries naturally appears in several relevant data mining and information retrieval applications. Many information retrieval systems employ relevance feedback. The idea is that the system learns iteratively from the users rating of the presented results to improve the retrieval quality. For example the concept of Kim and Chung [12] extends the basic idea of query point movement. Instead of moving the query point based on user feedback towards an assumed ideal query point the extended concept of complex similarity queries allows a set of multiple query points $Q = \{q_1, \ldots, q_n\}$. The top-k result tuples can be defined by a new distance function, which requires that the result tuple is close to at least one of the query points in Q:

$$dist(x, Q) = \min_{q_i \in Q}{}_k\{dist(x, q_i)\},\ 1 \leq i \leq n$$

The distance function can be expressed as a SQL query using n top-k rankings, one for each query point. The combined results of the top-k queries are ordered according to

SID	PID	QUAN.	SALES
1	1	500	1200,00
1	2	300	100,50
...			
10	5	100	50,00

(a) fact table

SID	G	F	EU
1	0.2	0.3	0.2
2	0.1	0	0.05
3	0	0.2	0.1
...			

(b) weighting table

Figure 1. Example for multiple top-k queries over joins

the distance to their nearest query points and the k tuples with the smallest distance are returned.

Example 2: Another application scenario appears in data warehouse environments, where multiple top-k queries over joins are useful. Consider following example, where a fact table holds objects like products or shops and corresponding facts. These informations are either stored in the data warehouse or computed with SQL-statements. The table in figure 1(a) holds some facts for sold products (PID) in shops (SID), like quantity and sales. The objects can now be ranked according to the different facts with regard to weighting factors. Such factors represent the importance of the objects in different contexts and they are used to align raw data and to statistically correct samples. For example, typical weighting factors for shops are the market power with regard to geographic location, e.g. Germany, France and Europe. This weighting factors are typically stored in a dimension table (figure 1(b)). To rank the objects into multiple directions with regard to the multiple facts and multiple weighting factors a join between the fact table and the weighting table is necessary and the result have to be ordered according to multiple ranking functions. In this case, the parameters of the ranking functions come from both relations.

Our Contribution

In this paper we consider a class of ranking functions described by $f(g_1(R.A_1, R.A_2, \ldots), g_2(S.B_1, S.B_2, \ldots))$ where $f(\cdot, \cdot)$ is monotonic in its two input attributes. The functions $g_1()$ and $g_2()$ might be any functions taking inputs from tables R and S respectively. We also consider multiple ranking functions taking only inputs from one relation.

Moreover, it is worth mentioning that without applying very specific optimization strategies the top-k-computation is done by computing the ranks for *all* tuples requiring one sort for each individual ranking criterion. Finally, for applying specific optimization algorithms, the currently available top-k ranking formulations show the problem that the information about the top-k predicate is structurally very far from the ranking declaration implying very sophisticated query graph pattern recognition mechanisms to detect situations in which the query could be optimized.

To soften the two major problems - no direct support of top-k queries in the SQL formulations and no internal optimization algorithms for computing multiple top-k queries simultaneously, we propose the following concepts in this paper:

- First of all, we introduce a small SQL extension of ORDERING SETS to simplify the declaration of multiple rankings. Additionally, we inject LIMIT BY clauses in the ORDERINGS SETS as well as within the already existing OVER-clause.
- We discuss the limitations of existing rank optimizations in the presence of multiple ranks and give a potential extension of an early stop algorithm based on sort-merge joins.
- We finally propose a variation of the well-known hash-join algorithm which considers the presence of multiple top-k columns. This variation outperforms all other join strategies and can be easily integrated into existing database engines.

The rest of the paper is organized as follows: After gleaning related work in the following section, we present our SQL extension for computing multiple top-k queries in section 3. In section 4 we consider simple queries with multiple ranks, but without joins. Thereafter, in section 5, we describe an extension of an early stop algorithm and introduce our proposed extension of a hash-join method considering the existence of rankings. In section 6, we demonstrate the improved efficiency of our algorithms by describing the results of extensive experiments run on a prototypical implementation. The paper closes with a summary and conclusion.

2. Related Work

The goal behind top-k queries is to apply a scoring function on multiple attributes coming from one or multiple tables to select the best k tuples ranked by the function. So far, top-k queries with single ranking function have been intensively studied in the last years of database research. In particular it is worth mentioning that top-k queries have been considered in various contexts.

Carey and Kossmann [3] extended SQL's SELECT statement by a STOP AFTER clause, which limits the cardinality of a query result. The authors showed that this clause especially in combination with ORDER BY leads to significant better query plans and execution times. In the follow up paper [4] they presented extended implementation techniques for the STOP AFTER clause based on range partitioning. Donjerkovic and Ramakrishnan [7] proposed to map a top-k query to a range query with the range $[max, \kappa]$

where κ is chosen in a probabilistic way so that the range contains approximately k tuples. While this and the previous articles focused on orderings based on the column values of a single attribute themself, later papers take also ranking conditions based on multiple attributes, e.g. multi-dimensional metrics, into account. Chaudhuri et. al [1, 5] studied the use of multi-dimensional histograms to evaluate top-k queries with multi-attribute ranking conditions, namely metrics like Maximum, Euclidean and Manhattan. Here a top-k query is mapped to a multi-dimensional range query centered around a given query point. In their work they included experiments with ranking conditions based on up to four attributes. Cheng and Ling [6] proposed an approximative variant of the method of Chaudhuri et al. based on sampling, which scales better to high-dimensional data (up to 100 attribute) and has only a small loss of accuracy. Another approach was taken by Hristides et al. [9], who used multiple materialized views to efficiently answer top-k queries, with ranking conditions based on linear functions of the attributes of a relation. For a given ranking condition the best matching materialized view is selected to approximate the query answer.

None of the above described approaches considered top-k queries in conjunction with joins. Ilyas et. al [10, 11] proposed a new rank join operator producing *single* top-k results progressively during the join results. They consider a set of tables R_1 to R_n, where each tuple in R_i is associated with a local score. The global score is computed according to a function f combining the local scores of the individual tables. In section 5 we give a more detail description, because one of our algorithm extends the rank join approach to evaluate multiple top-k join results. In [11] Ilyas et. al present a rank-aware query optimization framework integrating the rank-join operators into relational query engines. The generation of a rank-aware query plan is done with a probabilistic model for estimating the input cardinality, and cost of the rank-join operators.

In a recent article Slivinskas, Jensen and Snodgrass [13] identified the optimization problem of database queries containing ORDER BY as a very important problem, which has been underestimated in the database community. They propose an extended algebra taking a single ORDER BY and top-k queries into account and give several formal transformation rules for such queries.

However, research so far on top-k queries considers only queries with a single ranking, i.e. sort and limitation condition.

3. SQL Extension for Top-k Queries

This section outlines minimal SQL extensions providing a new concept of computing multiple top-k queries within a single select statement. In a first step, we revise the current state of the art, demonstrate the problems in retrieving multiple top-k database entries and finally introduce the ORDERING SET() and LIMIT BY() concepts from a syntactical as well as a semantic point of view.

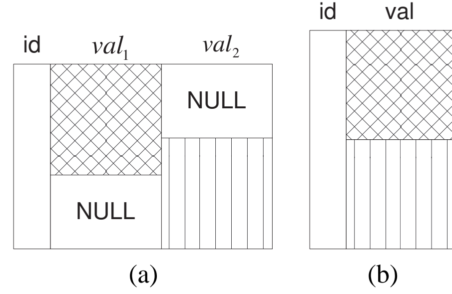


Figure 2. Schematic sketch of the possibilities for returning result of queries with multiple ranking conditions.

3.1. Multiple Top-k Queries

Conceptually there are two possible methods to return multiple top-k results without unnecessary information. Figure 2 illustrates the shape of the resulting tables.

The following query pattern basically extends the basic SQL pattern given in the motivation with the assumption that only the first k values of each sorting criterion have to be considered. This can be achieved by performing an $n-1$-ary outer join to compute the required result (figure 2a).

```
SELECT v1.id ,v1.pos1 ,COALESCE(v1.ind1 ,0) AS ind1 ,
                v2.pos2 ,COALESCE(v2.ind2 ,0) as ind2
FROM (
   (SELECT v1.id, v1.pos1, 1 AS ind1
    FROM (
        SELECT id ,
            RANK() OVER(ORDER BY f1()) AS pos1 ,
        FROM ...
        WHERE ...) u1
    WHERE u1.pos1 <=: k) v1(id, pos1)
    FULL OUTER JOIN
   (SELECT v2.id, v2.pos2, 1 AS ind2
    FROM (
        SELECT id ,
            RANK() OVER(ORDER BY f2()) AS pos2 ,
        FROM ...
        WHERE ...) u2
     WHERE u2.pos2 <=: k) v2(id, pos2)
     ON v1.id = v2.id )
```

Within this query template, each subquery locally computes the individual ranking results which are then 'concatenated' using a full outer join so that the individual ranks

larger than the given k are set to NULL. Additionally, an indicator function *COALESCE* is used to differentiate between natural NULL values and NULL values generated by full outer joining the individual top-k queries. In the worst case, this may yield an extremely sparse table where each tuple holds only one valid rank. So for computing n ranks of size k, we may yield a cardinality between k and $n \cdot k$.

Alternatively (figure 2b), the local results of the individual ranking values can be concatenated vertically by performing an union after computing the local ranks. For the running example, the corresponding query pattern might look like the following:

```
SELECT u1.id, '1' AS indicator, u1.pos1 AS pos
FROM (
    SELECT id,
           RANK() OVER (ORDER BY f1()) AS pos1,
    FROM ...
    WHERE ...) u1
WHERE u1.pos1 ≤ :k
UNION
SELECT u1.id, '2' AS indicator, u2.pos2 AS pos
FROM (
    SELECT id,
           RANK() OVER (ORDER BY f2()) AS pos2,
    FROM ...
    WHERE ...) u2
WHERE u2.pos2 ≤ :k
```

An additional indicator column denotes the local result set. This solution is perfect if there is almost no overlap in the result set implying that a single row appears only once within the top-k values with regard to a single ranking.

Comparing both alternatives from a query formulation and query optimization perspective leads to the following observation. The individual subqueries are computed locally and combined in a subsequent step, which is either a union or a full outer join so that applying sophisticated rank-operators eventually computing multiple top-k results becomes extremely difficult. Additionally, the query structures of both variants are inadequate to serve as language expressions because of the huge statements necessary to express the same pattern and repetitive computation of the (potentially complex) table expressions in the FROM clauses.

To put it into a nutshell, it is clear that SQL does not adequately support multiple orderings in combination with a limitation of the output stream either for vertically or horizontally constructed result sets. The query expressions are extremely voluminous. Additionally, it is extremely difficult for the rewrite system inside of the database engine to detect these query patterns and to apply specific optimization techniques.

3.2. The ORDERING SET-Operator

To weaken the problems of multiple orderings and limiting the output stream, we propose a much simpler language construct, namely ORDER BY ORDERING SET, which operates quite similar to the GROUPING SET-operator and therefore fits nicely into the set of SQL extensions.

The ORDERING SET()-operator (as an extension of the ORDER BY-clause) denotes that the same data is sorted according to multiple ordering criteria and may be seen quite similar to the construct of a GROUPING SET()-operator, which is an extension of the simple GROUP BY-clause. Additionally, the individual ordering criteria may be extended with a LIMIT BY-parameter to restrict the number of rows for the particular ordering criterion. The ORDERING SET()-operator delivers the tuples of a table in the order according to the actual ranking criterion. When all tuples of the table are returned or the limit is reached the next ranking is processed.

```
SELECT ..., f1(), f2()
FROM ...
WHERE ...
ORDER BY ORDERING SET(
     (f1() DESC LIMIT BY :k),
     (f2() DESC LIMIT BY :k) )
```

To illustrate the ORDERING SET()-operator, we refer to the example in figure 3, which shows how to compute the first 3 rows with the highest values in $f_1()$ and $f_2()$.

The result (right table) of the ORDERING SET()-operator, which ranks first according to column f_1 and then f_2 with the limit $k = 3$. The horizontal line in the right table indicates when the second ordering starts.

Like the normal ORDER BY expression, one single ORDERING SET expression can exhibit multiple sorting criteria including ASC and DESC annotations to distinguish between ascending and descending ordering. A single ORDERING SET expression is equivalent to a normal ORDER BY expression, e.g.

```
ORDER BY ORDERING SET( f1() ASC, f2() DESC)
≡
ORDER BY f1() ASC, f2() DESC
```

Although in the general case, the value for k may be different for each individual ranking, in many applications k

ID	f_1	f_2
1	4	11
2	3	22
3	2	33
4	1	44

⇒

ID	f_1	f_2
1	4	11
2	3	22
3	2	33
4	1	44
3	2	33
2	3	22

Figure 3. Example of a 2-ary ranking of size 3

ID	$f_1()$	$f_2()$
1	44	11
2	33	22
3	22	33
4	11	44

⇒

ID	ORDERING($f_1()$)	ORDERING($f_2()$ DESC)	ORDERING($f_1()$ DESC, $f_2()$)
4	1	0	0
3	1	0	0
4	0	1	0
3	0	1	0
1	0	0	1
2	0	0	1

Figure 4. Example for the ORDERING function

will be the same for all sorting criteria. In this case, we allow an alternative global limitation for the ORDERING SETS as a shortcut, e.g.

```
ORDER BY ORDERING SET( (f1() LIMIT BY :k),
                       (f2() LIMIT BY :k) )
≡
ORDER BY ORDERING SET( ( f1() ) ,( f2() ) )
                    LIMIT BY :k
```

Like the GROUPING()-function for the GROUPING SET-extension we introduce the ORDERING()-function, which indicates to which ordering set a tuple in the result set belongs to. The function returns 1 if the current row was sorted according to the given sorting criterion. Thus, ORDERING(x) returns 1 if the current row was sorted according to expression x. In case the ordering set is defined over multiple sorting criteria (e.g. $f_1()$ DESC, $f_2()$ ASC) the ORDERING()-function takes also a list of expressions.

To illustrate the semantics in more detail, we consider the following ORDERING SET()-clause returning the first three rows of each ranking:

```
SELECT ...,
       ORDERING(f1()),
       ORDERING(f2() DESC),
       ORDERING(f1() DESC, f2())
FROM ...
WHERE ...
ORDER BY ORDERING SET( (f1()), (f2() DESC),
    (f1() DESC, f2()) ) LIMIT BY 2;
```

Figure 4 illustrates the identification of the individual ordering set membership for each row.

In case the ranking criteria are compatible with each other, e.g. only one expression is used for each ordering set, the result table can be explicitly transformed into the schema shown in figure 2b with the help of the ORDERING()-functions by adding a CASE statement like the following:

```
SELECT ...,
       1 * ORDERING(f1()) +
       2 * ORDERING(f2()) AS indicator,
       CASE WHEN ORDERING(f1()) = 1 THEN f1()
            WHEN ORDERING(f2()) = 1 THEN f2()
       END AS value, ...
FROM ...
WHERE ...
ORDER BY ORDERING SET((f1()),(f2())) LIMIT BY :k
```

3.3. The LIMIT BY Over-Clause Extension

Similar to the relationship of GROUP BY (with GROUPING SETS()) and PARTITION BY in the context of the OVER()-clause, we extend the functionality of reporting functions by a local LIMIT BY clause resulting in multiple benefits. This implies that the restriction of the output data is now close to the RANK() function avoiding nested queries. The scenario above may now be specified without any nesting by the following expression:

```
SELECT ...,
       RANK() OVER(ORDER BY f1()
              LIMIT BY 10) AS pos1,
       RANK() OVER(ORDER BY f2()
              LIMIT BY 10) AS pos2,
       LIMIT() OVER(ORDER BY f1()
              LIMIT BY 10) AS indicator1,
       LIMIT() OVER(ORDER BY f2()
              LIMIT BY 10) AS indicator2
FROM ...
WHERE ...
```

From a local perspective of a single column, the values are sorted according to the given ORDER BY criterion. In a second step, the LIMIT BY-clause propagates the first k rows from the preceding sort operator to the following RANK() function. All subsequent values are replaced by NULL values indicating that they are not contributing to the overall result. As an indicator, the new LIMIT()-function returns a numeric 0 if the corresponding original value with regard to the given OVER()-clause is omitted and 1 if the original value is part of the aggregation process (in most cases applied to the RANK()-operator). The same semantics applies in the presence of an additional PARTITION BY-clause with an optional window specification. The limitation applies to each partition locally without affecting the succeeding window definition.

3.4. Summary

Supporting ordering in relational database systems has a long tradition to pre-process returning data for presentational use. With the advent of data warehouse and information retrieval applications limited orderings (i.e. ranks) and multiple orderings (according to different combinations of

the data space) are becoming tremendously important. We introduced a small and seamless SQL-extension dedicated to support these requirements. The ORDERING SETS and the LIMIT BY extension fits seamlessly into the SQL language and greatly enhances query capability by reducing the complexity of the query statement. The following two sections outline the implementation of a special operator for simple queries and join queries with multiple ranks.

4. Simple Queries with Multiple Ranks

This section introduces the MRANK()-operator supporting our new language concepts and details the underlying algorithm. In the presence of joins the MRANK()-operator will be combined with join algorithms as shown in the next section. Since the mechanism of locally computing ranks using the OVER-clause with the LIMIT BY-extension is similar to the global construct of ORDERING SETS(), we restrict the following discussion to the latter one.

Assume the underlying data is stored in a relation $R(tid, col_1, \ldots, col_m)$ and the ranking functions $F = \{f_1, \ldots, f_n\}$ order the objects in descending manner. When mapping the ORDERING SET()-operator to queries of forms like presented in section 3.1 the current implementations compute the query body individually, apply the specific ordering functions, return the first k rows, and concatenate the single tuple streams using an union operator thus forming the overall result stream. Figure 5a) illustrates this approach. Unfortunately, such implementations require the complete sort of the underlying data stream according to each ordering function f_i. Instead we propose a novel (logical) operator MRANK() with multiple corresponding (physical) operators, which can be directly exploited when parsing the query. Figure 5b) shows how the query plan changes.

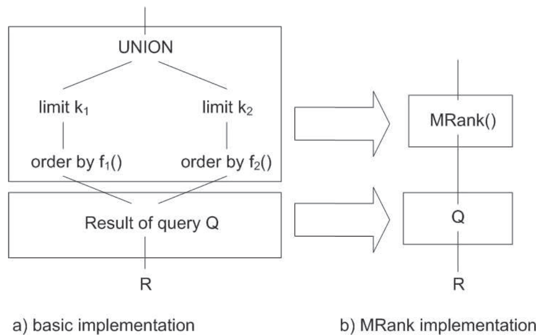


Figure 5. Applying MRank()-operator to compute limited ordering sets

The simplest implementation of the MRANK()-operator is based on data structures holding a minimal set of data in main memory. The algorithm holds a heap structure for every ranking criterion. It computes a single top-k query holding the $rowid$ and the values of the ordering function (Algorithm 1).

Algorithm 1 Main memory based algorithm

Require: Relation $R(id, col_1, \ldots, col_n)$
Query Q with ordering functions $F = \{f_1, \ldots, f_n\}$ and local limits $k_1, \ldots, k_n$
1: H:= set of heap data structures $h_i[id, f_i]$ of size k_i for all top-k operators $1 \leq i \leq n$
2: {Phase 1: Compute Multiple Orderings}
3: **for all** tuple $t \in R$ **do**
4: **for all** $i \in H$ **do**
5: **if** $h_i.count() < k_i$ **then**
6: $h_i.add(t[id], f_i(t))$
7: **else if** $f_i(t) > min(h_i)$ **then**
8: $h_i.remove(min(h_i))$
9: $h_i.add(t[id], f_i(t))$
10: **end if**
11: **end for**
12: **end for**
13: {Phase 2: Generate Output Stream}

In a first phase, each row of the incoming data stream is split according to the individual sorting criteria and added to the heap structure if the number of elements in the heap is still below the limit. If the heap size has already reached the limit, the current value c replaces the minimum value m stored at the top of the heap, if the $c > m$. When all heaps have been checked in that way the next tuple is processed.

In a second phase, the entries of the heaps are sorted, the final positions are assigned and the final rows are constructed and given to the next operator.

In case not enough main memory is available to hold all heaps, vertical or horizontal splitting and temp table techniques can be applied. However, due to the lack of space we do not present those technique in this paper .

Please note that the query body may include join-expressions. Still the basic MRANK()-operator can be applied, but the whole join has to be performed. In the next section we investigate how to push down the top-k criteria which may result in an early stop of the join.

5. Join Queries with Multiple Ranks

For single top-k queries with joins [10] proposes a rank join, which avoids to perform a full join of the underlying tables. First, we give an extension of the idea to deal with multiple top-k queries, which however requires the ranking functions to be linear combinations of the attributes.

S.id	$g_2(S)$				
D	10	20	18	18	17
A	14	**24**	22	22	21
C	15	25	**23**	23	22
B	20	30	28	**28**	27
	$g_1(R)$	10	8	8	7
	R.id	A	C	B	C

Figure 6. Evaluation of a single top-k join query

Second, we propose a new join method for multiple top-k queries, which is faster and is not limited to linear ranking functions.

5.1. Early Stop in Sort Merge Joins

The section of related work already mentioned the core principle of the approach given by [10]. From a join operator perspective, the main idea consists in stopping the flow of incoming tuples from the join partners, if future combinations of join tuples can never be within the set of the top-k tuples.

Given the join tables R and S and the ranking function $f(g_1(R), g_2(S))$ the key idea of the rank join is to process the tuples from R and S in decreasing order of $g_1(R)$ respectively $g_2(S)$. Let be x_{max} the first tuple (which yields the maximum of $g_1(R)$) and x the current tuple from R and y_{max} and y the analogous tuples from S then $T = max\{f(g_1(x_{max}), g_2(y)), f(g_1(x), g_2(y_{max}))\}$ is an upper bound for the ranks of unprocessed join combinations. If T is smaller than the smallest rank of the top-k tuples seen so far, than the join can be stopped early as no future tuple combination will be included in the top-k result.

Consider the following example with the ranking function $f(R, S) = g_1(R) + g_2(S)$, $g_1(R) = R.A_1 + R.A_2$ and $g_2(S) = S.B_1 + S.B_2$. The join condition is R.id=S.id. To achieve an early stop, the input data streams are sorted according to local scores $g_1(R)$ respectively $g_2(S)$. In the example of figure 6, after reading the third tuple of R and S the join can be early finished. The two circled results of the three matches are the top-2 tuples based on the ranking function. All other potential join combinations can never contribute to the top-2. An early stop after reading three tuples from R and only two tuples from S is not possible although two join combinations are found. This is because the upper bound T is still 25 and thus exceeds the smallest rank of top-k tuples seen so far, which is 23.

Although the idea works well for single top-k queries, however it is not directly applicable in the context of multiple independent ranking criteria. The problem is that in general we have no ordering of R and S for which we can determine an upper bound for all local scores of the different ranking functions.

In the special case that the ranking functions differ only in their local scores $f_i(R, S) = f(g_{1,i}(R), g_{2,i}(S)),\ 1 \leq i \leq n$ and the local scores have the form $g_{1,i}(R) = \alpha_{1,i} \cdot R.A_1 + \alpha_{2,i} \cdot R.A_2 + \ldots$ and $g_{2,i}(S) = \beta_{1,i} \cdot S.B_1 + \beta_{2,i} \cdot S.B_2 + \ldots$ we can extend the idea from [10] to multiple top-k queries.

Therefore we define a global score function for R $\bar{g}_1(R) = \max_{1 \leq i \leq n} \{\alpha_{1,i}\}|R.A_1| + \max_{1 \leq i \leq n} \{\alpha_{2,i}\}|R.A_2| + \ldots$. The global score function $\bar{g}_2(S)$ for S is analogous. It is easy to see that global scores $\bar{g}_1(R), \bar{g}_2(S)$ are always larger or equal than the respective local scores $g_{1,i}(R),\ g_{2,i}(S)$.

The tuples from tables R and S are processed in decreasing order according to their global scores $\bar{g}_1(R), \bar{g}_2(S)$. With the global sorting criteria we cannot assume that the largest local scores appear in the first tuples, so we must keep the maximum local scores $g_{1,i}^{max},\ g_{2,i}^{max}$ seen so far for each of the n ranking functions as we scan through R and S. Also the top-k tuples for each ranking seen so far are stored in heap structures H_i. Each heap has the smallest rank min_{H_i} of the particular ranking on top.

Let be x the current tuple from R and y the current tuple from S. Then $T_i = max\{f(g_{1,i}^{max}, \bar{g}_2(y)), f(\bar{g}_1(x), g_{1,i}^{max})\}$ is an upper bound for the ranks of future join combinations of ranking i. We can stop early if the condition $min_{H_i} \geq T_i$ holds for all rankings. In the experiment section we show that the proposed global sorting criteria has a negative effect on the performance of the algorithm because the linear combination of many attributes slowly pushes the upper boundaries to the lower values and thus delaying the early stops.

5.2. Early Stop in Hash Joins

The global orderings of R and S are suboptimal for most of the ranking functions. To avoid that disadvantage we can extend the hash-join principle to compute multiple top-k rankings and make use of early stops as much as possible. In this subsection we do not assume any special structure of the local scores $g_{1,i}(R)$ and $g_{2,i}(S)$ as well as the $f_i(\cdot, \cdot)$ might be different but monotonic.

The basic idea of the hash join extension is to partition the incoming (and not necessarily sorted) data stream of one join partner (in general the smaller table, say R) according

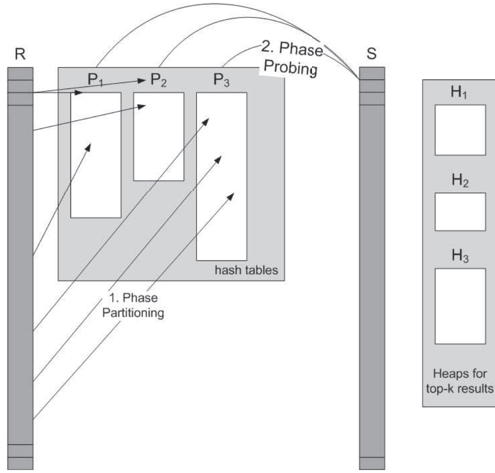


Figure 7. Evaluation multiple top-k with hash join

to the local scores $g_{1,i}(R)$ (and not according to the join attribute). Therefore we need n hash tables of size l_i, one for each local score.

Our hash-join for multiple top-k queries process R iteratively, with two phases per iteration. In a first phase we select the top-l_i tuples of R according to the local scores $g_{1,i}(R)$ into the hash tables P_i. For this we use the MRANK()-operator for simple multiple top-k queries (see section 4). Figure 7 illustrates an example with the three top-k ordering functions. Note that a single tuple may be placed into more than one hash table. As all hash tables fit in main memory this phase requires one scan over R.

In a second probing phase, all tuples of the join partner S are probed against the hash entries. If a join partner is found in hash table P_i, the joined tuple is inserted into the corresponding top-k heap structure H_i (used already within the MRANK()-operator) if the ith rank function of the combined tuple yields a value larger than the smallest top-k value for this ranking seen so far. The probing phase requires one scan over S. Then the hash tables are emptied and in the next iteration the next l_i tuples are filled into the hash tables.

The two phases are repeated until all entries of table R are handled once in each hash table P_i or the computation of all top-k values stops early. For early stops we maintain for each ranking an individual upper bound $T_i = f_i(min\{g_{1,i}(P_i)\}, max\{g_{2,i}(S)\})$. Note that the tuples of R are put into the hash table P_i in decreasing order according to $g_{1,i}()$. The maximal local score $g_{2,i}(S)$ can be determined during the probing phase of the first iteration. If the probing phase of the first iteration is not finished we use the maximum seen so far.

It is worth mentioning that each local partition P_i holds the complete tuple such that the final result can be computed without any further effort. An important effect on the performance of the algorithm have the cardinalities of the hash tables P_i. Given an amount C of main memory (in number of tuples) we determine the cardinalities $l_i = |P_i|$ as follows:

$$|P_i| = \frac{|C|}{\sum_{i=1}^{n} k_i} \cdot k_i$$

In this case the partitioning of the main memory depends only on the local limits of the multiple rankings. The runtime of the hash join for multiple top-k rankings is $O(iter \cdot (|R| + |S|))$, where $iter$ is the number of iterations. In the experimental section we show that in most cases the number of iteration is quite low, because the individual upper bounds T_i are quite tight.

5.3. Summary

In this section, we outlined two alternatives to push-down the limitation of multiple ranks into join operators. The first idea is based on the proposal of [10]. This only suitable if the sorting expressions have special structure and are highly correlated. For the general case of arbitrary sorting expressions, we propose a solution based on the hash join technique, which is expected to run faster than the sort-merge join approach.

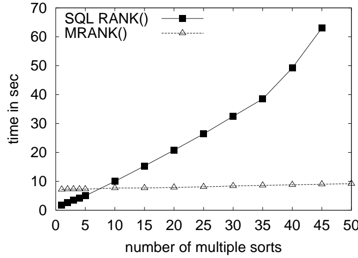
6. Experiments

We conducted multiple experiments of our MRANK operator to demonstrate the benefit in multiple situations, i.e. with different implementations and different parametric environments. All experiments were carried out on a Linux machine with an AMD Athlon XP 3000+ CPU and 1.5 GB main memory. The following subsections describe different scenarios based on single table expressions and – most important – in combination with joins.

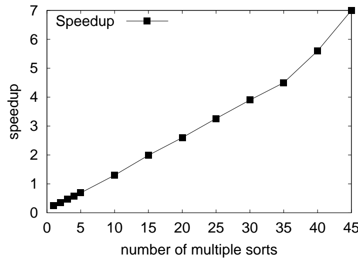
6.1. Simple Queries with Multiple Ranks

In this section we present our experiment results, which are based on simple queries. The real data we used come from the UCI KDD Archive and contain 32-dimensional color histograms of 66.615 images. For this experiments we used the commercial DB2 V8.1 database system. We also implemented a prototype of our MRANK-operator as C++ client on top of DB2, which has to read the data over an ODBC connection. So our MRANK-operator had a much slower access to the data as the system itself.

We simulated an image search application based on complex similarity search queries. For the experiment we varied the number of query points in the query point set Q from 1



(a)



(b)

Figure 8. Experimental comparison: images

to 45. Each query point translates to a separate ranking criteria. The limit parameter was set to $k = 20$, which means that each query point the 20 best matching objects were returned. The results are shown in figure 8. The MRANK implementation performs better than the database systems original implementation when more than 5 query points (different ranking functions) are used. We argue that the minor performance for small query point sets (1-5 points) is caused by the top of database implementation of our prototype. In case of 45 query points the MRANK-operator is 7 times faster than the database system. As the query point set and the limit k were reasonably small the main memory approach was used only.

6.2. Join Queries with Multiple Ranks

We proposed two algorithms to optimize multiple ranks over joins. The first algorithm is an extension of [10] considering multiple top-k ranking, which have a special form (linear). Ilays et. al. [10] showed that the rank join operator outperforms existing methods in database today. Our approaches performs also better than the today's database operators, because we avoid multiple sorts after joining R and S. Therefore we concentrate on analyzing and comparing our two proposed algorithms for joins in this section.

In the first experiment we compared the run time of the two approaches. The result is shown in figure 9. For this experiments we generated relations where the local scores are independently from the join condition and varied the number of multiple top-k ordering functions ($k = 10$). The relation R contains $50,000$ and S $150,000$ tuples. With a join selectivity of 0.01, $75,000,00$ join combinations exists. All temporary data structures are kept in main memory. For the hash join we limited the main memory space for all hash tables to $1,000$ tuples and respectively to $5,000$ tuples. We distributed the available space proportional to individuals hash tables. With increasing number of ranking functions the sizes of the individual hash tables decrease.

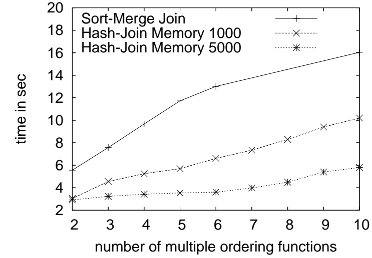


Figure 9. Comparing Sort-Merge and Hash Join approach

The hash join outperforms the sort merge approach, because the hash join does not sort the relations according global score functions. Furthermore, the upper bounds for early stop condition of the sort merge join become less tight as the number of ranking functions increases. Beyond a certain number of rankings all tuples of R and S have to be processed (see figure 10). A second observation from figure 9 is that the hash join gets faster when more main memory is available because of fewer scans of S.

That fact is further investigated in the second experiment. It shows the effect of the main memory on the performance of the algorithm. We varied the number of tuples in R and generated S, such that each tuple in R had 5 join partners in S. In the experiments we computed ten top-k ranking function, with $k = 20$. Figure 11(a) shows the result for main memory space of $1,000$ and $5,000$ tuples for the hash tables. The figure shows clearly, that the hash join can utilize the larger main memory very effectively.

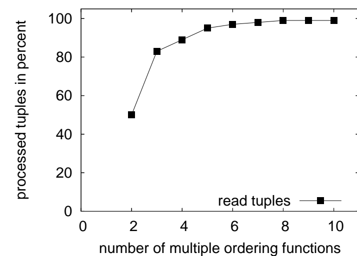


Figure 10. Processed tuples, Sort-Merge Join

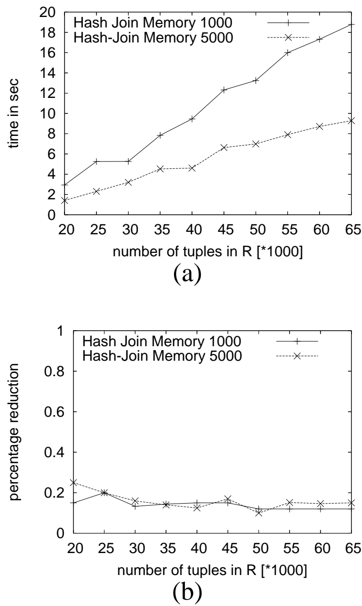


Figure 11. Hash Join

Our hash join approach extents the classic hash technique to join two relations considering multiple ranking functions. The classic hash join technique puts a fraction of the tuples from the smaller relation in the main memory. With a scan over S all tuples of S are probed against the tuples in the main memory hash tables. The amortized complexity is $O(R + \frac{C}{|R|} \cdot S)$, because the relation R have to be read only once. In our approach we invest more time to built up the hash tables but can reduce on the other hand the number of scans over S because of early stops. Figure 11(b) presents the percentage reduction of the number of scan over S compared to the classic hash join with the same amount of main memory.

7. Conclusion

In this paper, we analyzed the problem of supporting multiple top-k queries from a relational database engine perspective. We proposed a minimal SQL extension to ease the specification of multiple rankings within one SQL query and gave some ideas of applications which can benefit from it. Additionally, we proposed a variant of the well-known hash-join strategy which enables an early pruning of potential join candidates. Finally, we demonstrated the feasibility of our approach with a variety of different experiments. With our proposed SQL extension of ORDERING SET and column wise limitation in combination with an optimized implementation, we are convinced that this technology pushes the envelope and makes relational database technology more applicable for a huge range of data-intensive applications.

References

[1] Nicolas Bruno, Suraji Chaudhuri, and Luis Gravano. Top-k selection queries over relational databases: Mapping strategies and performance evaluation. *ACM Trans. Database Syst.*, 27(2):153–187, 2002.

[2] Nicolas Bruno, Luis Gravano, and Amélie Marian. Evaluating top-k queries over web-accessible databases. In *Proceedings of the 18th International Conference on Data Engineering*, pages 369–, 2002.

[3] Michael J. Carey and Donald Kossmann. On saying ”enough already!” in sql. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 219–230. ACM Press, 1997.

[4] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an sql query engine. In *Proceedings of 24rd International Conference on Very Large Data Bases*, pages 158–169. Morgan Kaufmann, 1998.

[5] Surajit Chaudhuri and Luis Gravano. Evaluating top-*k* selection queries. In *Proceedings of 25th International Conference on Very Large Data Bases*, pages 397–410, 1999.

[6] Chung-Min Chen and Yibei Ling. A sampling-based estimator for top-*k* query. In *Proceedings of the 18th International Conference on Data Engineering*, pages 617–629, 2002.

[7] Donko Donjerkovic and Raghu Ramakrishnan. Probabilistic optimization of top n queries. In *VLDB'99, Proceedings of 25th International Conference on Very Large Data Bases, September 7-10, 1999, Edinburgh, Scotland, UK*, pages 411–422. Morgan Kaufmann, 1999.

[8] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 2001.

[9] Vagelis Hristidis, Nick Koudas, and Yannis Papakonstantinou. Prefer: a system for the efficient execution of multi-parametric ranked queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 259–270. ACM Press, 2001.

[10] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. In *Proceedings of 29th International Conference on Very Large Data Bases (VLDB'03,September 9-12, 2003, Berlin, Germany)*, pages 754–765, 2003.

[11] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[12] Deok-Hwan Kim and Chin-Wan Chung. Qcluster: relevance feedback using adaptive clustering for content-based image retrieval. In *Proceedings of the 2003 ACM SIGMOD international conference on on Management of data*, pages 599–610, 2003.

[13] Giedrius Slivinskas, Christian S. Jensen, and Richard T. Snodgrass. Bringing order to query optimization. *SIGMOD Rec.*, 31(2):5–14, 2002.